Holistic System Design for Deterministic Replay

by

Dongyoon Lee

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2013

Doctoral Committee:

Assistant Professor Satish Narayanasamy, Chair
Professor Peter M. Chen
Associate Professor Jason N. Flinn
Professor Stéphane Lafortune

2013

To my family who has shaped my life:
My wife, Songyi Park
My father, Jongwoo Lee
My mother, Soonjung Kim
My daughter, Joyce Dabin Lee
My son, Justin Hyunbin Lee

A C K N O W L E D G M E N T S

First and foremost, praises and thanks to God, the Almighty, for His blessings throughout my whole life and my research work. This dissertation is not the product of my effort alone. There are many people with whom I am indebted over the last few years of graduate study.

I am forever grateful to my advisor Satish Narayanasamy for giving me the opportunity to do research with him and providing invaluable guidance throughout this research. His patience, guidance, enthusiasm, and effort have deeply inspired me and become essential to the birth of this thesis. It was a great privilege and honor to work and study under his guidance.

I would like to express my profound gratitude to my committee, Peter Chen, Jason Flinn and Stéphane Lafortune, for helping me shape this thesis. I especially thank Peter and Jason for their patience and guidance in research projects. Many of the research work in this thesis were developed in collaboration with them.

I would like to thank my colleagues at University of Michigan. Jie Yu and Abhayendra Singh spent a lot of time with me in office while discussing research idea. Benjamin Wester helped me learn kernel programming and debug Respec. Kaushik Veeraraghavan gave me invaluable advice not only in research but also in job search. I thank Jessica Ouyang and David Devecsery for helping make my work better through discussion and collaboration. I would also like to deeply thank Zijiang Yang at Western Michigan University. He provided me with the opportunity to work on SMT-based program analysis. Mahmoud Said also helped implementing the SMT-based algorithm that we used for Rosa.

I would like to thank my family in Korea. I am sincerely thankful to my father, Jongwoo Lee, and mother, Soonjung Kim, for their unstinting support in my graduate study; my brother, Dongmook Lee, for being a constant source of cheer; and Songyi's parents for encouraging words, all of which make all the things happen.

Lastly, I would like to especially thank my wife, Songyi Park, for her love and endless prayers to complete this research work. The sacrifice she made throughout my years in Ann Arbor are simply ineffable. Songyi, this thesis is dedicated to you. I am very much thankful to my daughter, Joyce Lee, and my son, Justin Lee, for being the greatest happiness and joy of my life. May the Almighty God richly bless all of you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Holistic System Design for Deterministic Replay

by

Dongyoon Lee

Chair: Satish Narayanasamy

Deterministic replay systems record and reproduce the execution of a hardware or software system. While it is well known how to replay uniprocessor systems, it is much harder to provide deterministic replay of shared memory multithreaded programs on multiprocessors because shared memory accesses add a high-frequency source of non-determinism. This thesis proposes efficient multiprocessor replay systems: *Respec*, *Chimera*, and *Rosa*.

Respec is an operating-system-based replay system. Respec is based on the observation that most program executions are data-race-free and for programs with no data races it is sufficient to record program input and the happens-before order of synchronization operations for replay. Respec speculates that a program is data-race-free and supports rollback and recovery from misspeculation. For racy programs, Respec employs a cheap runtime check that compares system call outputs and memory/register states of recorded and replayed processes at a semi-regular interval.

Chimera uses a sound static data race detector to find all potential data races and instrument pairs of potentially racing instructions to transform an arbitrary program to make it data-race-free. Then, Chimera records only the non-deterministic inputs and the order of synchronization operations for replay. However, existing static data race detectors generate excessive false warnings, leading to high recording overhead. Chimera resolves this problem by employing

a combination of profiling, symbolic analysis, and dynamic checks that target the sources of imprecision in the static data race detector.

Rosa is a processor-based ultra-low overhead (less than one percent) replay solution that requires very little hardware support as it essentially only needs a log of cache misses to reproduce a multiprocessor execution. Unlike previous hardware-assisted systems, Rosa does not record shared memory dependencies at all. Instead, it infers them offline using a Satisfiability Modulo Theories (SMT) solver. Our offline analysis is capable of inferring interleavings that are legal under the Sequentially Consistency (SC) and Total Store Order (TSO) memory models.

CHAPTER 1

Introduction

A deterministic replay system records and reproduces the execution of a hardware or software system. The ability to reproduce an execution can be used to improve systems along many dimensions, including reliability, security, and q. For example, deterministic replay is an efficient way to keep the state of a backup synchronized with the state of a primary machine [17]; it can be used to parallelize or offload heavyweight analysis from production machines [55, 21]; it can be combined with minor perturbations to diagnose or avoid faults [62, 72]; it can enable detailed analysis for forensics [24] or computer architecture research [50, 83]; and it can provide the illusion of reverse execution and time-travel debugging [69, 39]. In the recent past, Microsoft, Marie, and Intel have realized this need and have produced replay tools such as iDNA [13], ReTrace [83], and PinPlay [61].

The general idea behind deterministic replay is to log all non-deterministic events during recording and reproduce these events during replay. Deterministic replay for uniprocessors can be provided at low overhead because non-deterministic events occur at relatively low frequencies (e.g., interrupts or data from input devices and clocks), so logging them adds relatively little overhead.

Unfortunately, it is much harder to provide deterministic replay of shared memory multithreaded programs on multiprocessors because shared memory accesses add a high-frequency source of non-determinism. A variety of software-based approaches have been proposed to reproduce this non-determinism by logging a precise order of shared memory accesses, but these approaches are prohibitively slow for many parallel applications [25, 61]. This severely limits the use of deterministic replay tools because they cannot be used in production systems or even for in-house testing. Other software approaches that target efficiency only support race-free programs [63, 57] or do not guarantee deterministic replay [60, 5].

On the other hand, existing hardware-assisted solutions [81, 49, 34, 47, 48, 19, 32, 82] require invasive changes to the coherence mechanism (one of the most hard-to-verify components in a multiprocessor) and also require complex hardware structures for optimizing the size of memory order logs. Another important limitation of many previous solutions is that they guarantee replay solely for sequentially consistent (SC) executions. However, most modern processors

support only a relaxed memory model as SC disallows many common optimizations. For instance, SPARC and x86-based processors support variants of the Total Store Order (TSO).

We observe that previous approaches think of deterministic replay as producing a replayed execution that is identical in every way to the original execution, leading to high performance overhead in software systems and complex processor design in hardware systems.

The key insight in this thesis is that completely identical replay is both impractical and unnecessary. It is impractical to replay low-level details of the original execution like the detailed behavior of the microarchitecture or the electrical behavior of the executing circuitry. More importantly, it is unnecessary to replay these low-level details, since most proposed uses of deterministic replay require only higher-level states to be reproduced, such as the program state of the process and system outputs. We exploit our insight by proposing deterministic replay systems that do not record the precise order of shared memory accesses to avoid performance overhead during recording, but still provide useful replay guarantee.

Another insight in this thesis is that it is possible to reduce the cost of detecting and logging shared-memory dependencies by constraining the original execution in such a way that threads interleave each other at the larger granularity rather than an instruction. This thesis provides a solution that forces threads to interleave at the coarsened granularity, and records coarse-grained shared-memory dependencies for efficient logging, but still preserves parallelism in common cases for performance.

The following sections briefly introduce three novel deterministic replay systems proposed by this thesis: *Respec*, *Chimera*, and *Rosa*; and describe how we exploit our insight in enabling efficient deterministic replay of multiprocessor systems.

1.1 Respec

Respec is a low-overhead software-only system that is implemented in Linux operation system kernel. Respec aims to support online deterministic replay in which the recorded and replayed processes execute concurrently. The online replay has been demonstrated to play an important role in providing safely guarantees in fault tolerance systems and decoupled runtime checks. Past software solutions, however, could not be used for online replay as they either suffer from high recording overhead or do not guarantee deterministic replay.

Respec is based on the observation that most program executions are data-race-free and it is sufficient for data-race-free programs to record the happens-before order of synchronization operations. Thus, instead of paying huge overhead to detect and log rare data races, Respec speculates that a program is data-race-free and supports rollback and recovery from misspeculation.

For racy programs, Respec employs a cheap runtime check that compares system call out-

puts and memory/register states of recorded and replayed processes at a semi-regular interval. This stems from our first insight that completely identical replay is both impractical and unnecessary, and it is sufficient for many replay uses to guarantee that replayed execution produces the same output and the final state as recorded execution. We call this relaxed, but sufficient guarantee, *external determinism*, and leverage external determinism when we evaluate whether the replayed run matches the original run by comparing only the output via system calls and the final program state. Reducing the scope of comparison helps reduce the frequency of failed replay and subsequent rollback. This technique results in low recording and replay overhead for the common case of data-race-free execution intervals and still ensures correct replay for execution intervals that have data races.

The experimental results show only 55 percent overhead on average to record and replay programs with four threads. To the best of our knowledge, this work is the first software-only system which can support efficient online replay of multi-threaded programs as other software tools require a prohibitive amount of performance overhead.

1.2 Chimera

Respec requires a redundant execution on spare cores to enable low-overhead deterministic replay and therefore it incurs about two times the throughput cost. This thesis introduces *Chimera*, which is the first approach that uses static program analysis to build an efficient software solution without redundant execution.

Similar to Repsec, Chimera stems from the observation that it is easy to provide deterministic multiprocessor replay for data-race-free programs (one can just record non-deterministic inputs and the order of synchronization operations). Chimera uses a sound static data race detector to find all potential data-races and instrument pairs of potentially racing instructions to transform an arbitrary program to make it data-race-free. However, existing static data race detectors generate excessive false warnings because static analysis cannot reason about non-mutex happens-before relations (e.g., barriers), and a sound pointer analysis is necessarily conservative and thus too imprecise. Our experiments show that logging all potential data race instructions reported by static data race detector slows the execution down by more than 50 times.

Chimera resolves this problem by employing a combination of profiling, symbolic analysis, and dynamic checks that target the sources of imprecision in the static data race detector. This hybrid program analysis allows us to protect potential data races at the higher granularity (such as a loop or a function) while still preserving parallelism in common cases. Instrumenting potential data races at a loop or function granularity subsequently forces runtime to constrain possible thread interleavings because it prevents fine-grained interleaving between threads, resulting in low overhead in detecting and logging shared-memory dependencies.

By drastically reducing the overhead of recording potential data-races, we show that Chimera is an attractive recording solution, especially for server (e.g., Apache) and desktop applications (e.g., pbzip), for which Chimera's recording overhead is only about 2.4 percent on average.

1.3 Rosa

Processor support could enable us to build ultra-low overhead (less than one percent) replay solutions. However, hardware solutions should be complexity-effective enough that processor vendors are encouraged to include a deterministic replay feature in the next-generation processors. This thesis proposes *Rosa*, which requires very little hardware support for replay as it essentially only requires a log of cache misses to reproduce a multiprocessor execution. We show that processor support for logging data fetched on cache misses is sufficient for replaying each thread independently.

Unlike previous recording solutions, Rosa does not record shared memory dependencies at all. Instead, it infers them offline, before replay, using a Satisfiability Modulo Theories (SMT) solver. The order of shared memory accesses reconstructed by the SMT solver could be different from the original execution. This implies that Rosa does not guarantee full-level replay fidelity, which is the key to enabling simple hardware design. However, cache miss logging guarantees that each instruction in the replayed execution reads and writes the same values as the recorded execution.

Our offline analysis is capable of inferring interleavings that are legal under both Sequential Consistency (SC) and Total Store Order (TSO) memory models. TSO is the most common consistency model implemented in modern processors, but is not supported in many previous solutions.

1.4 Roadmap

The rest of this thesis is organized as follows:

Chapter 2 presents an overview of background research in deterministic replay. We describe and categorize previous software and hardware solutions for deterministic replay.

Chapter 3 presents *Respec*, an operating-system-based solution for multiprocessor deterministic replay. This chapter includes the design, implementation, and evaluation of Respec.

Chapter 4 describes the design, implementation, and evaluation results of Chimera including how Chimera uses static/dynamic program analysis to build an efficient software replay solution.

Chapter 5 describes *Rosa*, a lightweight hardware-assisted replay solution using cache-miss logging and offline analysis. We describe the hardware design and offline analysis of Rosa,

followed by evaluation.

Chapter 6 concludes the thesis. We summarize our contributions and discuss future work.

CHAPTER 2

Background and Related Work

With the advent of multiprocessor systems, it is now the role of the programmers to explicitly expose parallelism and take advantage of parallel computing resources. However, parallel programming is inherently complex as programmers have to reason about all possible thread interleavings. This problem is compounded by the fact that shared-memory multiprocessors are non-deterministic in the sense that a given input is not guaranteed to produce the same output across different executions. Therefore, to sustain the growth in computing that we have enjoyed over the last few decades, it is critical that we provide programmers with solutions that can drastically simplify parallel programming.

A deterministic replay system, which logs all non-deterministic events during recording and reproduces these events during replay, has been shown to be useful in building many such tools as time-travel debugger and fault tolerant systems by overcoming the inherent non-determinism in multiprocessor systems. Along this line, this thesis proposes synergistic solutions that exploit the strengths of various layers in a computing stack from static analysis to processor architecture for enabling deterministic replay of multiprocessor systems.

As another direction to the address non-determinism issue in traditional multiprocessor systems, researchers have proposed deterministic execution systems, in which for a given input a system is guaranteed to produce the same interleaving between threads, thus in turn producing the same results.

Sections 2.1 and 2.2 describe previous software and hardware replay systems respectively, and highlight the contributions of our proposed replay solutions: Respec, Chimera, and Rosa. Then, Section 2.3 discusses the relationship between deterministic execution and deterministic replay and briefly refers to previous deterministic execution systems.

2.1 Software Deterministic Replay Systems

Early software-based solutions are mostly limited to support replay only, on an uniprocessor system. Deterministic replay for a uniprocessor execution can be provided at low cost as it does

not need to track shared-memory dependencies.

IGOR [27], one of the earliest recorders, uses copy-on-write checkpointing support in the operating system to record and reproduce an intermediate state of a process. In addition to checkpointing support, to ensure deterministic replay of a program from a particular state, it is also necessary to record non-deterministic system input such as interrupts, DMA, and also the values of any non-deterministic instructions such as the x86 RDTSC (ReaD TimeStamp Counter) instruction. These events can be recorded in any of the layers in the software system stack. Systems like Hyperviser [17], Boothe [16], and Flashback [69] make it possible for the operating system to record and replay the non-deterministic events. DejaVu [20] and jRapture [71] record most (but not all) of the non-deterministic system events by using the Java Virtual Machine (JVM). ReVirt [24] and ReTrace [83] use support in the virtual machine monitor that interfaces between the guest and host operating systems. Unlike Respec, none of these systems support multiprocessor replay, because they cannot record and replay the non-deterministic order between shared memory accesses executed by concurrent threads.

Systems like ReVirt [24] and DejaVu [20] support replay of multithreaded programs on a uniprocessor system by deterministically replaying the thread schedules. However, replay of a multithreaded program on a multiprocessor has remained a difficult problem. One of the first systems to address this problem is InstantReplay [40]. It monitors every memory access to a shared object to record the order in which different threads accessed it. Recent systems such as Intel's PinPlay [50, 61] and Microsoft's iDNA [13] also record every memory access to enable multiprocessor replay. But, monitoring every memory access is expensive (iDNA [13], for instance, is about 5–15 times slower than the native execution). Instead of monitoring every memory access, SMP-ReVirt [25] uses memory protection bits to detect all the shared memory dependencies and recorded the memory order. But, handling a memory protection fault for every shared memory dependency is also inefficient (up to nine times slower).

Instead of recording the order of all shared memory accesses, RecPlay [63] and JaRec [30] monitor just the synchronization operations and record their order of execution. This approach only ensures deterministic replay of a program up until the first data race, which limits the use of a replayer in many ways. For example, while debugging using a replayer, a programmer might want to understand the after effects of a data race bug in order to triage it, which is not possible with RecPlay. After finding a data race bug, a tester might not want to wait for the developer to fix it before he/she could carry on with further tests. Also, for continuously checking the correctness of production runs it is necessary to replay past the first data race. In fact, most real world applications contain benign data races [52]. For such applications, a replay tool is most useful only if it can replay past the benign data races.

Recent work on software-only replay systems achieved relatively low logging overhead by not eagerly recording shared-memory dependencies. ODR [5] and PRES [60] record less infor-

mation than necessary to reproduce the same interleaving between threads. Instead of detecting and recording shared memory dependencies at runtime, they perform offline searches to reconstruct thread interleavings. This class of systems shows notable performance improvement during recording, but the offline search is not guaranteed to succeed in a bounded amount of time (failing to find a solution in some experiments). In some cases, the first deterministic replay may take a prohibitively long time when the search does not scale. However, subsequent replays will have low overhead, because a solution will have previously been found.

On the other hand, Respec, proposed in this thesis, enables guaranteed deterministic replay of shared memory multiprocessors at a low cost. As a concurrent work with ODR [5] and PRES [60], Respec shares a similar intuition that it is not necessary to record precise shared-memory dependencies for replay. However, Respec does not require offline search, so Respec can be used for online replay in which record and replay are performed concurrently and efficiently.

As another line of work, LEAP [35] uses static escape analysis to provide efficient multiprocessor replay. LEAP improves the efficiency of a recorder by monitoring accesses to only shared variables that are determined using a static escape analysis. LEAP also improves efficiency by ignoring accesses to variables that are immutable after initialization. Monitoring and logging accesses to all mutable shared variables determined using a conservative static analysis can be quite expensive at runtime. LEAP can slow down a program by more than two times in the average case and six times in the worst case [35], which is higher than Respec (1.5x).

This thesis also proposes Chimera, which leverages hybrid program analysis that combines static data race detection, profiling, and symbolic analysis. The technique allows us to enable efficient software-only multiprocessor replay.

2.2 Hardware Deterministic Replay Systems

Recent hardware-assisted replay systems [81,49,34,47,48,19,32,82] show that recording shared memory dependencies with hardware support only incur less than one percent of performance overhead. Therefore, they all focus on reducing the log size and the amount of hardware states required to detect and log shared-memory dependencies. While they succeeded in reducing the cost of hardware real estate, the hardware complexity of those solutions is still so high that processor manufacturers have been reluctant to adopt them.

To illustrate the complexity of a shared-memory dependency logger, we now discuss two recently proposed hardware solutions, DeLorean [47] and ReRun [34].

DeLorean assumes support for BulkSC [18]. It divides a thread's execution into what are called *chunks*. The underlying BulkSC mechanism ensures that each chunk is executed atomically. Given this execution environment, each core in DeLorean just needs to record the size

of each chunk that it executes. Also, a global arbiter records the total order between chunks executed in different cores. DeLorean drastically reduces the log size required for recording shared-memory dependencies. However, it introduces additional hardware complexity for supporting BulkSC [18], a global arbiter for logging the order between chunks, and support for logging chunk sizes.

ReRun [34] forms episodes and records their sizes along with a total order between them. Similar to a chunk in DeLorean, an episode is also a sequence of instructions that appear to be atomic in an execution. But ReRun differs from DeLorean in implementation details, such as the conditions for terminating an episode. Before sending an invalidation acknowledgment or a data update message to any coherence request, a processor core in ReRun terminates its episode if it detects a read-write or a write-write conflict between the requesting access and one of its past memory accesses. An episode is also terminated when a cache block gets evicted, because the core would no longer receive any coherence message for the evicted cache block. Thus, ReRun guarantees the atomicity property for an episode.

ReRun is very efficient in terms of log size (about four bytes/kilo-instruction) and performance. However, it needs significant hardware support. Moreover, to support replay, Rerun also needs system support for recording program input. Capo [48] discusses the problem of interfacing software system support with a hardware-based record-and-replay system. Capo advocates software system support for recording program input, which requires a copy-on-write checkpoint mechanism and support for logging non-deterministic system events. But the performance cost of a software-based program input recording approach could lead to about a 20 to 40 percent performance loss, as shown in Capo [48].

Another important limitation of previous works is that most of them are limited to the SC memory model due to its simplicity. Supporting a relaxed consistency memory model remains a challenging problem. To the best of our knowledge, only a few replay systems support replay under a relaxed memory model. RTR [82] is the first to propose a solution for TSO. It proposes to dynamically detect loads that violate SC while executing on a TSO processor and then explicitly record their values. Detecting SC violation requires monitoring if a load memory location is modified between the time the load accesses the location and the time when all the preceding memory accesses have finished. In addition, RTR requires hardware support for detecting and logging shared-memory dependencies precisely. Instead, we propose to record Strata along with cache miss information. Strata require no communication between processor cores or changes to the coherence mechanism to record them. Instead, at fixed periodic intervals each processor core records their memory counts and outstanding stores in the store buffer. Thus the required modifications are "local" to a processor core. We believe that a local solution is simpler than a solution that require changes to the coherence mechanism, as coherence protocol design and verification is hard. Effectively, our solution reduces the hardware complexity of the recorder

by relaxing the precision required in recording shared-memory dependencies.

ReRun [34] also uses the same technique as RTR to detect potentially SC-violating loads. LReplay [19] records pending period information and supports some relaxed models such as the Godson-3 consistency by logging load instructions that violate SC. However, it only considers store atomic multiprocessor systems, which is not the case for the TSO memory model in x86 processors.

This thesis proposes Rosa, a hardware-assisted replay solution, which requires minimum hardware extension for replay and does not require modifying cache coherence protocols. Moreover, Rosa can support replay of both SC and TSO memory model systems.

2.3 Deterministic Execution Systems

Deterministic execution systems help programmers by ensuring that the thread interleaving observed is always the same for a given program and an input [22]. This approach obviates the need for recording the order of shared-memory accesses (unlike deterministic replay), but must still record any non-deterministic program input to reproduce an execution (like deterministic replay). If the goal is to reproduce a multiprocessor execution (e.g. for debugging or replication), then deterministic execution system has the benefit of saving log spaces. However, proposed deterministic execution systems, which artificially orchestrate thread interleavings and memory operations, have reported higher performance overhead compared to replay solutions.

While deterministic execution can be supported fairly efficiently for programs without data-races [57, 15], current software-only solutions for racy programs incur high overhead [11]. Efficiency can be improved by using custom hardware [22, 23, 33]; or by restricting the class of programs supported to fork-join parallelism [12] or shared-nothing address spaces [8]. Another approach is to isolate concurrent memory updates using virtual memory protection and deterministically merge the memory state of different threads at deterministic times [42]. While this approach guarantees determinism efficiently, it may result in program behaviors that are not consistent with the processor's memory consistency model.

This thesis proposes deterministic replay (not deterministic execution) solutions. However, Chimera, proposed in this thesis, transforms a program into an equivalent data-race-free program under the new set of synchronization operations. We believe that future work could leverage this property to design an efficient software only solution for deterministic execution as well.

CHAPTER 3

Respec: Operating System Support for Multiprocessor Replay

This chapter describes Respec, a new way to support deterministic replay of a shared memory multithreaded program execution on a commodity multiprocessor. Respec's goal is to provide fast execution in the common case of data-race-free execution intervals and still ensure correct replay for execution intervals with data races (albeit with additional performance cost). Respec targets *online* replay in which the recorded and replayed processes execute concurrently.

Respec is based on two insights. First, Respec can optimistically log the order of memory operations less precisely than the level needed to guarantee deterministic replay, while executing the recorded execution speculatively to guarantee safety. After a configurable number of misspeculations (that is, when the information logged is not enough to ensure deterministic replay for an interval), Respec rolls back execution to the beginning of the current interval and re-executes with a more precise logger. Second, Respec can detect a misspeculation for an interval by concurrently replaying the recorded interval on spare cores and checking if its system output and final program states (architectural registers and memory state) matches those of the recorded execution. This is based on our insight that matching the system output and final program states of the two executions is sufficient for most applications of replay.

Respec works in the following four phases:

First, Respec logs most common, but not all, synchronization operations (e.g., lock and unlock) executed by a shared memory multithreaded program. Logging and replaying the order of all synchronization operations guarantees deterministic replay for the data-race-free portion of programs [63], which is usually the vast majority of program execution.

Second, Respec detects when logging synchronization operations is insufficient to reproduce an interval of the original run. Respec concurrently replays a recorded interval on spare cores and compares it with the original execution. Since Respec's goal is to reproduce the visible output and final program states of the original execution, Respec considers any deviation in system call output or program state at the end of an interval to be a failed replay. Respec permits the original and replayed execution to diverge during an interval, as long as their system output

and the program memory and register states converge by the end of that interval.

Third, Respec uses speculative execution to hide the effects of failed replay intervals and to transparently rollback both recorded and replayed executions. Respec uses operating system speculation [54] to defer or block all visible effects of both recorded and replayed executions until it verifies that these two executions match.

Fourth, after rollback, Respec retries the failed interval of execution by serializing the threads and logging the schedule order, which guarantees that the replay will succeed for that interval.

Our results show that Respec shared memory multiprocessor record and replay is efficient. For a combination of PARSEC and SPLASH-2 benchmarks, as well as the pbzip2, pfscan, aget and Apache applications, Respec adds only an average 18% overhead to execution time when two threads are replayed and 55% when four threads are replayed.

The rest of this chapter is organized as follows. Section 3.1 describes two design dimensions along which replay systems vary and the choices we made for Respec. Section 3.2 and Section 3.3 describes the design and implementation of Respec. Section 3.4 evaluates the performance of Respec for programs with and without data races. Section 3.5 briefly introduces Doubleplay extended from Respec to support offline replay, and presents other extension. Lastly, Section 3.6 concludes the chapter.

3.1 Replay Guarantee

Replay systems provide varying guarantees. This section discusses two types of guarantees that are relevant to Respec: fidelity level and online versus offline replay.

3.1.1 Fidelity level

Replay systems differ in their fidelity of replay and the resulting cost of providing this fidelity. One example of differing fidelities is the abstraction level at which replay is defined. Prior machine-level replay systems reproduce the sequence of instructions executed by the processor and consequently reproduce the program state (architectural registers and memory state) of executing programs [17, 24, 82]. Deterministic replay can also be provided at higher levels of a system, such as a Java virtual machine [20] or a Unix process [69], or lower levels of a system, such as cycle accuracy for interconnected components of a computer [66]. Since replay is deterministic only above the replayed abstraction level, lower-level replay systems have a greater scope of fidelity than higher-level replay systems.

Multiprocessor replay adds another dimension to fidelity: how should the replaying execution reproduces the interleaving of instructions from different threads. No proposed application

of replay requires the exact time based ordering of all instructions to be reproduced. Instead, one could reproduce data from shared memory reads, which, when combined with the information recorded for uniprocessor deterministic replay, guarantees that each thread executes the same sequence of instructions. Reproducing data read from shared memory can be implemented in many ways, such as reproducing the order of reads and writes to the same memory location, or logging the data returned by shared memory reads.

Replaying the order of dependent shared memory operations is sufficient to reproduce the execution of each thread. However, for most applications, this degree of fidelity is exceedingly difficult to provide with low overhead on commodity hardware. Logging the order or results of critical shared memory operations is sufficient but costly [25].

Logging higher-level synchronization operations is sufficient to replay applications that are race-free with respect to those synchronization operations [63]. However, this approach does not work for programs with data races. In addition, for legacy applications, it is exceedingly difficult to instrument *all* synchronization operations. Such applications may contain hundreds or thousands of synchronization points that include not just Posix locks but also spin locks and lock-free waits that synchronize on shared memory values. Further, the libraries with which such applications link contain a multitude of synchronization operations. GNU glibc alone contains over 585 synchronization points, counting just those that use atomic instructions. Instrumenting all these synchronization points, including those that use no atomic instructions, is difficult.

Further, without a way to correct replay divergence, it is incorrect to instrument only some of the synchronization points, assuming that uninstrumented points admit only benign data races. Unrelated application bugs can combine with seemingly benign races to cause a replay system to produce an output and execution behavior that does not match those of the recorded process. For instance, consider an application with a bug that causes a wild store. A seemingly benign data race in glibc's memory allocation routine may cause an important data structure to be allocated at different addresses. During a recording run, the structure is allocated at the same address as the wild store, leading to a crash. During the replay run, the structure is allocated at a different address, leading to an error-free execution. A replay system that allowed this divergent behavior would clearly be incorrect. To address this problem, one can take either a pessimistic approach, such as logging all synchronization operations or shared memory addresses, or an optimistic approach, such as the rollback-recovery Respec uses to ensure that the bug either occurs in both the recorded and replayed runs or in neither.

The difficulty and inefficiency of pessimistic logging methods led us to explore a new fidelity level for replay, which we call *externally deterministic replay*. Externally deterministic replay guarantees that (1) the replayed execution is indistinguishable from the original execution *from the perspective of an outside observer*, and (2) the replayed execution is a *natural*

execution of the target program, i.e., the changes to memory and I/O state are produced by the target program. The first criterion implies that the sequence of instructions executed during replay *cannot be proven to differ* from the sequence of instructions executed during the original run because all observable output of the two executions are the same. The second criterion implies that each state seen during the replay was able to be produced by the target program; i.e. the replayed execution must match the instruction-for-instruction execution of one of the possible executions of the unmodified target system that would have produced the observed states and output. Respec exploits these relaxed constraints to efficiently support replay that guarantees identical output and natural execution even in the presence of data races and unlogged synchronization points.

We assume an outside observer can see the output generated by the target system, such as output to an I/O device or to a process outside the control of the replay system. Thus, we require that the outputs of the original and replayed systems match. Reproducing this output is sufficient for many uses of replay. For example, when using replay for fail-stop fault tolerance [17], reproducing the output guarantees that the backup machine can transparently take over when the primary machine fails; the failover is transparent because the state of the backup is consistent with the sequence of output produced before the failure. For debugging [27, 39], this guarantees that all observable symptoms of the bug are reproduced, such as incorrect output or program crashes (reproducing the exact timing of performance bugs is outside our scope of observation).

We also assume that an outside observer can see the final program state (memory and register contents) of the target system at the end of a replay interval, and thus we require that the program states of the original and replayed systems match at the end of each replay interval.

Reproducing the program state at the end of a replay interval is mandatory whenever the program states of both the recording and replaying systems are used. For example, when using replay for tolerating non-fail-stop faults (e.g., transient hardware faults), the system must periodically compare the state of the replicas to detect latent faults. With triple modular redundancy, this comparison allows one to bound the window over which at most one fault can occur. With dual modular redundancy and retry, this allows one to verify that a checkpoint has no latent bugs and therefore is a valid state from which to start the retry.

Another application of replay that requires the program states of the original and replayed systems to match is parallelizing security and reliability checks, as in Speck [55]. Speck splits an execution into multiple epochs and replays the epochs in parallel while supplementing them with additional checks. Since each epoch starts from the program state of the original run, the replay system must ensure that the final program states of the original and replayed executions match, otherwise the checked execution as a whole is not a natural, continuous run.

Note that externally deterministic replay allows a more relaxed implementation than prior definitions of deterministic replay. In particular, externally deterministic replay does not guar-

antee that the replayed sequence of instructions matches the original sequence of instructions, since this sequence of instructions is, after all, not directly observable. We leverage this freedom when we evaluate whether the replayed run matches the original run by comparing only the output via system calls and the final program state. Reducing the scope of comparison helps reduce the frequency of failed replay and subsequent rollback.

3.1.2 Online versus offline replay

Different uses of deterministic replay place different constraints on replay speed. For some uses, such as debugging [27,39] or forensics [24], replay is performed after the original execution has completed. For these *offline* uses of replay, the replay system may execute much slower than the original execution [60].

For other uses of replay, such as fault tolerance and decoupled [21] or parallel checks [55], the replayed execution proceeds in parallel with the original execution. For these *online* uses of replay, the speed of replayed execution is important because it can limit the overall performance of the system. For example, to provide synchronous safety guarantees in fault tolerance or program checking, one cannot release output until the output is verified [43].

In addition to the speed of replay, online and offline scenarios differ in how often one needs to replay an execution. Repeated replay runs are common for offline uses like cyclic debugging, so these replay systems must guarantee that the replayed run can be reproduced at will. This is accomplished either by logging complete information during the original run [24], or by supplementing the original log during the first replayed execution [60]. In contrast, online uses of replay need only replay the run a fixed number of times (usually once).

Respec is designed for use in online scenarios. It seeks to minimize logging and replay overhead so that it can be used in production settings with synchronous guarantees of fault tolerance or program error checking. Respec guarantees that replay can be done any number times when a program is executing. If replay needs to be repeated offline, Respec could store the log in permanent storage. The recorded log would be sufficient for deterministically replaying race-free intervals offline. For offline replay of racy intervals, a replay search tool [5, 60, 41] could be used.

3.2 Design

This section presents the design of Respec, which supports online, externally deterministic replay of a multithreaded program execution on a multiprocessor.

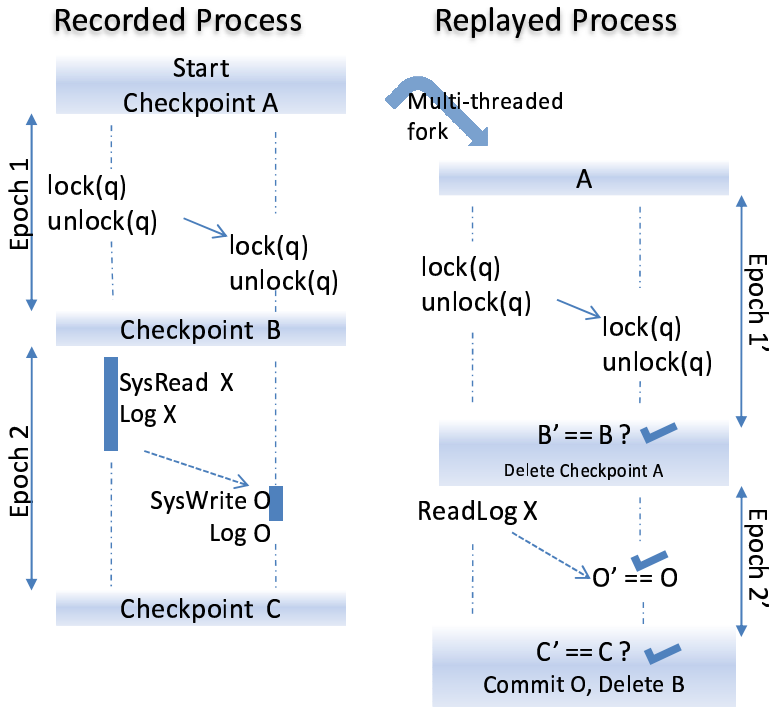

Figure 3.1: An execution in Respec with two epochs

3.2.1 Overview

Respec provides deterministic replay for one or more processes. It replays at the process abstraction by logging the results of system calls and low-level synchronization operations executed by the recording process and providing those logged results to the replayed process in lieu of re-executing the corresponding system calls and synchronization operations. Thus, kernel activity is not replayed.

Figure 3.1 shows how Respec records a process and replays it concurrently. At the start, the replayed process is forked off from the recorded process. The fork ensures deterministic reproduction of the initial state in the replayed process. Respec checkpoints the recording process at semi-regular intervals, called *epochs*. The replayed process starts and ends an epoch at exactly the same point in the execution as the recording process.

During an epoch, each recorded thread logs the input and output of its system calls. When a replayed thread encounters a system call, instead of executing it, it emulates the call by reading the log to produce return values and address space modifications identical to those seen by the recorded thread. To deterministically reproduce the dependencies between threads introduced by system calls, Respec records the total order of system call execution for the recorded process and forces the replayed process to execute the calls in the same order.

To reproduce non-deterministic shared memory dependencies, Respec optimistically logs just the common user-level synchronization operations in GNU glibc. Rather than enforcing

a total order over synchronization operations, Respec enforces a partial order by tracking the causal dependencies introduced by synchronization operations. The replayed process is forced to execute synchronization operations in an order that obeys the partial ordering observed for the recording process. Enforcing the recorded partial order for synchronization operations ensures that all shared memory accesses are ordered, provided the program is race free.

Replay, however, could fail when an epoch executes an unlogged synchronization or data race. Respec performs a *divergence check* to detect such replay failures. A naive divergence check that compares the states of the two executions after every instruction or detects unlogged races would be inefficient. Thus, Respec uses a faster check. It compares the arguments passed to system calls in the two executions and, at the end of each epoch, it verifies that the memory and resister state of the recording and replayed process match. If the two states agree, Respec commits the epoch, deletes the checkpoint for the prior epoch, and starts a new epoch by creating a new checkpoint. If the two states do not match, Respec rolls back recording and replayed process execution to the checkpoint at the beginning of the epoch and retries the execution. If replay again fails to produce matching states, Respec uses a more conservative logging scheme that guarantees forward progress for the problem epoch. Respec also rolls back execution if the synchronization operations executed by the replayed process diverge from those issued by the recorded process (e.g., if a replay thread executes a different operation than the one that was recorded) since it is unlikely that the program states will match at the end of an epoch.

Respec uses speculative execution implemented by Speculator [54] to support transparent application rollback. During an epoch, the recording process is prevented from committing any external output (e.g., writing to the console or network). Instead, its outputs are buffered in the kernel. Outputs buffered during an epoch are only externalized after the replayed process has finished replaying the epoch and the divergence check for the epoch succeeds.

3.2.2 Divergence Checks

Checking intermediate program state at the end of every epoch is not strictly necessary to guarantee externally deterministic replay. It would be sufficient to check just the external outputs during program execution. However, checking intermediate program state has three important advantages. First, it allows Respec to commit epochs and release system output. It would be unsafe to release the system output without matching the program states of the two processes. Because, it might be prohibitively difficult to reproduce the earlier output if the recorded and replayed processes diverge at some later point in time. For example, a program could contain many unlogged data races, and finding the exact memory order to reproduce the output could be prohibitively expensive. Second, intermediate program state checks reduce the amount of execution that must rolled back when a check fails. Third, they enable other applications such as

fault tolerance, parallelizing reliability checks, etc., as discussed in Section 3.1. Though intermediate program state checks are useful, they incur an additional overhead proportional to the amount of memory modified by an application. Respec balances these tradeoffs by adaptively configuring the length of an epoch interval. It also reduces the cost of checks by parallelizing them and only comparing pages modified in an epoch.

Respec's divergence check is guaranteed to find all instances when the replay is not externally deterministic with respect to the recorded execution. But, this does not mean that execution of an unlogged race will always cause the divergence check to fail. For several types of unlogged races, Respec divergence check will succeed. This reduces the number of rollbacks necessary to produce an externally deterministic replay.

First, the replayed process might produce the same causal relationship between the racing operations as in the recorded execution. Given that Respec logs a more conservative order between threads (a total order for system calls and even the partial order recorded for synchronization operations is stricter than necessary as discussed in Section 3.3.3.1), the replayed process is more likely to reproduce the same memory order.

Second, two racing memory operations might produce the same program state, either immediately or sometime in future, irrespective of the order of their execution. This is likely if the unlogged race is a synchronization race or a benign data race [52]. For example, two racing writes could be writing the same value, the write in a read-write race could be a silent write, etc. Another possibility is that the program states in the two processes might converge after a transient divergence without affecting system output. Note that a longer epoch interval would be beneficial for such cases, as it increases the probability of checking a converged program state.

Figure 3.2 shows an epoch with an unlogged synchronization race that does not cause a divergence check to fail. The second thread waits by iterating in a spin loop until the first thread sets the variable `x`. Because there is no synchronization operation that orders the write and the reads, the replayed process might execute a different number of reads than the recorded process. However, the program states of the replayed and recorded processes eventually converge, and both processes would produce the same output for any later system call dependent on `x`. Thus, no rollback is triggered.

However, for harmful races that should happen rarely, Respec's divergence check could trigger a rollback. Figure 3.3 shows an epoch with a harmful data race where the writes to a shared variable `y` are not ordered by any logged synchronization operation. The replayed execution produces a memory state different from the recorded execution. This causes the divergence check to fail and initiate a recovery process. This example also shows why it is important to check the intermediate program states before committing an epoch. If we commit an epoch without matching the program states, the two executions would always produce different output

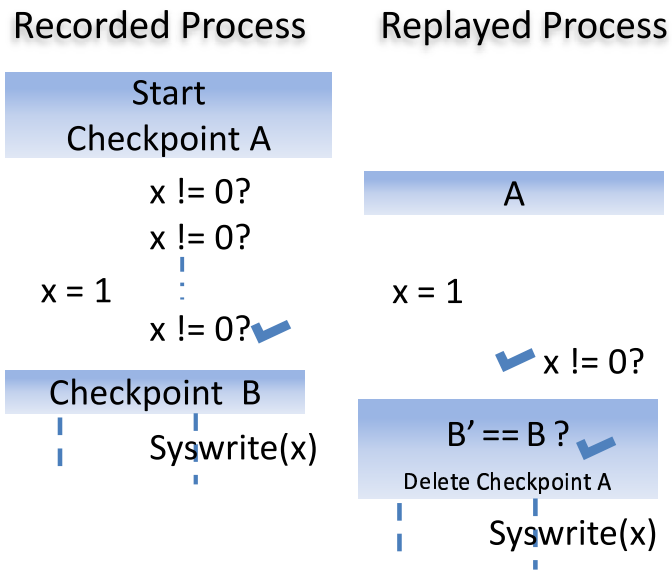

Figure 3.2: A race that produces the same program state irrespective of the order between the racing memory operations. Although the number of reads executed by the replayed process is different from the recorded process causing a transient divergence, the executions eventually converge to the same program state.

at the system call following the epoch. Yet, the replay system could not roll back past the point where the executions diverged in order to retry and produce an externally deterministic replay.

3.3 Implementation

3.3.1 Checkpoint and multithreaded fork

Rollback and recovery implementations often use the Unix copy-on-write `fork` primitive to create checkpoints efficiently [29,54]. However, Linux's `fork` works poorly for checkpointing multithreaded processes because it creates a child process with only a single thread of control (the one that invoked `fork`). We therefore created a new Linux primitive, called a *multi-threaded fork*, that creates a child process with the same number of threads as its parent.

Not all thread states are safe to checkpoint. In particular, a thread cannot be checkpointed while executing an arbitrary kernel routine because of the likelihood of violating kernel invariants if the checkpoint is restored (this is possible because kernel memory is not part of the checkpoint). For example, the restarted thread would need to reacquire any kernel locks held prior to the checkpoint since the original checkpointed process would release those locks. It would also need to maintain data invariants; e.g., by not incrementing a reference count already incremented by the original process, etc.

Consequently, Respec only checkpoints a thread when it is executing at a known safe point:

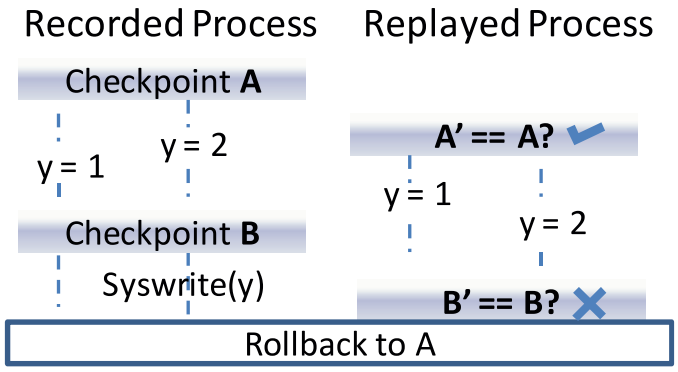

Figure 3.3: An execution with a data race that causes the replayed process to produce a memory state different from that of the recorded process. The divergence check fails and the two processes are rolled back to an earlier checkpoint.

kernel entry, kernel exit, or certain interruptible sleeps in the kernel that we have determined to be safe. The thread that initiates a multithreaded fork creates a barrier on which it waits until all other threads reach a safe point. Once all threads reach the barrier, the original thread creates the checkpoint, then lets the other threads continue execution. For each thread, the multithreaded fork primitive copies the registers pushed onto the kernel stack during kernel entry, as well as any thread-level storage pointers. The address space is duplicated using `fork`'s copy-on-write implementation.

Respec uses the multithreaded fork primitive in two circumstances: first, to create a replayed process identical to the one being recorded, and second, to create checkpoints of the recorded process that may later be restored on rollback. In the first case, the recorded process simply calls the multithreaded fork primitive directly. In the second case, the checkpointing code also saves additional information that is not copied by `fork` such as the state of the file descriptors and pending signals for the child process. The child process is not put on the scheduler's run queue unless the checkpoint is restored; thus, unless a rollback occurs, the child is merely a vessel for storing state. Respec deletes checkpoints once a following checkpoint has been verified to match for the recorded and replayed processes.

Respec checkpoints the recorded process at semi-regular intervals, called *epochs*. It takes an initial checkpoint when the replayed process is first created. Then, it waits for a predetermined amount of time (the epoch interval) to pass. After the epoch interval elapses, the next system call by any recorded thread triggers a checkpoint. After the remaining threads reach the multithreaded fork barrier and the checkpoint is created, all threads continue execution. The recorded process may execute several epochs ahead of the replayed process. It continues until either it is rolled back (due to a failed divergence check) or its execution ends.

Respec sets the epoch interval adaptively. There are two reasons to take a checkpoint. First, a new checkpoint bounds the amount of work that must be redone on rollback. Thus, the fre-

quency of rollback should influence the epoch interval. Respec initially sets the epoch interval to a maximum value of one second. If a rollback occurs, the interval is reduced to 50 ms. Each successful checkpoint commit increases the epoch interval by 50 ms until the interval reaches its maximum value. The second reason for taking a checkpoint is to externalize output buffered during the prior epoch (once the checkpoint is verified by comparing memory states of the recorded and replayed process). To provide acceptable latency for interactive tasks, Respec uses output-triggered commits [56] to receive a callback when output that depends on a checkpoint is buffered. Whenever output occurs during an epoch, we reduce that epoch's interval to 50 ms. If the epoch has already executed for longer than 50 ms, a checkpoint is initiated immediately. Note that the actual execution time of an epoch may be longer than the epoch interval due to our barrier implementation; a checkpoint cannot be taken until all threads reach the barrier.

3.3.2 Speculative execution

The recorded process is not allowed to externalize output (e.g., send a network packet, write to the console, etc.) until both the recorded and replayed processes complete the epoch during which the output was attempted and the states of the two processes match. A conservative approach that meets this goal would block the recorded process when it attempts an external output, end the current epoch, and wait for the replayed process to finish the epoch. Then, if the process states matched, the output could be released. This approach is correct, but can hurt performance by forcing the recorded and replayed process to execute in lockstep.

A better approach is available given operating system support for speculative execution. One can instead execute the recorded thread speculatively and either buffer the external output (if it is asynchronous) or allow speculative state to propagate beyond the recorded process as long as the OS guarantees that the speculative state can be rolled back and that speculative state will not causally effect any external output. We use Speculator [54] to do just that.

In particular, Speculator allows speculative state to propagate via fork, file system operations, pipes, Unix sockets, signals, and other forms of IPC. Thus, additional kernel data structures such as files, other processes, and signals may themselves become speculative without blocking the recorded process. External output is buffered within the kernel when possible and only released when the checkpoints on which the output depends are committed. External inputs such as network messages are saved as part of the checkpoint state so that they can be restored after a rollback. If propagation of speculative state or buffering of output is not possible (e.g., if the recorded thread makes an RPC to a remote server), the recorded thread ends the current epoch, blocks until the replayed thread catches up and compares states, begins a new epoch, and releases the output. We currently use this approach to force an epoch creation on

all network operations, which ensures that an external computer never sees speculative state. Respec allows multiple pairs of processes to be recorded and replayed independently, with the exception that two processes that write-share memory must be recorded and replayed together.

3.3.3 Logging and replay

Once a replayed process is created, it executes concurrently with its recorded process. Each recorded thread logs the system calls and user-level synchronization operations that it performs, while the corresponding replayed thread consumes the records to recreate the results and partial order of execution for the logged operations.

Conceptually, there is one logical log for each pair of recorded and replayed threads. Yet, for performance and security reasons, our implementation uses two physical logs: a log in kernel memory contains system call information and a user-level log contains user-level synchronization operations. If we logged both types of operations in the kernel's address space, then processes would need to enter kernel mode to record or replay user-level synchronization operations. This would introduce unacceptable overhead since most synchronization operations can be performed without system calls using a single atomic instruction. On the other hand, logging all operations in the application's address space would make it quite difficult to guarantee externally deterministic replay for a malicious application's execution. For instance, a malicious application could overwrite the results of a *write* system call in the log, which would compromise the replayed process's output check. By placing only the data necessary for performance at user-level, the verification of log records described in Section 3.3.7 is considerably simplified.

3.3.3.1 User-level logging

At user-level, we log the order of the most common low-level synchronization operations in glibc, such as locks, unlocks, futex waits, and futex wakes. The Posix thread implementation in glibc consists of higher-level synchronization primitives built on top of these lower-level operations. By logging only low-level operations, we reduce the number of modifications to glibc and limit the number of operation types that we log. Our implementation currently logs synchronization primitives in the Posix threads, memory allocation, and I/O components of glibc. An unlogged synchronization primitive in the rest of glibc, other libraries, or application code could cause the recorded and replayed processes to diverge. For such cases, we rely on rollback to re-synchronize the process states. As our results show, logging these most common low-level synchronization points is sufficient to make rollbacks rare in the applications we have tested.

Respec logs the entry and exit of each synchronization operation. Each log record contains

the type of operation, its result, and its partial order with respect to other logged operations. The partial order captures the total order of all the synchronization operations accessing the same synchronization variable and the program order of the synchronization operations executed in the same thread.

To record the partial order, we hash the address of the lock, futex, or other data structure being operated upon to one of a fixed number of global record clocks (currently 512). Each recorded operation atomically increments a clock and records the clock's value in a producer-consumer circular buffer shared between the recorded thread and its corresponding replayed thread. Thus, recording a log record requires at most two atomic operations (one to increment a clock and the other to coordinate access to the shared buffer). This allows us to achieve reasonable overhead even for synchronization operations that do not require a system call.

Using fewer clocks than the number of synchronization variables reduces the memory cost, and also produces a correct but stricter partial order than is necessary to faithfully replay a process. A stricter order is more likely to replay the correct order of racing operations and thereby reduce the number of rollbacks, as discussed in Section 3.2.2.

When a replayed thread reaches a logged synchronization operation, it reads the next log record from the buffer it shares with its recorded thread, blocking if necessary until the record is written. It hashes the logged address of the lock, futex, etc. to obtain a global replay clock and waits until the clock reaches the logged value before proceeding. It then increments the clock value by one and emulates the logged operation instead of replaying the synchronization function. It emulates the operation by modifying memory addresses with recorded result values as necessary and returning the value specified in the log. Each synchronization operation consumes two log records, one on entry and one on exit, which recreates the partial order of execution for synchronization operations.

Respec originally used only a single global clock to enforce a total order over all synchronization operations, but we found that this approach reduced replay performance by allowing insufficient parallelism among replayed threads. We found that the approach of hashing to a fixed number of clocks greatly increased replay performance (by up to a factor of 2–3), while having only a small memory footprint. Potentially, we could use a clock for each lock or futex, but our results to date have shown that increasing beyond 512 clocks offers only marginal benefits.

3.3.3.2 Kernel logging

Respec uses a similar strategy to log system calls in the kernel. On system call entry, a recorded thread logs the type of call and its arguments. For arguments that point to the application's address space, e.g., the buffer passed to `write`, Respec logs the values copied into the kernel during system call execution. On system call exit, Respec logs the call type, return value, and

any values copied into the application address space. When a replayed thread makes a system call, it checks that the call type matches the next record in the log. It also verifies that the arguments to the system call match. It then reads the corresponding call exit record from its log, copies any logged values into the address space of the replayed process and returns the logged return value.

Respec currently uses a single clock to ensure that the recorded and replayed process follow the same total order for system call entrance and exit. This is conservative but correct. Enforcing a partial order is possible, but requires us to reason about the causal interactions between pairs of system calls; e.g., a file *write* should not be reordered before a *read* of the same data.

Using the above mechanism, the replayed process does not usually perform the recorded system call; it merely reproduces the call's results. However, certain system calls that affect the address space of the application must be re-executed by the calling process. When Respec sees log records for system calls such as *clone* and *exit*, it performs these system calls to create or delete threads on behalf of the replayed process. Similarly, when it sees system calls that modify the application address space such as *mmap2* and *mprotect*, it executes these on behalf of the replayed process to keep its address space identical with that of the recorded process. This replay strategy does not recreate most kernel state associated with a replaying process (e.g., the file descriptor table), so a process cannot transition from replaying to live execution. To support such a transition, the kernel could deterministically re-execute native system calls [24] or virtualized system calls [58].

When the replayed process does not re-execute system calls, we do not need to worry about races that occur in the kernel code; the effect on the user-level address space of any data race that occurred in the recorded process will be recreated. For those system calls such as *mmap2* that are partially re-created, a kernel data race between system calls executed by different threads may lead to a divergence (e.g., different return values from the `mmap2` system call or a memory difference in the process address space). The divergence would trigger a rollback in the same manner as a user-level data race.

Because signal delivery is a source of non-determinism, Respec does not interrupt the application to deliver signals. Instead, signals are deferred until the next system call, so that they can be delivered at the same point of execution for the recorded and replayed threads. A data races between a signal handler and another thread is possible; such races are handled by Respec's rollback mechanism.

3.3.4 Detecting divergent replay

When Respec determines that the recorded and replayed process have diverged, it rolls back execution to the last checkpoint in which the recorded and replayed process states matched. A

rollback *must* be performed when the replayed process tries to perform an external output that differs from the output produced by the recorded process; e.g., if the arguments to a `write` system call differ. Until such a mismatch occurs, we need not perform a rollback. However, for performance reasons, Respec also eagerly rolls back the processes when it detects a mismatch in state that makes it *unlikely* that two processes will produce equivalent external output. In particular, Respec verifies that the replayed thread makes system calls and synchronization operations with the same arguments as the recorded thread. If either the call type or arguments do not match, the processes are rolled back. In addition, at the end of each epoch, Respec compares the address space and registers of the recorded and replayed processes. Respec rolls the processes back if they differ in any value.

Checking memory values at each epoch has an additional benefit: it allows Respec to release external output for the prior epoch. By checking that the state of the recorded and the replayed process are identical, Respec ensures that it is possible for them to produce identical output in the future. Thus, Respec can commit any prior checkpoints, retaining only the one for which it just compared process state. All external output buffered prior to the retained checkpoint is released at this time. In contrast, if Respec did not compare process state before discarding prior checkpoints, it would be possible for the recorded and replayed process to have diverged in such a way that they could no longer produce the same external output. For example, they might contain different strings in an I/O buffer. The next system call, which outputs that buffer, would always externalize different strings for the two processes.

Respec leverages kernel copy-on-write mechanisms to reduce the amount of work needed to compare memory states. Since the checkpoint is an (as-yet-unexecuted) copy of the recorded process, any modifications made to pages captured by the checkpoint induce a copy-on-write page fault, during which Respec records the address of the faulted page. Similarly, if a page fault is made to a newly mapped page not captured by the checkpoint, Respec also records the faulting page. At the end of each epoch, Respec has a list of all pages modified by the recorded process. It uses an identical method to capture the pages modified by a replayed process; instead of creating a full checkpoint, however, it simply makes a copy of its address space structures to induce copy-on-write faults. Additionally, Respec parallelizes the memory comparison to reduce its latency.

In comparing address spaces, Respec must exclude the memory modified by the replay mechanism itself. It does this by placing all replay data structures in a special region of memory that is ignored during comparisons. In addition, it allocates execution stacks for user-level replay code within this region. Before entering a record/replay routine, Respec switches stacks so that stack modifications are within the ignored region. Finally, the shared user-level log, which resides in a memory region shared between the recorded and replayed process, is also ignored during comparisons.

3.3.5 Rollback

Rollback is triggered when memory states differ at the end of an epoch or when a mismatch in the order or arguments of system calls or synchronization operations occurs. Such mismatches are always detected by the replayed process, since it executes behind the recorded process. Respec uses Speculator to roll back the recorded process to the last checkpoint at which program states matched. Speculator switches the process, thread, and other identifiers of the process being rolled back with that of the checkpoint, allowing the checkpoint to assume the identity of the process being rolled back. It then induces the threads of the recorded process to exit. After the rollback completes, the replayed process also exits.

Immediately after a checkpoint is restored, the recorded thread creates a new replayed process. It also creates a new checkpoint using Speculator (since the old one was consumed during the rollback). Both the recorded and replayed threads then resume execution.

Given an application that contains many data races, one can imagine a scenario in which it is extremely unlikely for two executions to produce the same output. In such a scenario, Respec might enter a pathological state in which the recorded and replayed processes are continuously rolled back to the same checkpoint. We avoid this behavior by implementing a mechanism that guarantees forward progress even in the presence of unbounded data races. This mechanism is triggered when we roll back to the same checkpoint twice.

During retry, one could use a logger that instruments all memory accesses and records a precise memory order. Instead we implemented a simpler scheme. We observe that the recorded and replayed process will produce identical results for even a racy application as long as a single thread is executed at a time and thread preemptions occur at the same points in thread execution. Therefore, Respec picks only one recorded thread to execute; this thread runs until either it performs an operation that would block (e.g., a `futex` wait system call) or it executes for the epoch interval. Then, Respec takes a new checkpoint (the other recorded threads are guaranteed to be in a safe place in their execution since they have not executed since the restoration of the prior checkpoint). After the checkpoint is taken, all recorded threads continue execution. If Respec later rolls back to this new checkpoint, it selects a new thread to execute, and so on. Respec could also set a timer to interrupt user-level processes stuck in a spin loop and use a branch or instruction counter to interrupt the replayed process at an identical point in its execution; such mechanisms are commonly used in uniprocessor replay systems [24]. Thus, Respec can guarantee forward progress, but in the worst case, it can perform no better than a uniprocessor replay system. Fortunately, we have not yet seen a pathological application that triggers this mechanism frequently.

3.3.6 Offline replay support

When requested, Respec can optionally save information to enable an offline replay of the recorded process. This information includes the kernel's log of system calls, the user-level log of synchronization operations, and an MD5 checksum of address space and register state at the end of each committed rollback. Since not all races are logged, offline replay of the recorded process is not guaranteed to succeed in the first attempt. However, since the recorded process has been replayed successfully at least once, it is likely that offline replay will eventually succeed, although it may require a number of rollbacks and retries. Combining Respec output with an offline replay search tool, such as is done in ODR [5] and PRES [60], would be a promising approach to reduce search time.

3.3.7 Security considerations

One use of deterministic replay is to parallelize security checks by running them on one or more replayed processes [55]. This section describes the security issues that must be considered when using replay in this type of adversarial context, in which the software being replayed may actively try to disrupt the replay system. For example, an attacker who compromises the replay system could try to force the replayed process to skip over the execution interval that a security check would have detected as suspicious.

Recall that the goals of our externally deterministic replay system are to ensure that: (1) the replayed execution matches the output of the original execution (including the state at the end of a replay epoch), and (2) the replayed execution is a natural execution of the target program. By a "natural execution", we mean that the replayed execution must match the instruction-for-instruction execution of one of the possible executions of the original system (i.e., the system without the kernel and library support for replay). While the replayed execution may diverge from the original execution within an epoch, it must still converge back with the state of the original execution by the end of the epoch.

Meeting these goals ensures that if the replayed process passes all security checks, a natural run exists that passes all security checks, so the output and state produced by the original and replayed processes is safe with respect to those checks. The most an attacker can do is to choose *which* natural run the replayed process executes, but that natural run must still match the output and state of the original process. For example, an attacker could try to avoid detection by a buffer overflow security check by (1) overflowing a buffer in the original process and (2) causing the replayed process to execute a different natural path that did not overflow the buffer (and hence did not trigger the security check). However, the same output and state could have been produced by the program *without* the buffer overflow, since that is exactly what the replayed process has done.

We next argue that Respec meets these goals, even when replaying malicious software. We assume that the software being recorded and replayed cannot corrupt kernel data. Therefore, we place as much of Respec as possible within the kernel. Speculator, the kernel log, and memory checkpoints are all placed in the kernel.

The only replay data that the malicious software can corrupt is the user-level log shared between the recorded and replayed processes; this log contains the order of user-level synchronization operations. Respec must therefore treat the user-level log as suspect. If, for example, Respec trusted the header length fields in each user-level log record, the malicious software might be able to cause a buffer overflow in the replay system and cause the replayed process to deviate from the set of natural runs. More subtly, the malicious software could record an order of synchronization operations that could not be generated by a natural run. For example, the malicious software could write a log record that caused a lock operation to succeed even when the lock was already held by another thread.

To protect against these attacks, Respec has an optional verification mode that can be used during replay. When verification mode is enabled, the replayed process copies each record from the shared user-level log to a non-shared memory region, then verifies that the log record represents a *possible* execution path. For instance, it rejects a log record that shows a lock operation completing successfully when the lock is already held by another thread. To verify records, the replayed process shadows the state of locks, futexes, and other logged data structures (the original memory reserved for these structures is used for this purpose since the replayed process does not execute the actual synchronization operations). If a logged action would be invalid given the shadowed state, the action is rejected and a mismatch reported. Since Respec only logs a few types of low-level operations, such verification is not difficult. However, since verification adds overhead, it can be disabled to improve performance when replay is not being used for security checks.

3.4 Results

Our evaluation answers the following questions:

- What is the overhead of Respec record and replay for common applications and benchmarks?
- How often does imprecise logging lead to rollbacks for those application and benchmarks?
- What is the cost of rollback and retry when it occurs?

application	threads	synch. ops.	system calls	epochs	pages compared	original time (s) & stdev.	redundant time (s) & stdev.	Respec time (s) & stdev.	slowdown wrt redundant	slowdown wrt original
blackscholes	1	524330	30	4	2973	7.04 (0.00)	7.04 (0.00)	7.34 (0.02)	4%	4%
	2	524345	36	4	2974	3.54 (0.00)	3.54 (0.00)	3.79 (0.04)	7%	7%
	3	524361	41	4	2976	2.37 (0.00)	2.37 (0.00)	2.77 (0.05)	17%	17%
	4	524376	47	4	2977	1.79 (0.00)	1.94 (0.08)	2.24 (0.08)	15%	25%
bodytrack	1	387794	6180	11	4091	8.60 (0.01)	8.58 (0.03)	8.74 (0.02)	2%	2%
	2	389907	7539	11	4235	4.89 (0.01)	4.83 (0.08)	5.13 (0.06)	6%	5%
	3	393314	9982	7	3469	3.55 (0.02)	3.53 (0.05)	3.82 (0.13)	8%	8%
	4	395835	11060	10	5733	2.86 (0.02)	3.08 (0.09)	4.82 (0.21)	56%	68%
fluidanimate	1	541964	2749	5	26475	6.69 (0.00)	6.72 (0.04)	6.74 (0.01)	0%	1%
	2	4773663	3017	5	28309	4.04 (0.00)	4.13 (0.02)	4.60 (0.03)	11%	14%
	4	7545571	3136	5	28895	2.52 (0.01)	2.61 (0.01)	4.35 (0.19)	67%	73%
swaptions	1	1922355	102	7	413	6.77 (0.00)	6.78 (0.00)	7.87 (0.04)	16%	16%
	2	1905088	214	6	447	3.39 (0.00)	3.40 (0.01)	3.89 (0.06)	15%	15%
	3	1905178	286	4	464	2.34 (0.00)	2.34 (0.01)	2.56 (0.03)	10%	10%
	4	1800897	818	4	471	1.76 (0.18)	1.78 (0.02)	2.06 (0.16)	16%	17%
streamcluster	1	59681	7239	14	3121	10.97 (0.08)	11.09 (1.41)	11.01 (0.66)	-1%	0%
	2	108360	31159	8	2769	5.62 (0.66)	5.66 (0.47)	5.62 (0.56)	-1%	0%
	3	157874	56674	8	2798	3.35 (0.53)	4.64 (0.54)	5.68 (0.56)	22%	69%
	4	208367	83029	8	2834	2.51 (0.32)	4.63 (0.37)	5.88 (0.52)	27%	134%
ocean	1	2648	61	5	202790	4.93 (0.00)	5.00 (0.00)	7.06 (0.05)	41%	43%
	2	5203	855	4	154503	2.50 (0.00)	3.03 (0.01)	4.29 (0.03)	42%	72%
	4	10366	2642	3	106102	1.49 (0.03)	2.28 (0.02)	3.25 (0.08)	43%	119%
raytrace	1	1115639	1143	2	8352	0.79 (0.01)	0.80 (0.01)	1.34 (0.01)	68%	70%
	2	1123858	5632	2	8352	0.64 (0.01)	0.65 (0.01)	1.29 (0.03)	99%	101%
	3	1117220	7046	2	8362	0.59 (0.00)	0.59 (0.00)	1.47 (0.12)	148%	150%
	4	1118038	6102	2	8352	0.57 (0.00)	0.57 (0.00)	1.55 (0.10)	171%	173%
volrend	1	632	448	2	7372	1.83 (0.01)	1.83 (0.01)	1.88 (0.01)	3%	3%
	2	141678	619	2	7377	1.33 (0.01)	1.34 (0.00)	1.39 (0.01)	3%	4%
	3	143319	2784	2	7381	1.19 (0.00)	1.19 (0.00)	1.27 (0.00)	7%	7%
	4	142177	4084	2	7385	1.10 (0.00)	1.13 (0.06)	1.21 (0.05)	7%	9%
water-nsq	1	67055	535	3	4760	3.05 (0.11)	3.09 (0.10)	3.16 (0.13)	2%	4%
	2	122848	908	2	4203	1.68 (0.03)	1.68 (0.03)	1.74 (0.02)	4%	4%
	3	156131	1281	2	6254	1.14 (0.01)	1.23 (0.03)	1.34 (0.03)	9%	18%
	4	181343	1840	2	8305	0.90 (0.02)	0.91 (0.01)	1.38 (0.01)	52%	54%
fft	1	110	22	1	0	0.82 (0.00)	0.84 (0.09)	0.84 (0.00)	1%	2%
	2	166	41	1	0	0.53 (0.00)	0.56 (0.04)	0.57 (0.00)	1%	8%
	4	275	77	1	0	0.40 (0.01)	0.44 (0.01)	0.45 (0.00)	3%	12%
radix	1	109	18	2	16408	4.50 (0.01)	4.51 (0.65)	4.61 (0.02)	2%	3%
	2	193	35	2	16416	2.29 (0.01)	2.35 (0.01)	2.41 (0.03)	3%	5%
	4	392	81	2	31206	1.16 (0.00)	1.28 (0.00)	1.44 (0.04)	12%	24%
pfscan	1	267	75	3	19	1.94 (0.01)	1.94 (0.01)	1.99 (0.05)	3%	2%
	2	301	83	2	15	1.15 (0.01)	1.16 (0.03)	1.19 (0.03)	3%	4%
	3	340	91	2	16	0.92 (0.00)	0.94 (0.01)	0.97 (0.02)	3%	5%
	4	376	99	2	16	0.77 (0.00)	0.83 (0.02)	0.98 (0.04)	19%	28%
pbzip2	1	993	297	20	33654	4.59 (0.00)	4.81 (0.16)	4.83 (0.10)	0%	5%
	2	958	359	11	33699	2.35 (0.00)	2.42 (0.09)	2.73 (0.13)	13%	16%
	3	954	386	8	33794	1.64 (0.05)	1.70 (0.03)	2.03 (0.15)	19%	24%
	4	1005	409	6	33050	1.33 (0.00)	1.44 (0.04)	1.93 (0.23)	34%	45%
aget	1	8618	14681	4147	29039	2.05 (0.16)	N/A	2.19 (0.14)	N/A	7%
	2	8739	13905	3921	27985	1.93 (0.00)	N/A	2.17 (0.08)	N/A	13%
	3	8770	13096	3689	26348	1.94 (0.00)	N/A	2.08 (0.04)	N/A	7%
	4	8432	12944	3642	26269	1.96 (0.04)	N/A	2.08 (0.06)	N/A	6%
apache	1	3808	11114	18065	5654	8.08 (0.06)	N/A	8.13 (0.02)	N/A	1%
	2	3557	10919	17898	5851	7.89 (0.03)	N/A	8.56 (0.12)	N/A	9%
	3	3417	10902	17922	5899	7.40 (0.04)	N/A	9.42 (0.16)	N/A	27%
	4	3571	10945	17937	5824	6.98 (0.04)	N/A	10.04 (0.12)	N/A	44%

Table 3.1: Respec performance. Results are the mean of ten trials with the exception of pbzip2 and aget, which show the mean of 100 and 50 trials respectively. Values in parentheses show standard deviations.

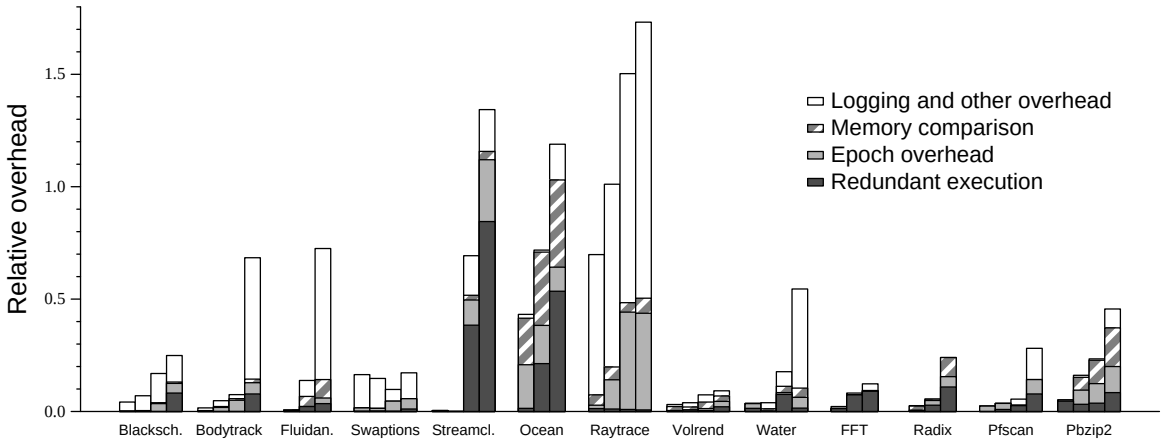

Figure 3.4: Breakdown of overhead per benchmark. For most benchmarks, results are presented for 1, 2, 3, and 4 threads (left to right). For Fluidanimate, Ocean, FFT, and Radix, results are presented for 1, 2, and 4 threads.

3.4.1 Methodology

We ran all experiments on a 2 GHz 8-core Xeon processor with 3 GB of RAM running CentOS Linux version 5.3. The Linux kernel is a stock Linux 2.6.27 kernel, which we modified to support Respec deterministic replay. In addition, the kernel has the Speculator support for speculative execution. We also modified the GNU glibc library version 2.5.1 to support Respec.

We used three sets of benchmarks. The first set is five benchmarks from the PARSEC suite [14]: blackscholes, bodytrack, fluidanimate, swaptions, and streamcluster. The second set is six benchmarks from the SPLASH-2 suite [79]: ocean, raytrace, volrend, water-nsq, fft, and radix. The final set is four parallel applications: pbzip2, which we use to compress a 17 MB log file in parallel; pfscan, which we use to search in parallel for a string in a directory with 952 MB of log files; aget, which we use to retrieve a 21 MB file over a local network; and Apache, which we test using ab (Apache Bench) to simultaneously send 100 requests each from four concurrent clients over a local network.

For all benchmarks, we ensure that all files are in the file cache in kernel memory before execution begins. Thus, our experimental results do not include any disk I/O time, which would mask the relative overhead of deterministic replay. We report three values for each experiment: the original execution time of the application running on a stock system, a "redundant" execution time in which two copies of the application are run concurrently on a stock system by forking an execution at the start, and the execution time using Respec to provide deterministic replay. The redundant execution is a lower bound on execution time for online replay; of course, the execution outputs could diverge. The Respec execution time measures the time for both the recorded and replayed processes to finish.

For each benchmark, we vary the number of worker threads from one to four. Many bench-

threads	rollback freq	original time (s) & stdev.	type	Respec time (s) & stdev.	slowdown wrt original
1	0%	4.59 (0.00)	overall	4.83 (0.10)	5%
2	13% once	2.35 (0.00)	w/o rollback	2.70 (0.09)	15%
			w/ rollback	2.97 (0.12)	26%
			overall	2.73 (0.13)	16%
3	9% once 2% twice	1.64 (0.05)	w/o rollback	2.00 (0.10)	22%
			w/ rollback	2.29 (0.17)	40%
			overall	2.03 (0.15)	24%
4	15% once 1% twice	1.33 (0.00)	w/o rollback	1.88 (0.16)	41%
			w/ rollback	2.24 (0.29)	68%
			overall	1.93 (0.23)	45%

Table 3.2: Rollback frequency in pbzip2

marks have additional control threads which do little work during the execution; we do not count these in the number of threads. Pbzip2 uses two additional threads: one to read file data and one to write the output; these threads are also not counted in the number of threads shown. Unless otherwise mentioned, all results are the mean of ten trials.

3.4.2 Record and replay performance

Table 3.1 shows the overall performance results for Respec. The first two columns show the application or benchmark executed and the number of worker threads used. The next four columns give statistics about Respec execution: the number of user-level synchronization operations logged, system calls logged, epochs executed, and memory pages compared. The next three columns show the original, redundant, and Respec execution times. The last two columns show Respec's overhead with respect to the lower bound of the redundant execution time and with respect to the original execution time. For the two networked applications (aget and apache), measuring redundant execution time is difficult because the two separate processes contend for network resources, whereas with Respec only the recorded process sends and receives network packets.

Figure 4.5 provides a breakdown of the overhead normalized to the original execution time for all non-networked benchmarks. Each data set shows results for 1, 2, 3 (if feasible), and 4 worker threads. The dark shaded area in each bar show relative overheads for redundant execution. The lighter shaded areas show relative overhead associated with use of multiple epochs, excluding memory comparison cost. We believe that this overhead is mainly due to page fault overhead. The diagonally hashed areas show relative overhead due to memory comparison. The remaining region shows all other overhead, the majority of which is likely due to logging

threads	rollback freq	original time (s) & stdev.	type	Respec time (s) & stdev.	slowdown wrt original
1	10% once 2% twice	2.05 (0.16)	w/o rollback	2.19 (0.14)	7%
			w/ rollback	2.21 (0.13)	8%
			overall	2.19 (0.14)	7%
2	20% once 2% twice	1.93 (0.00)	w/o rollback	2.17 (0.08)	13%
			w/ rollback	2.17 (0.05)	13%
			overall	2.17 (0.08)	13%
3	24% once	1.94 (0.00)	w/o rollback	2.08 (0.05)	7%
			w/ rollback	2.09 (0.02)	8%
			overall	2.08 (0.04)	7%
4	18% once 2% twice	1.96 (0.04)	w/o rollback	2.07 (0.05)	6%
			w/ rollback	2.08 (0.02)	6%
			overall	2.08 (0.06)	6%

Table 3.3: Rollback frequency in aget

synchronization operation and system calls. Respec's implementation makes it difficult for us to provide similar breakdowns for networked applications (Apache and aget) because use of multiple epochs is required to correctly interact with clients on different computers, due to the output commit problem (as discussed in Section 3.3.2).

Examining the results, we see that Respec overhead is generally quite low. For 2 worker threads, Respec has average overhead with respect to the original execution of only 18% across all benchmarks. This overhead gradually increases with the number of threads; with 4 threads, Respec's average overhead is 55%. Compared to the lower bound of redundant execution, Respec's average overhead is 16% with 2 threads and 40% with 4 threads. We conjecture that this increase derives mostly from the increased synchronization between replay threads.

It is informative to examine some of the benchmarks with extreme characteristics. Fluidanimate from the PARSEC suite and raytrace from the SPLASH-2 suite execute over two million logged synchronizations per second with four threads. Most of these operations are uncontended lock and unlock operations, which do not require a system call. In these cases, Respec overhead derives from the cost of logging these user-level operations. Ocean and streamcluster show larger overheads with respect to the original execution for 4 threads, but show significantly less overhead for 2 threads. In fact, for 4 threads, simply executing two copies of these benchmarks concurrently shows similar large increases in execution time, indicating that most of the overhead derives simply from sharing the limited memory bandwidth and processor caches for redundant execution rather than from Respec itself. Many benchmarks show a spike in overhead when the number of worker threads increases from 3 to 4; part of the reason for this spike may be that our 8 core machine has no CPU capacity to spare for auxiliary threads if 4 cores each

are used to record and replay worker threads.

Respec overhead for the four applications at the bottom of Table 3.1 is relatively low, averaging 11% with 2 worker threads and 31% with 4 worker threads. This lower overhead is to be expected since these applications issue fewer system calls and synchronization operations than the PARSEC and SPLASH-2 benchmarks.

3.4.3 Rollback frequency

Rollbacks were infrequent events for our benchmarks. In fact, only pbzip2 and aget were rolled back during our experiments. Pbzip2 has one benign application-level data race in which an output thread repeatedly spins on two variables (once for each chunk of data being compressed) waiting for a worker thread to modify them from zero to one. While the race is benign, it affects the number of system calls issued in some executions. It would also be difficult to identify and log this race other than through manual code inspection since the spin loop does not use atomic instructions. Additionally, pbzip2 uses the stdlibc++ library, which we have not modified to log synchronization operations.

To better understand the frequency of rollbacks, we ran pbzip2 100 times. Table 3.2 shows that 13–16% of the executions with more than one worker thread contained one rollback. In general, the cost of rollback was reasonable. Rollbacks contribute 8% of Respec's total overhead when pbzip2 uses multiple worker threads.

For aget, one thread reads and displays download progress without obtaining a lock. This data race is benign since display of slightly stale status information is acceptable. Table 3.3 shows that the divergent output and memory state leads to rollbacks. However, the performance impact of these rollbacks is negligible because Respec checkpoints aget very frequently (on every network receive).

3.4.4 The cost of rollback

To better understand the cost of rollbacks, we *artificially* inserted rollbacks into some of our benchmarks by emulating the failure of a divergence check during benchmark execution. We disabled Respec's adaptive epoch algorithm and manually set the epoch interval to a configured value.

Figure 3.5 shows the additional time needed to complete four benchmarks when a single rollback is artificially introduced for different epoch intervals. The results show that rollback overhead is roughly proportional to the length of the epoch interval. Intuitively, the execution time increases by the amount of work that must be redone after a rollback. Any fixed cost introduced by the rollback mechanism itself appears to be minimal.

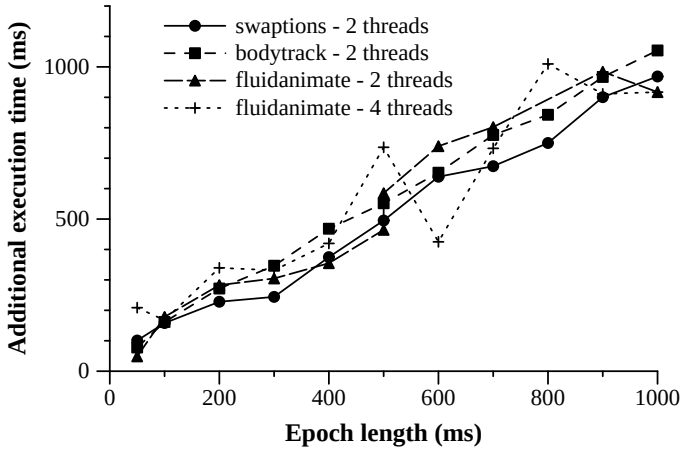

Figure 3.5: Impact of epoch interval on rollback overhead

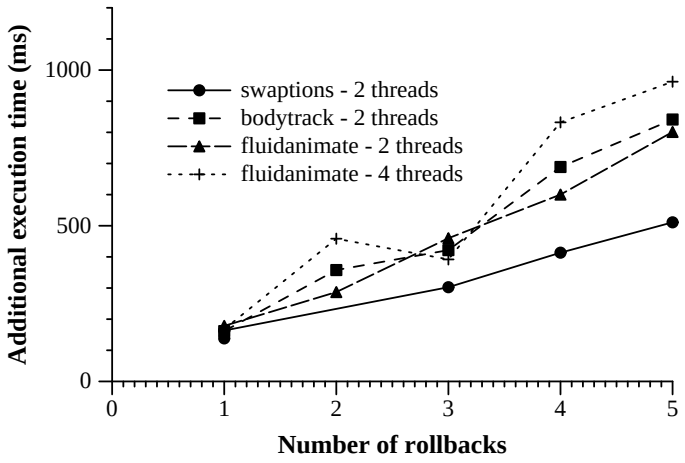

Figure 3.6: Impact of number of rollbacks on rollback overhead

For a final experiment, we varied the number of rollbacks during benchmark execution while keeping the epoch interval fixed at 100 ms. Figure 3.6 shows that the cost of rollbacks is proportional to the number of rollbacks. Interestingly, fluidanimate and bodytrack show a cost of approximately 160–180 ms per rollback (60–80 ms greater than the epoch interval length). Upon further investigation, we found the reason to be our barrier implementation for checkpointing; it can take several tens of milliseconds for all threads to reach the barrier for these two benchmarks.

3.5 Extensions

Respec, proposed in this Chapter, is originally designed to support online replay in which recording and replaying are performed concurrently (e.g., for fault toleranace, offloaded program analysis, etc.). However, Respec is not suitable to support offline replay as Respec needs to rollback recorded and replayed processes in cases when external determinism is violated, and for recovery Respec has to constrain original execution as well (e.g., running all threads on a uniprocessor to guarantee forward progress). For offlien replay in which original execution has been already completed, Respec may perform offline search to find a thread interleaving, which Respec doesn't record by default, that leads to the same output and final state. Though such offline search is feasible, this approach does not guarantee reproducibility due to hugh search space.

To address this problem, I also helped extend Respec to build DoublePlay [75], which guarantees offline multiprocessor replay (e.g., for debugging) using a novel execution model called uniparallelism. Uniparallelism slices multithreaded execution into multiple time intervals (epochs), and replays them concurrently on separate processors using Respec's online replay support. By running each epoch on a single processor, DoublePlay does not need to log the order of shared-memory accesses. Instead, similar to uniprocessor replay, the system only logs the order in which threads in an epoch are timesliced on the processor. This strategy allows us to record multiprocessor execution for a low latency cost (28%) and at the expense of 2x cores.

In addition to efficient deterministic replay, uniparallel execution, that provide the benefits of uniprocessor execution while scaling performance with increasing cores in multiprocessors, has shown to be useful in developing many useful tools. Veeraraghavan et al. [74] proposes Frost that support detecting and surviving data races using complementary schedules on top of uniparallel execution. Wester et al. [77] uses uniparallelism to provide parallel data race detection that spread checking across multiple cores. Ouyang et al. [59] shows that uniparallel execution can be used to provide region serilizability, more intuitive program semantics for parallel programs.

3.6 Conclusion

Respec provides an operating system based solution for fast, online shared memory multiprocessor replay. Respec uses two novel techniques to achieve efficiency: external determinism, a new fidelity level for replay, and speculative execution. External determinism provides adequate guarantees for most applications of replay, but its relaxed constraints yield sufficient freedom to support efficient multiprocessor replay. Respec uses speculative execution to optimistically log only the most common synchronization operations, relying on rollback and retry to guarantee correctness in the rare cases where the recorded and replayed processes diverged due to unlogged races. These two techniques allow Respec to concurrently record and replay multithreaded programs with an average overhead of 18% for two threads and 55% for four threads.

CHAPTER 4

Chimera: Hybrid Program Analysis for Multiprocessor Replay

In the previous Chapter 3, we describe Respec that supports efficient multiprocessor replay based on external determinism and speculative execution. However, Respec requires a redundant execution on spare cores to enable low-overhead replay, and therefore it incurs about 2x throughput cost. This Chapter 4 proposes another efficient software-based solution, named Chimera, based on static program analysis that does not require redundant execution.

Again, the fundamental challenge in multiprocessor replay comes from recording the non-deterministic interleaving among threads, a property that is necessary to deterministically replay programs that contain data-races. Recording and replaying the frequent interactions among thread accesses to shared data can slow execution by an order of magnitude or more. However, if one could somehow statically guarantee that a program is data-race-free, then it is not necessary to record thread interactions — past research has shown that recording and replaying the happens-before order of synchronization operations is sufficient to ensure deterministic replay [63]. In most applications, synchronization operations are relatively infrequent compared to memory accesses, and therefore logging them is relatively cheap. Many uses of deterministic replay, including program debugging, reproducing errors encountered in the field in a test environment, replication for fault tolerance, and forensics following a computer intrusion, are especially useful for programs with bugs, so working only for bug-free programs (i.e., programs without data-races) is not a viable option.

Unfortunately, statically proving the absence of data-races in a program without rejecting data-race-free programs is hard. If there is a chance that a program contains a data-race, then one must record the order of potentially racing operations in order to guarantee that the recorded program can be replayed deterministically. One could discover such operations with a dynamic data-race detector. However, despite significant advances, dynamic data-race detection in software slows program execution by nearly 8x [28] for state-of-the-art detectors. Thus, logging the order of potentially racing instructions is no less of a problem than detecting a data race.

In this chapter we discuss Chimera, a deterministic replay system that employs a new hy-

brid program analysis to handle programs with data races. Chimera combines static data race analysis with off-line profiling and targeted, dynamic checks to provide deterministic replay efficiently.

Chimera instruments a program to log all non-deterministic inputs (e.g., system call results), the thread schedule on each processor core, and the happens-before relationships due to synchronization operations. This information is sufficient to guarantee that the program can later be replayed deterministically, provided the program contains no data-races.

To provide replay for racy programs, Chimera uses a sound but imprecise static data-race detector (RELAY [76]) to find potential data-races. Every memory instruction that potentially races with another instruction is placed inside a code region protected by a *weak-lock*. Chimera records all happens-before relationships due to weak-locks in addition to the relationships due to the original program synchronization. Thus, Chimera guarantees deterministic replay for all programs.

We use weak-locks instead of traditional locks in order to be conservative and avoid introducing artificial deadlocks. A weak-lock is essentially a time-out lock, where mutual exclusion is compromised if the weak-lock is not acquired in reasonable amount of time. In the rare case when a weak-lock times out, Chimera deterministically preempts the thread that currently holds the weak-lock and forces it to yield the weak-lock to the thread that timed out on the weak-lock; the original holder of the weak-lock must reacquire the weak-lock before resuming its execution. This approach splits the code region protected by the weak-lock into two regions across the preemption. Because this timeout mechanism enables Chimera to preserve the invariant that only one thread holds a given weak-lock at any given time, Chimera can support deterministic replay by reproducing the order of weak-locks at the same preemption point.

Unsurprisingly, we find that a sound data-race detector reports a large number of false data-races, and thus adding a weak-lock for every reported data-race results in prohibitively high overhead. Chimera employs two critical optimizations to drastically reduce this cost.

Both optimizations attempt to increase the granularity of a weak-lock, in terms of the size of the locked code region and the amount of data the lock protects. Coarser weak-locks reduce the cost of instrumentation but may serialize threads unnecessarily and compromise parallelism. Chimera's optimizations navigate this performance trade-off by targeting the main sources of imprecision in a static data-race detector [76].

The first optimization is based on the observation that a large fraction of false data-race reports are due to the inability of the static data-race detector to account for the happens-before relations due to synchronization operations other than locks. One example of this is that a number of data-races are reported between initialization code and the rest of the program because the static tool does not account for the happens-before relation due to fork-join synchronization. To address this problem, we profile the program offline over a variety of inputs. If the code re-

gions containing potentially racing instructions are non-concurrent in all profile runs, Chimera increases the granularity of the weak-lock to protect the entire code region instead of just one instruction. Chimera currently treats functions as code regions. This optimization reduces the number of times weak-locks are acquired and released during a function's execution.

The second optimization pertains to the remaining set of false racy pairs that are part of function pairs that ran concurrently in at least one profile run. This optimization targets the inaccuracy caused by the conservative points-to analysis [70, 6] on which RELAY is based. Due to this analysis, RELAY overestimates the set of shared objects that could be accessed by a memory instruction and also underestimates the set of locks that could be acquired. We observe that while the numeric values for the address bounds of an object accessed by a memory instruction are generally hard to determine precisely during static analysis, one can often estimate reasonable bounds in the form of a symbolic expression [65].

Therefore, in our implementation, we compute symbolic address bounds of objects that can be accessed by a racing instruction within a loop. Using this information, we increase the granularity of the weak-lock to the entire loop containing the race, such that it protects the loop for the data variables specified by the loop's symbolic address bounds. This avoids the cost of instrumentation for every iteration of the loop.

Our evaluation shows that Chimera is more efficient than the state-of-the-art software solutions that guarantee multiprocessor replay [75]. We show that recording a set of server (e.g., Apache) and desktop (e.g., pbzip, aget) applications incurs only about 2.4% performance overhead, and recording a set of memory-intensive scientific applications (SPLASH [79] incurs about 86%. Replay overhead is also similar to that of recording. We find that our two optimizations play a significant role in bringing the average overhead from 53x (when all races are naively instrumented) to 1.39x.

Programs transformed by Chimera are data-race-free under the new set of synchronization operations. Though our immediate motive for this transformation is to provide deterministic record and replay, we envision that future work may be able to leverage the data-race-freedom provided by Chimera to provide stronger guarantees such as sequential consistency and deterministic execution [57], since these properties are much easier to guarantee in the absence of data-races.

4.1 Design

This section provides a design overview of the Chimera multiprocessor replay system.

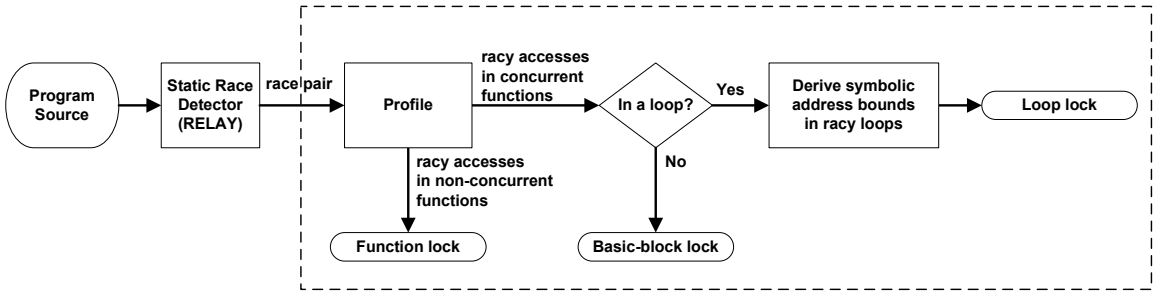

Figure 4.1: Chimera Overview

4.1.1 Background

A program is said to be data-race-free if none of its executions exhibit a data-race. Two memory instructions are said to be *racy* if at least one of them is a write, and there is at least one execution where the two are executed in different threads and not ordered by any happens-before relation due to synchronization operations. For clarity, we define a few terms that we use in this paper. A race-pair is a pair of static memory instructions that are racy. The two functions (or loops) that contains the race-pair are referred to as a racy-function-pair (or a racy-loop-pair).

Chimera records non-deterministic input (e.g., interrupts and file reads) and happens-before relations due to synchronization accesses in a program. This is sufficient to later provide deterministic replay for data-race-free programs because all memory instructions are ordered by some happens-before relation [63]. However, it is insufficient to later provide deterministic replay for programs that contain data-races.

4.1.2 Design Overview

Figure 4.1 presents an overview of how Chimera transforms a potentially racy program to a data-race-free program by adding additional synchronization and runtime constraints. Chimera's transformation does not attempt to correct a given program, it simply makes it easier to deterministically record and replay the program's execution.

Chimera analyzes a given program using the RELAY [76] static data-race detector. RELAY is sound, except for two corner cases (assembly instructions and pointer arithmetic). However, the unsoundness is modularized and can be addressed using additional analysis [9, 78] (Section 4.2.2).

In the simplest implementation of Chimera, each race-pair is placed inside a code section protected by an unique weak-lock w. Recording and replaying the happens-before relation due to weak-locks enables Chimera to record and replay the order of all racy accesses and thus guarantee deterministic replay for all programs.

A sound static data-race detector is imprecise as it has to make very conservative assump-

tions. This results in a huge number of false data-races, and naively recording all those races results in high overhead. The insight of this paper is that by employing a combination of profiling, symbolic address bounds analysis and dynamic checks, the overhead is significantly reduced to the point where deterministic record and replay is viable even for production systems.

We discuss two specific optimizations. The first optimization is based on our observation that for many false race-pairs, the code regions containing them are almost never executed concurrently. One main cause for this imprecision is the static data-race detector's inability to account for non-mutex synchronization operations. Chimera learns which code regions are almost always non-concurrent by profiling executions with a set of representative inputs. It uses profile information to increase the granularity of weak-locks, both in terms of size of the code region and the amount of shared objects they protect, which reduces the number of weak-lock operations at runtime. Chimera's profiler treats every function as a code region, though other granularities could be considered. As shown in Figure 4.1, racy-pairs in non-concurrent functions are handled using weak-locks instrumented at the granularity of a function (referred to as *function-locks*).

Not all false data-races are part of non-concurrent code regions. Two code regions can overlap in time, but still may not exhibit a data race if they access different sets of shared objects. However, a static data-race detector may not always be able to prove that the set of shared objects accessed in concurrently executed code regions are disjoint due to imprecise pointer analysis. While it is hard to accurately compute the numeric values for address bounds statically for a code region, it is often possible to derive a symbolic expression for the upper and lower bounds of an object that will be accessed within a code region [65].

For data-races that are not found to be part of non-concurrent functions, Chimera checks if they are part of a loop. If a data-race is not part of any loop, then Chimera simply instruments a weak-lock at the granularity of a basic block (referred to as a *basic block lock*). In case the basic block has a function call, Chimera instruments a weak-lock at the granularity of an instruction (referred to as an *instruction lock*).

If a data-race is part of a loop, Chimera derives a symbolic address bound for the range of addresses that a racy instruction can access within the loop. A race-pair is then guarded by instrumenting a loop-lock. The loop-lock is also a weak-lock, but it protects a range of addresses, which are computed at runtime using the symbolic expression derived statically. If the symbolic bounds expression is too imprecise (e.g., one of the bounds is infinity), and if the loop body is reasonably large in size, then Chimera instruments at the granularity of a basic block. In this manner, Chimera avoids the risk of over-serializing the execution of loops.

4.1.3 Weak-Lock Design

Chimera ensures that the instrumented weak-locks do not introduce a deadlock. Chimera orders the set of weak-locks constructed for each granularity of a code region (basic block, loop, and function) and ensures that they are always acquired in the same order. When a program has nested code regions (e.g., a function calling a function, a loop calling a function, etc.), an outer region releases all its weak-locks before starting the inner region, and acquires the weak-locks back after exiting the inner region. The order in which weak-locks of different granularities are acquired is also consistent. Function-locks are always acquired before loop and basic-block locks. Loop-locks are always acquired before basic-block locks. Hence, there cannot be a deadlock between weak-locks.

Chimera avoids deadlocks that may happen when a weak-lock protected code region contains a programmer specified synchronization wait. The "weak" part of the weak-lock is meant for handling such deadlocks. If a weak-lock is stalled for more than a threshold period of time, the stalled weak-lock invokes a special system call to handle the potential deadlock. The system call handler identifies the thread that currently owns the stalled weak-lock by examining the log files used to record the order of weak-lock acquires and releases. The kernel preempts the current owner, and forces it to release and reacquire the weak-lock that timed-out. This allows the stalled thread to acquire the weak-lock and proceed with its execution.

Though the above mechanism may compromise the atomicity of a weak-lock protected code region, we always preserve the invariant that only one thread holds a given weak-lock at any given time. Thus, recording and replaying the exact order of forced weak-lock release and reacquire operations with respect to instrumented weak-lock operations is sufficient to guarantee deterministic replay. This requires that Chimera record and replay the exact instance when a thread is preempted and forced to release its weak-locks. For this purpose, we plan to use a mechanism from the DoublePlay replay system [75] in which the kernel records the instruction pointer and the branch count (measured via hardware performance counters) at the point of preemption. We have not yet ported this implementation to the Chimera infrastructure as none of our benchmarks have exhibited a weak-lock timeout.

4.1.4 Discussion

Any data-race that exists in the original program can manifest in the transformed program. However, Chimera now records the order between the racing instructions. Increasing the granularity of weak-lock (e.g., to a basic-block) would make it less likely for instructions from two racy basic-blocks to interleave. If there is only one race between two racy basic-blocks, then all thread interleavings in the original program can manifest at approximately the same probability in the transformed program. However, if there is more than one race between two basic

blocks, then Chimera's weak-locks will try to serialize them. While preventing fine-grained interleaving of smaller code regions may be beneficial for masking certain atomicity violations in production systems [44], a programmer trying to record and debug a test run might consider this to be a limitation of Chimera's optimizations.

4.2 Static Data-Race Detection

Chimera uses the RELAY [76] static data-race detector to identify potential data-races. In this section, we briefly summarize the RELAY detection algorithm, and then we discuss soundness and completeness of RELAY.

4.2.1 RELAY

RELAY is a lockset-based static race detection tool that scales to millions lines of code. A `lockset` for a program point is the set of locks held at that point. A lockset-based analysis assumes that for every shared object there is at least one common lock that is held whenever that object is accessed. The tool reports a race if a pair of memory accesses in different threads could access the same shared object, the intersection of their locksets is empty, and at least one of the accesses is a write.

We briefly summarize RELAY's analysis, but details can be found in the original paper [76]. RELAY starts by analyzing every leaf function in the static call graph ignoring the calling context. For each leaf function, it computes a summary. A function's summary soundly approximates the effect of the function on the set of locks held before the function execution. Also, it includes a summary of the set of shared objects accessed in the function and the lockset held during each of its accesses. For example, a summary of a function `bar(void *b)` may say that a write to the field `b->bob` can happen while holding a lock `b->lock`, and that the function releases the lock `b->lock` before returning. RELAY composes function summaries in a bottom-up manner over the call graph by plugging in the summaries of the callee functions to compute the summaries of the callers.

Thus, RELAY performs a bottom-up calling-context-sensitive analysis on the call graph to compute the access summaries for all functions that are thread entry points. This is done using a combination of flow-insensitive points-to [70, 6] and symbolic analysis.

4.2.2 Soundness

Chimera only instruments data-races found by the static data-race detector. Therefore, its deterministic replay guarantees are based on the soundness of the static data-race detector it uses.

RELAY has three potential sources of unsoundness, but they are modularized and each one can be addressed separately using known techniques. First, RELAY ignores memory operations that occur inside blocks of assembly code when calculating lockset summaries. However, this issue could be addressed with additional engineering that integrates memory access analysis for assembly instructions [9] with RELAY, or via manual annotations.

Second, the points-to analysis [70, 6] used by RELAY does not handle pointer arithmetic. RELAY's pointer analysis assumes that after any arithmetic operation on a pointer, the pointer still points to the same object. When this assumption does not hold true, the pointer analysis is not guaranteed to be sound. As a result, we can guarantee replay for an execution only until the first buffer overflow. However, this does not fundamentally affect Chimera's analysis. Enhancing pointer analysis to handle pointer arithmetic [78], or ensuring language safety would address this problem.

Finally, RELAY post-processes data-race warnings using unsound filters, but we do not use them.

4.2.3 False Positives

To provide soundness, RELAY makes conservative assumptions, resulting in its reporting a high number of false data-races. Instrumenting weak-locks for every false data-race results in prohibitively high overhead.

There are two main sources of false positives. First, RELAY accounts for lock synchronizations, but ignores happens-before relationships due to non-mutex synchronization operations such as fork/join, barriers, and conditional variables. As a result, RELAY may report a data-race between memory operations that can never execute concurrently. The second main source of false positives is due to the conservative pointer analysis it uses [70, 6]. Conservative pointer analysis would cause RELAY to underestimate the lockset held by a code region and overestimate the variables that could be accessed by a memory instruction.

Our experimental results in Section 4.6 show that RELAY reported data-race warnings on about 14% of memory operations in a dynamic execution. Instrumenting them with weak-locks to record the order of those potential data-races incurs an approximately 53x slowdown. We next discuss two important optimizations based on profiling (Section 4.3) and symbolic bounds analysis (Section 4.4) that significantly reduce this cost.

4.3 Profiling Non-Concurrent Functions

A static data-race detector may report races between code regions that are never executed concurrently. One reason for this is the inadequacy of the static analysis in accounting for non-

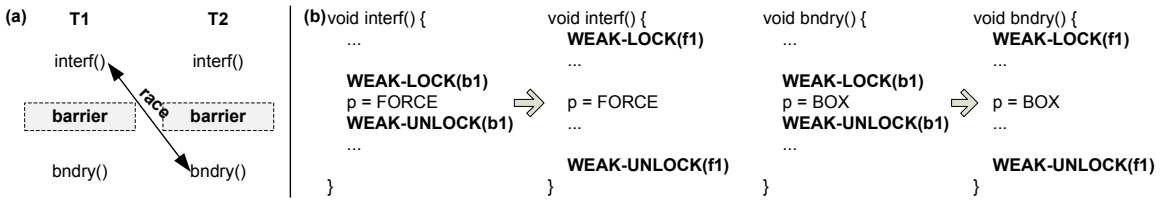

Figure 4.2: A false data race due to non-mutex synchronizations. (a) A false data-race reported for `water` application from the SPLASH benchmark [79]. Functions `bndry()` and `interf()` are never executed concurrently due to the barrier synchronization, which is not accounted for in RELAY. (b) The granularity of weak-locks is increased to function level in the two potentially racy functions because Chimera's profiler finds them to be non-concurrent.

mutex synchronization operations. To address this issue, Chimera uses a profile-guided analysis to determine code regions that are likely to never execute concurrently and use that information to increase the granularity of weak-locks without compromising an application's parallelism.

4.3.1 Overview

One important limitation of lockset based static data-race detectors, including RELAY, is that they account only for locks, but ignore happens-before relations due to non-mutex synchronization operations. Many false data-races may be reported due to this limitation. Figure 4.2(a) illustrates a false data-race reported for `water`. The data-race is false because the two supposedly racy functions are never executed concurrently due to a barrier synchronization. We also find that a number of false data-races are reported between initialization code and the rest of the code regions, as RELAY does not account for fork-join synchronization. Another source of false data-races, unrelated to non-mutex synchronizations, is the lack of static knowledge of control dependencies. For example, we found instances where a set of code regions are executed in only one thread, but RELAY reported false races among them. In all these cases, the two code regions containing the race-pair reported by RELAY are never executed concurrently.

We observe that such cases can be determined by profiling with a set of representative inputs. If a pair of potentially racy code regions are never executed concurrently in any of the profile runs, then there is sufficient confidence that they are likely to be *non-concurrent* in another execution. Profiling cannot guarantee that they will be non-concurrent in all executions. Nevertheless, we can take advantage of profiled information to increase the granularity of weak-locks to larger code regions and reduce the dynamic number of weak-lock operations.

If a pair of code regions containing a potential race-pair is likely to be non-concurrent, then Chimera increases the granularity of the weak-lock to protect the entire code region instead of just the basic blocks containing the race-pair. Figure 4.2(b) shows how this optimization affects the weak-lock instrumented to handle the false data-race that we discussed for `water`

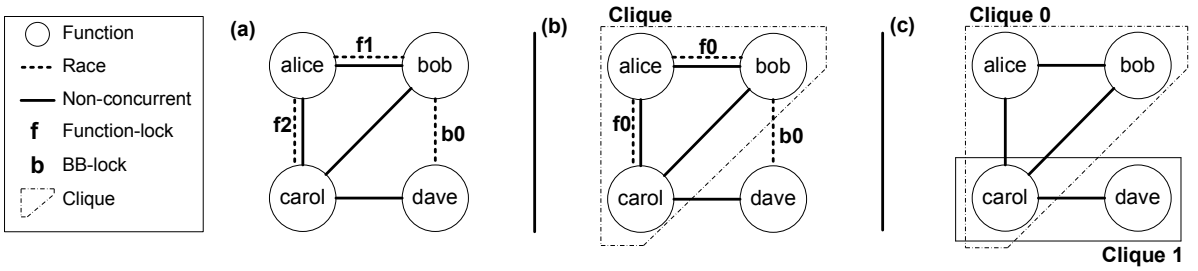

Figure 4.3: Clique analysis. (a) One weak-lock is instrumented for each race-pair. If a racy function-pair is non-concurrent, a function-level weak-lock (`f1, f2`) is used. Otherwise, a basic-block level weak-lock is used (`b0`). (b) Two potential data-races in a clique in a graph of non-concurrent function share one function-lock (`f0`). (c) Cliques in a graph representing non-concurrent functions.

(Figure 4.2(a)). In this study, we consider functions as code regions while performing non-concurrent region profiling, but our method could be applied for other region granularities as well. We refer to a weak-lock that protects a function as a *function-lock*.

By increasing the granularity of the weak-lock to the function-level, Chimera reduces the dynamic number of operations on that lock. Increasing the granularity in terms of the code region size for a weak-lock also creates the opportunity to use a single weak-lock to guard multiple potential data-races. The next section discusses an optimization that exploits this opportunity.

4.3.2 Clique analysis

We propose a clique analysis to determine which racy function-pairs can share the same function-lock. Sharing a function-lock reduces the cost of instrumentation.

Figure 4.3(a) shows a graph with a node for every function that contains at least one potential data-race. A dotted edge connects a pair of functions that could potentially race. A solid edge connects a pair of functions that are found to be non-concurrent in all of the profile runs. For example, `alice` is potentially racy and non-concurrent with `bob` and `carol`. Functions `bob` and `carol` are non-concurrent, but are proven to be race-free with each other. Functions `bob` and `dave` are racy and have also been found to be concurrent in some profile run.

One simple algorithm would be to assign a unique weak-lock for every racy-function-pair. If the race-function pair is also non-concurrent, then we can use a function-level lock as shown in Figure 4.3(a). Note that `bob` and `dave` could run concurrently, and so we do not use function-level weak-locks to guard potential races between them, as that could serialize those the two concurrent functions and compromise on parallelism. Instead, a weak-lock is instrumented at the basic-block granularity.

The above algorithm requires that `alice` acquires and releases two function-level weak-locks (`f1 and f2`) every time it is executed. However, `alice`, `bob`, and `carol` are po-

tentially non-concurrent with each other. Therefore, the two potential races could be guarded using a single function-lock `f0` as shown in Figure 4.3(b). This optimization would reduce the number of weak-lock operations.

To identify a group of functions which are mutually non-concurrent, we construct maximal cliques using a greedy algorithm in a graph of potentially non-concurrent functions (determined through profiling). A clique of an undirected graph is a subset of nodes where every node is connected to every other node. A maximal clique is a clique that cannot be extended by including one more adjacent node. Figure 4.3(c) shows a graph of potentially non-concurrent functions with two cliques, {`alice`,`bob`,`carol`} and {`carol`,`dave`}.

Once cliques are identified in a graph of non-concurrent functions, Chimera assigns function-locks as follows. For each race-pair, it checks if its racy functions are non-concurrent. If they are, then it finds the clique that the racy-function-pair is part of in the graph of non-concurrent functions. Chimera assigns the function-lock corresponding to that clique to both racy functions. For example in Figure 4.3(b), racy-function pairs {`alice`,`carol`} and {`alice`,`bob`} are both assigned a single function-lock `f0`. Notice that this weak-lock assignment is efficient for `alice` as it now has to acquire only one weak-lock as opposed to two. However, `bob` and `carol` are unnecessarily serialized (as they do not race with each other), which is still acceptable as they are also found to be non-concurrent during profiling.

It is possible that a racy-function-pair is part of two cliques. In that case, we use a greedy algorithm that chooses the weak-lock corresponding to the clique that contains the most number of racy-function-pairs.

4.4 Symbolic Bounds Analysis for Loops

Chimera's second optimization targets race-pairs that remain after applying the profile-based analysis described in the previous section. This optimization is based on symbolic address bounds analysis. It addresses the imprecision of the conservative but sound pointer analysis used in a static data-race detector.

4.4.1 Overview

Static data-race detectors [76, 38] use pointer analysis to determine the set of objects a memory instruction can access and also to determine the lockset at a program point. RELAY uses a combination of Steensgaard [70] and Andersen [6] flow-insensitive and context-insensitive pointer analysis, which are used in many static tools because they scale well to large programs. However, because these analyses are very conservative, RELAY overestimates the range of addresses that a memory instruction can access and underestimates the set of locks held at a

```
1:  for( i = 0 ; i < max_digits ; i ++ ) {
2:     WEAK-LOCK (&rank[0] to &rank[radix-1]);
3:     for ( j = 0 ; j < radix ; j++ ){
4:        rank[ j ] = 0;
5:     }
6:     WEAK-UNLOCK (&rank[0] to &rank[radix-1]);
7:
8:     WEAK-LOCK (-INF to +INF);
9:     for ( j = start ; j < stop ; j ++ ) {
10:       my_key = key_from[ j ] & bb;
11:       rank[ my_key ]++;
12:    }
13:    WEAK-UNLOCK (-INF to +INF);
14: }
```

Figure 4.4: Instrumenting weak-locks for a loop in the function `slave_sort` in `radix` using symbolic bounds.

program point, both of which cause it to report a number of false data races.

For example, we find a number of false data-races between two functions executed concurrently in different threads. This often happens when a programmer partitions work between threads, but the static analysis is unable to determine that the function will access different parts of a data structure. Figure 4.4 shows an example. RELAY reports a false data-race on the `rank` array in line 4 and 11, and also on `key_from` array in line 10. However, `radix` divides a large array into multiple portions and assigns different portions to concurrent threads to process them in parallel. Therefore, the base address of `rank` and `key_from` are different for each worker thread, and hence the threads do not access the same entry in those arrays concurrently.

It is hard to statically determine the absolute values of address bounds of an object accessed by memory operations in a code region. However, it has been shown that the lower and upper bounds in the form of a symbolic expression can often be derived statically [64, 65] with much better accuracy. Chimera uses this information to increase the granularity of weak-locks that it must instrument for race-pairs in concurrent code regions. The weak-lock is constructed in such a way that it protects a code region for a range of addresses specified by a symbolic expression. Thus, two potentially racy code regions can execute concurrently (provided our symbolic bounds are accurate enough). At the same time, Chimera can protect the regions with weak-locks instrumented at larger granularities to reduce the number of operations.

Figure 4.4 shows an example. RELAY reports that `line 4` could race with itself. Instrumenting a weak-lock inside the loop would be very expensive. Instead, Chimera instruments a weak-lock that provides mutual exclusion for the entire loop (lines 3-5) only for the address range from `&rank[0]` to `&rank[radix-1]`. This range is computed by a sound static symbolic address bounds analysis, which we discuss in the next section.

4.4.2 Symbolic Bounds Analysis

We implemented our symbolic bounds analysis based on the algorithm proposed by Rugina and Rinard [64, 65]. The goal of this analysis is to determine the symbolic expressions that specify the upper and lower bounds for a pointer or array index variable at a program point that is found to be potentially racy by the static data race detector. For the example in Figure 4.4, the analysis determines that the symbolic lower bound of j of the first inner loop (line 4-7) is 0 and the upper bound is the initial value of radix $radix_0 - 1$. It also finds that `line 4` can access a memory region from `&rank[0]` to `&rank[radix-1]`. Details about the algorithm are discussed by Rugina and Rinard [65].

The effectiveness of our optimization depends on the accuracy of the lower and upper bounds. The analysis we use is sound, but imprecise. If the bounds are too conservative, we may serialize concurrent code regions unnecessarily. There are two main sources of imprecision. The first case is when the address of the racy object is based on the value of a variable that cannot be determined outside the code region. For example, precise symbolic bounds for the `rank` array accesses in the second inner loop (line 9-12) cannot be determined. The value of the index variable my_key cannot be computed outside the loop as it depends on the value read from another array key_from inside the loop (line 11). However, we can derive the symbolic bounds for the array key_from accurately. The second source of inaccuracy is when the racy object's bounds depends on an arithmetic operation (e.g., the modulo operation or logical AND/OR) not supported in the analysis.

4.4.3 Choosing the Granularity for Code Region

Rugina and Rinard's analysis [64, 65] describes a generic algorithm for larger code regions including inter-procedural analysis, but, as a first step, we applied their technique only for loops with no function calls in the loop body. As a result, our current implementation may not exploit all opportunities for optimization.

If the symbolic bounds are too imprecise, care must be taken to ensure that we do not over-serialize loops. If the derived symbolic expression for an address range is from negative infinity to positive infinity, we consider it to be *too imprecise* to be useful. Otherwise, we consider it to be *precise enough*. In that case, we balance the number of weak-lock operations with the loop serialization cost.

If the symbolic bounds of a racy loop is precise enough, we assign a weak-lock at the loop granularity (the first inner loop in Figure 4.4). If the bounds are too imprecise, we estimate via profiling the average number of instructions executed by a loop iteration. If the estimate is less than a `loop-body-threshold`, we still instrument at the loop granularity because the cost of operations on the weak-lock does not warrant exposing parallelism in the loop. Otherwise, we

instrument a basic-block lock inside the loop body. If a loop is nested, we select the outermost loop with precise enough bounds.

4.5 Implementation

This section presents the implementation details of the Chimera record and replay system.

4.5.1 Analysis, Instrumentation, and Runtime System

Our analysis and instrumentation framework is implemented in OCaml, using CIL [53] as a front end. To profile concurrent function pairs (Section 4.3), we instrumented the entry and exit of each function using CIL's source-to-source translation. To statically derive symbolic bounds of racy loops (Section 4.4), we also performed data flow analysis on a racy loop and produced linear programming constraints using CIL. Then, we used `lpsolve` [2], a mixed integer linear programming solver, to find a solution for static bounds that a racy loop may access. Finally, based on the results of the above static analysis, we used CIL to instrument weak-locks at the function, loop, basic block, or instruction granularity.

We modified the Linux kernel to record and replay non-deterministic input from system calls and signals. We also modified GNU pthread library version 2.5.1 to log the happens-before order of the original synchronization operations and the weak-locks added by Chimera.

4.5.2 Static Analysis and Source code

We used RELAY [76] to perform pointer analysis and to collect a set of potential data-races. We applied Andersen's inclusion-based pointer analysis [6] to resolve function pointers, and Steensgaard's unification-based approach [70] to perform alias analysis between lvalues. While performing pointer analysis, RELAY first translates function local arrays and address-taken variables to heap variables (making them global) in order to derive pointer constraints in a unified manner. RELAY performs static analysis on this modified source code. This can lead to unnecessary false data-races on local variables. To resolve this, we filtered out race warnings on a 'heapified' local variable that did not escape its function.

To perform sound static analysis, we made sure that all library source code (except for `apache` and `pbzip2`) are included in our static analysis. For the standard C library, we used uClibc [73] which is smaller and easier to analyze than the GNU glibc library, as it is developed for embedded Linux systems. The uClibc library involves all the necessary functions such as `libc` and `libm`.

For `apache`, we did not include libraries such as `gdbm`, `sqlite3`, etc., because they do not contain code that gets executed for the input we use in our study. It is possible that the

application		LOC	profile environment	evaluation environment
desktop	aget	1.2K	2 workers, download a 29KB file from local network	2,4,8 workers, download a 10MB file from http://ftp.gnu.org
	pfscan	2.1K	2 workers, scan 236 KB of small 22 files	2,4,8 workers, scan 952 MB of 8 log files
	pbzip2	4.8K	2 workers, compress a 219 KB file, output to stdout	2,4,8 workers, compress 16 MB file, output to file
server	knot	1.3K	2 workers, 4 clients, 100 requests, 29KB file	2,4,8 workers, 16 clients, 1000 requests, 390KB file
	apache	99K	2 workers, 4 clients, 100 requests, 29KB file	2,4,8 workers, 16 clients, 1000 requests, 390KB file
scientific	ocean	5.3K	2 workers, 130*130 grid, 1e-01 error tolerance	2,4,8 workers, 1026*1026 grid, 1e-07 error tolerance
	water	2.5K	2 workers, 64 molecules, 5 steps	2,4,8 workers, 1000 molecules, 10 steps
	fft	1.4K	2 workers, 2^4 matrix , no inverse FFT check	2,4,8 workers, 2^{20} matrix, with inverse FFT check
	radix	1.3K	2 workers, 2^8 keys , no sanity check	2,4,8 workers, 2^{14} keys, with sanity check

Table 4.1: Benchmarks and input used for profiling and evaluating Chimera. The number of lines in the source program (LOC) is measured for the CIL representation. It does not include the size of library code: libc(41.7K) and libm(3.6K).

source code of a third party library may not be available for static analysis. When any part of the source code of a library used by a program is not analyzed, the soundness of static analysis may be compromised. One solution is to ask library builders to provide annotation (lockset summaries) for their library functions so that it can be fed into RELAY to perform a sound data-race analysis. Developing such annotations would be an one-time cost for library builders, and it would not place any burden on software developers that use those libraries. Another possible solution is to assume that a library function will only access the set of objects pointed to by the parameters passed as function arguments without acquring any new locks. However, this approach is not guaranteed to be sound, because a library could retain pointers passed to previous calls to the same library. Also, instructions in a library's function can have a data-race on some shared-variable that is internal to the library. We employed the latter approach for `pbzip2` (we excluded the `libbz2` library used by `pbzip2`)

We also converted the C++ `pbzip2` program into ANSI-C code by replacing the vector STL container with a linked-list-based C library, because our instrumentation framework, CIL [53], can only handle C programs, but not C++ constructs.

4.6 Results

This section evaluates Chimera's recording and replaying overhead and demonstrates the effectiveness of the profiling and symbolic bounds optimizations.

4.6.1 Methodology

We evaluated our system using three sets of benchmarks which are listed in Table 4.1. The first set consists of three desktop applications: `aget`, `pfscan`, and `pbzip2`. The second set has two web sever programs: `knot` and `apache`, which are evaluated using the ApacheBench (ab) client. The final set contains four scientific programs from SPLASH-2 [79]: `ocean`,

application		DRF Logs		logging order of potential data-races				performance				log size	
		system calls	synch. ops.	instr. log	basic blk. log	loop log	func. log	original time(ms)	record time(ms)	recording overhead	replay overhead	input log(KB)	order log (KB)
desktop	aget	16604	8424	28876	5191	15939	32416	5058	5114	1.01	0.06	20072	361
	pfscan	109	879	8	0	39	347	848	881	1.04	1.02	2	3
	pbzip2	592	2491	2621	81	1177	1540	1343	1371	1.02	1.03	1989	26
server	knot	8056	32	5136	0	251	2257	7137	7176	1.01	0.01	84	23
	apache	18301	36812	798891	266956	565863	1123337	18668	19376	1.04	0.02	178	6469
scientific	ocean	2750	9978	6237	8233	287642	37655	2328	5585	2.40	2.24	16	727
	water	10295	67202	21838	1409884	198993	1112798	1665	2820	1.69	1.75	101	12744
	fft	113	193	1843	38	49718	11595	586	1249	2.13	2.23	2	107
	radix	102	312	3	13	344	393	1599	1939	1.21	1.20	1	3

Table 4.2: Chimera record and replay performance. The results are the mean of five trials with 4 worker threads.

`water`, `fft`, and `radix`. To collect a set of concurrent function pairs for clique analysis (Section 4.3.2), we profiled each program 20 times with various inputs. The inputs used for profiling are significantly different from the input used for our performance evaluation.

Chimera is scalable to large programs. It is built on RELAY [76], which has been shown to scale to very large programs (e.g., Linux with 4.5 million lines of code). Chimera also uses static analysis to derive symbolic bounds, but it is a scalable intra-procedural analysis. Our benchmark set includes some fairly large programs. Table 4.1 provides the number of lines (LOC) of our benchmarks in their CIL representation. It does not account for the size of library code: libc(41.7K) and libm(3.6K).

Presence of assembly code and buffer overflow may compromise the soundness of static data-race analysis (Section 4.2.2). However, the programs we evaluated do not contain assembly code, except for a few library functions. Also, we are not aware of any buffer overflow bugs in our benchmarks. Also, we did not observe any weak-lock timeouts (Section 4.1.3) in any of our experiments.

We ran our experiments on a 2.66 GHz 8-core Xeon processor with 4 GB of RAM running CentOS Linux version 5.3. We modified Linux 2.6.26 kernel and GNU `pthread` library version 2.5.1 to support Chimera's record and replay features. All results are the mean of five trials with 4 worker threads (excluding main or control threads). Section 4.6.2 presents scalability results for which we used 2, 4, and 8 threads.

4.6.2 Record and replay performance

Table 4.2 shows Chimera's record and replay performance when all the optimizations (function, loop, and basic-block level weak-lock optimizations) are enabled. The first set of columns quantifies the number of logs generated for recording program input (read through systems calls) and the happens-before order of synchronization operations. These logs are sufficient to guarantee replay for data-race-free (DRF) programs. The second set of columns quantifies the number of logs due to various types of weak-locks. The next set of columns presents the

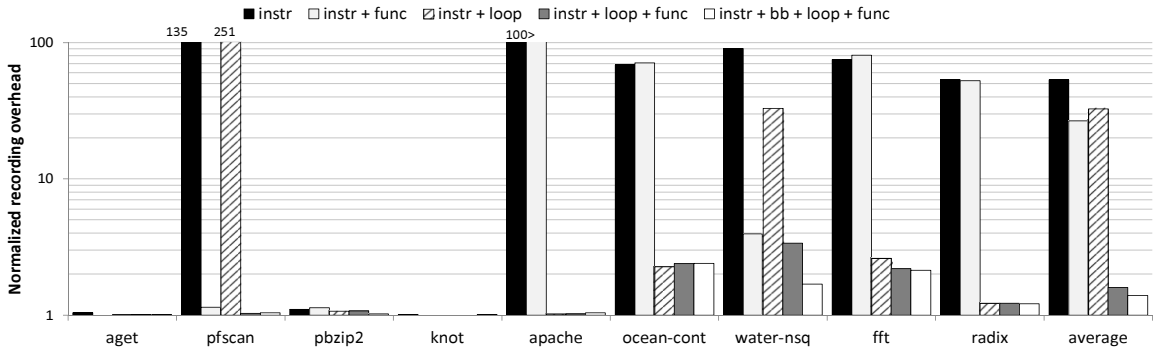

Figure 4.5: Normalized recording overhead for Chimera with different sets of optimizations

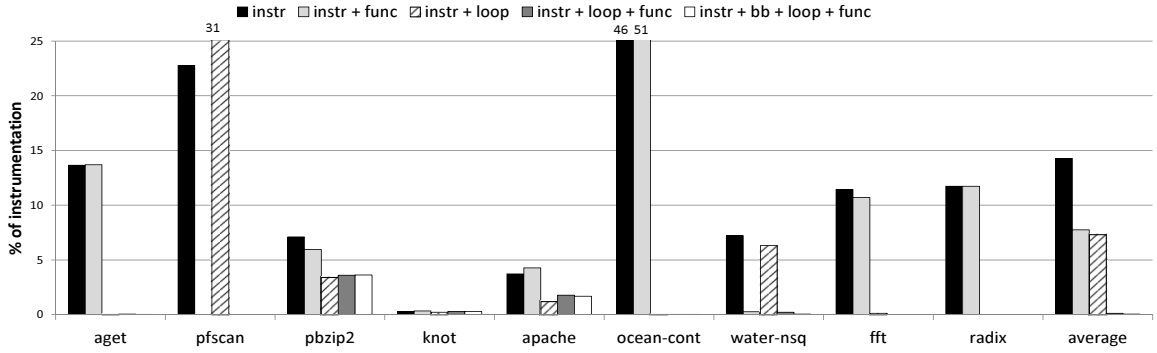

Figure 4.6: Proportion of instrumentation points for different logging schemes

performance overhead. The last set of columns quantifies the `gzip` compressed log sizes for recording the program input and the order of all synchronization operations (including weak-locks).

Chimera incurs negligible overhead for desktop and server applications. For scientific applications (with high frequency of accesses to shared variables) the overhead is relatively high. On average, our system incurs 40% performance overhead to record an execution with four worker threads. Replay overhead is similar to that of recording overhead for most applications, except for I/O intensive applications. Network intensive applications such as `aget`, `knot`, and `apache` replay much faster as we feed the recorded input directly to the replayed process without waiting for the network response. Chimera's performance overhead is an order of magnitude improvement over the state-of-the art software solutions that guarantee multiprocessor replay [75].

Log sizes of Chimera are within acceptable limits for various uses of replay. `aget` produces large logs because the contents of all the downloaded files are in the log. `water` also produces a large log size because of frequent user specified synchronizations and weak-locks.

4.6.3 Effectiveness of Optimizations

We analyzed the effect of different optimizations on recorder's overhead. Fine grained weak-locks (instruction and basic-block level weak-locks) enable higher concurrency, but they increase the number of program points instrumented resulting in higher performance and log size overhead. The opposite is true for coarser grained weak-locks such as function and loop level weak-locks. We use function-level weak-locks if two functions are likely to be non-concurrent (Section 4.3). We use loop-level weak-locks with runtime bounds checks if our static analysis can derive precise enough symbolic bounds (Section 4.4).

Figure 4.5 shows the performance overhead of Chimera's recorder with different sets of optimizations normalized to native execution time. As expected, instrumenting every potential data-race at the granularity of a source line (labeled as 'instr') incurs 53x slowdown. However, when we apply the profile-based optimization to increase the granularity of some weak-locks to function level ('inst+func') the overhead drops to 27x. If we use only symbolic analysis to coarsen the granularity of some weak-locks to loop level results in 33x overhead. However, when we employ all the optimizations together ('inst+bb+loop+func'), including basic block level weak-locks, the average overhead drops significantly to 1.39x.

Applications such as `pfscan` and `water` benefit significantly from function-level locks. In these applications, most data-races are in function-pairs ordered by some non-mutex synchronization operations that our static analysis could not account for. For applications such as `apache`, `ocean`, `fft`, and `radix`, loop-level locks reduce the recording overhead drastically. For example, in `apache`, RELAY reports a false data-race between memory operations within a hot loop in the `memset` library function that iterates approximately over 6 million times in our experiments. Function-level weak-lock is ineffective in this case, because two threads may execute the `memset` concurrently. However, our static analysis determines the bounds of addresses accessed within the hot loops of `memset` fairly accurately, which enabled us to use loop-level weak-locks effectively. We also observe noticeable benefits in coarsening weak-locks from instruction-level to basic-blocks (e.g., `water`).

Finally, for network applications like `aget`, `knot`, and `apache`, recording cost overlaps with I/O wait resulting in negligible overhead. Chimera could be used even in production systems for such applications.

Figure 4.6 shows the proportion of dynamic number of weak-lock operations with respect to the total number of dynamic memory operations. A naive dynamic data-race detector would have to instrument 100% of memory operations. This result shows the advantage of static data-race analysis and our optimizations in terms of reducing the number of instrumented points in the program. In general, results in Figure 4.6 for different optimizations are consistent with the recording overhead in Figure 4.5. This indicates that the savings obtained from coarser weak-locks was not overshadowed by any loss in parallelism.

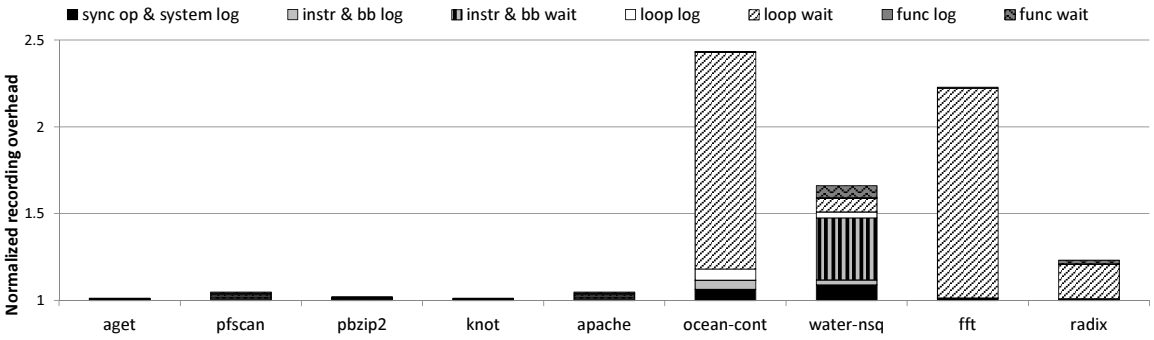

Figure 4.7: Sources of recording overhead

On average, naively monitoring all data-races reported by the static data-race detector requires us to instrument about 14% of all dynamic memory operations. By increasing the weak-lock granularity to function, loop, and basic block levels, we can reduce the proportion of weak-lock operations with respect to memory accesses down to 0.02% on average. In general, this result shows that increasing the granularity of weak-locks reduces the instrumentation cost.

However, in some fairly rare cases, increasing the locking granularity from instruction to loop-level may increase the frequency of weak-lock operations. The reason is due to control flow dependencies. In `pfscan`, there is a racy instruction inside a hot loop that is guarded by an `if` statement. If we use loop-level weak-lock, Chimera has to always perform weak-lock operations when the loop is executed. But if we use instruction-level weak-lock, the instrumented code will be executed only when the `if` condition gets satisfied.

In `apache` the number of weak-lock operations increase when we go from instruction-level to function-level granularity. The reason for this is behavior is best explained using a contrived example. Assume that RELAY finds a data-race between each of the functions `foo`, `bar`, and `qux`. Also, assume that all these functions are non-concurrent with each other, except for the function-pair `bar` and `qux`. For this example, Chimera will assign two different function level weak-locks (one for `foo-bar` and another for `foo-qux`). This allows `bar` and `qux` to run concurrently. As a result, `foo` is instrumented with two function-level locks, which may be more costlier than using one instruction-level lock if there is only one racy instruction inside `foo`.

We also studied the sensitivity of our profile-based non-concurrent function analysis to the number of profile runs. We did this study only for `pfscan` and `water-nsq`, because other applications shows little performance benefit from function-level logging (Figure 4.5). For these two applications, the number of concurrent function pairs observed quickly saturates after a small number of profile runs (five for `pfscan` and three for `water-nsq`).

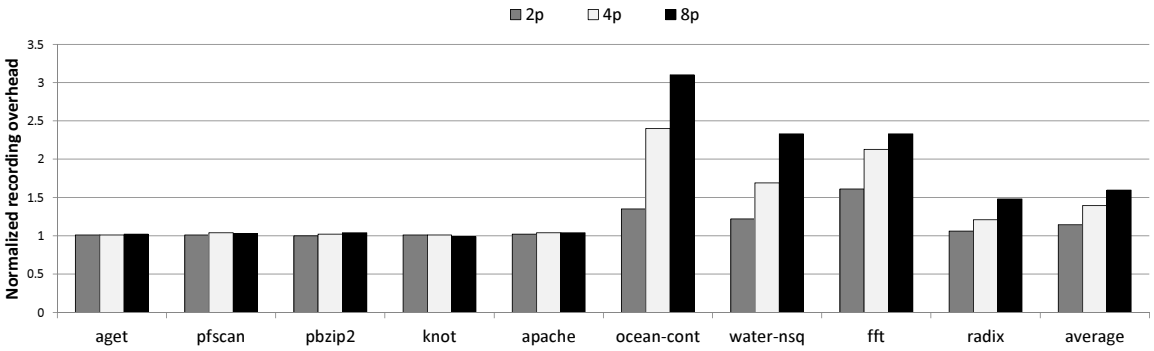

Figure 4.8: Scalability results on 2, 4, and 8 processor executions

4.6.4 Sources of Overhead and Scalability

Figure 4.7 provides a breakdown of the remaining sources of performance overhead in the Chimera recorder that incorporates all of our optimizations ('inst+bb+loop+func'). The results are normalized to the native execution time. We measure the performance of our system by instrumenting each type of weak-lock one by one. The performance overhead due to a weak-lock type is further broken down into the cost of logging the weak-lock operations and the cost due to weak-lock contention. To measure the time lost due to weak-lock contention, we subtracted the execution time of a program execution in which a weak-lock acquire operation always succeeds without waiting from the execution time of a program execution in which the weak-locks semantics are obeyed.

Contention for loop-level weak-locks dominate the overhead for scientific applications such as `ocean` and `fft`. The reason is that our static symbolic bounds analysis is not very precise for some performance critical loops in these programs because they tend to execute irregular array accesses and unmodeled arithmetic operations (Section 4.4.2). As a result, the bounds checks performed as part of loop-lock acquire operation over-serializes the execution. We also fail to use loop-level lock and resort to instruction-level logging (e.g., `water`), if the loop body contains a function call, because our symbolic analysis in intra-procedural.

Contention between loop-level weak-locks is the for increase in performance overhead as the number of threads increases for some applications (Figure 4.8). We believe that source-level inlining for small functions or inter-procedural symbolic bounds analysis could help reduce this overhead. Nevertheless, as we discussed earlier (Section 4.6.3), our current analysis already provides significant benefits with loop-level locks for many applications (e.g., `ocean`).

4.7 Conclusion

Chimera is the first software system for multiprocessors that leverages a static data-race detector tool to provide a low overhead replay solution. However, existing static data race de-

tectors generate enumerous false warnings because 1) static data race analysis cannot reason about non-mutex happens-before relations (such as fork-join, barriers, and signal-wait), 2) and a sound pointer analysis is necessarily conservative and thus too imprecise. Chimera resolves this problem by employing a combination of profiling, symbolic analysis, and dynamic checks that target the sources of imprecision in the static data race detector.

Chimera's transformations ensure that the resultant code is data-race-free when instrumented with the new set of synchronization operations. We believe that this technique could also prove quite useful for enabling stronger semantics for concurrent languages such as sequential consistency and for enabling deterministic execution.

CHAPTER 5

Rosa: Hardware Support and Offline Symbolic Analysis for Multiprocessor Replay

In Chapter 3 and Chapter 4, we introduce efficient software-only replay solutions: Respec and Chimera. Processor support could enable us to build ultra-low overhead ($<1\%$) replay solutions. However, hardware solutions should be complexity-effective enough that processor vendors are encouraged to include a deterministic replay feature in the next-generation processors.

In this Chapter, we propose a new hardware-assisted record-and-replay solution ,called Rosa, that does not detect and log shared-memory dependencies at all. Moreover, our system supports replay under the most commonly implemented relaxed memory model – Total Store Order (TSO). We first show that if we use a load-based checkpointing mechanism for recording program input [51], there is no need for shared-memory dependencies in order to replay each thread in isolation. The reason is as follows. For each thread in a multi-threaded program, a load-based checkpointing mechanism records the thread's initial register state and the values of a subset of the load instructions executed by the thread. We observe that this information alone is sufficient for deterministically replaying each thread in isolation, independent of the other threads. By deterministically replaying each thread in isolation, we can reproduce the exact same sequence of instructions executed by a thread during recording, as well as reproduce the input and output values of those instructions (input of a memory operation includes its address as well). Such mechanism works for any type of underlying memory consistency model as long as cache coherence is preserved.

However, replaying each thread in isolation is not sufficient for debugging a multi-threaded program, as programmers need to understand the interactions between the threads. This would require reproduction of shared-memory dependencies during replay. This information, however, can be determined using an offline analysis, which works as follows.

By using load-based checkpoints, we can obtain the trace of all the memory operations executed by a thread along with the address and input/output values of those memory operations. From the final core dump, we also obtain the final state of every memory location. Using all of this information, we determine the memory ordering constraints, compliant to underlying

memory coherence and consistency model, for all the memory operations executed by all the threads. These constraints are then encoded in the form of a satisfiability equation that can be solved by a SMT (Satisfiability Modulo Theory) solver. The solution represents the causal order of the memory operations executed by all the threads, using which a programmer can reason about the dependencies between the threads.

The equation consist of two types of constraints. The *coherence constraints* enforce that any memory access *M*'s old value should be same as the new value of the memory access to the same location that immediately precedes *M* in the derived causal order. The *consistency constraints* are specified by a particular memory model. For sequential consistency (SC) memory model, every memory access in a processor should follow the strict program order. On the other hand, Total Store Order (TSO) differs from SC in that it relaxes store-to-load program order and store atomicity [7]. The first relaxation allows a load to be scheduled ahead of an earlier store in the same thread provided they are accessing different locations. The second relaxation allows a store's value to be made visible to a local load before it is made visible to remote loads. We discuss how these two relaxations can be encoded as first-order logic formula and reproduce a TSO-compliant causal order using an SMT solver.

To bound the search space during offline analysis, we propose to log certain hints during recording. At periodic intervals, all the processor cores simultaneously record the number of committed memory operations along with the number of stores pending in its local store buffer. Using this information, we show that we can legally partition a multi-threaded program execution into smaller bounded intervals and determine a causal order for memory accesses in each interval separately. In our mechanism, hints needed for bounding the search can be logged without any additional communication between the processor cores. To further reduce offline analysis time, our offline analysis eliminates a majority of cache hits from the offline search. This is based on our observation that the causal order for most cache hits can be trivially inferred from the causal order between cache misses.

By sacrificing precision in logging causal order, we manage to design a low-complexity processor solution. The tradeoff is the offline analysis cost. However, offline analysis need to be performed only once. Once shared memory dependencies are resolved, later replays can be very efficient. We believe that developers would be willing to pay an one-time cost to reproduce a bug (by replaying a few seconds that preceded a crash) that manifested in the production sites and during beta-testing which is where a low-overhead processor recording solution would be crucial.

The rest of the paper is structured as follows. Section 5.1 discusses how load-based checkpoint scheme is sufficient for replaying each thread individually. In Section 5.2, we describe offline analysis for reporducing the causal order for the shared-memory operations across all the threads. Section 5.3 presents a solution to bound the search space and Section 5.4 proposes

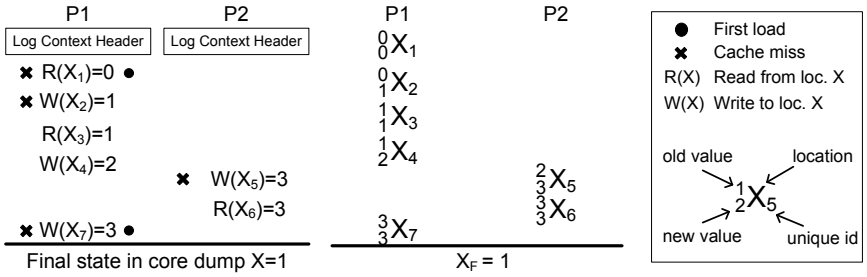

Figure 5.1: Load-Based Logging Example

a filtering technique that reduces offline analysis time. Then, Section 5.5 shows experimental results and Section 5.6 concludes this chapter.

5.1 Load-Based Checkpointing Architecture

The load-based checkpointing scheme was originally proposed in the BugNet architecture [51] as an alternative to the system-dependent logging scheme for recording program input. The original goal of BugNet was to avoid the system complexity in detecting and recording all types of non-deterministic system input. In this paper, we show that there is an added advantage to using a load-based checkpointing as we do not have to record shared-memory dependencies. These dependencies can be determined offline by analyzing load-based checkpoint logs.

In this section, we briefly describe our load-based checkpointing scheme that is a modified version of BugNet [51]. We extend the original design to support replay of the full system and programs with self-modifying code. We discuss a complexity-effective architecture design to support this scheme efficiently. We also describe its unique property that allows us to replay each thread in a multi-threaded program in isolation without the information about shared-memory dependencies. Then we describe additional architectural support required for logging hints that help us bound the complexity of our offline analysis described in Section 5.2.

5.1.1 Load-Based Program Input Logging

Let us first consider recording a single-threaded program's execution on a uni-processor system without any system events such as I/O, DMA, context switches or interrupts. A key insight in BugNet [51] is that to replay an interval of a program's execution, it is sufficient to record the program's initial register state, and then record the values of all the load instructions executed by the program during the interval. The value of a load instruction is recorded along with the instruction count corresponding to the load instruction. The instruction count of a memory instruction is the number of instructions that the program had executed since the beginning of the recorded interval. Unlike in BugNet [51], we also consider instruction read as a load. This

extension allows us to handle programs with self-modifying code.

Using the recorded log, a tool like Pin [45] can be used to perform the replay. The replayer emulates the states of the register and the virtual memory of the recorded program. It starts by initializing the register states, including the program counter, by reading from the log. All the memory states are initialized to invalid. Once initialized, the first instruction specified by the program counter is executed. Since our system treats an instruction read as a load, its machine code can be found in the recorded log. An instruction is replayed according to its type.

- *Non-memory operations* are executed by reading the input values from the emulated register states, and then writing the result back to the emulated register states.
- For a *load instruction*, its effective address is computed from the input register states. In addition, its value is read from the log and the emulated memory state is updated with that value.
- For a *store instruction*, its input value is read from the emulated register state, its effective address is computed, and then the emulated memory state is updated with the store value (so that later loads to the same location can get their values).

Thus, a program is deterministically replayed with exactly the same sequence of instructions along with their input and output values. The register and memory states for the program is also deterministically reproduced at every instruction replayed.

Recording every load value (including instruction reads) is expensive in terms of log size. In order to alleviate the problem, BugNet logs a load only if the load is the first memory access to the location. Such loads are called as *first-loads*. The values of non-first-loads need not be logged, as they can be read from the emulated memory state. Figure 5.1 shows a sample execution. Assume that all the instructions in the example access the same memory location `X`. Consider just the first four instructions executed by the processor `P1` for now. `R(`X_1`)` is the first-load (with a return value `1`), and it is logged (indicated by the solid dot on the right-side of the instruction). The values of the next three memory operations are not logged as they can be deterministically replayed using the emulated memory state.

In order to log an execution, the operating system first creates a checkpoint by recording the *context header* and turns on logging for the processor core. The context header contains the initial register state, a process identifier and the value of the timestamp counter of the processor core.

To detect and log first-loads we need processor support. BugNet [51] uses a bit per cache word in the private cache of a processor core to determine whether the location has been logged for the program. We use an even simpler design, where we just log the cache block fetched on a (load or store) cache miss, because any first access to a location would result in a compulsory

cache miss. In the case of a store miss, the data recorded for the cache block are the values before executing the store.

In BugNet, a memory location's value need not be logged if the first access to it is a store. Because, any store, including the first-store, can be deterministically replayed using the emulated register and memory states. However, we log the cache block fetched on a store miss if it is not due to upgrade store miss (e.g. `W(`X_2`)`), because it is possible that later on, the program could execute a first-load to a different word in the same cache block. For the upgrade store misses, we do not record the data, as it can be obtained from the previous load miss during replay. This might slightly increase the log size when compared to the BugNet design. But it avoids the need to use a bit per cache word, and also helps our offline analysis explained in Section 5.2. `W(`X_5`), and W(`X_7`)` operations (denoted with cross marks) in Figure 5.1 are (non-upgrade) store misses and are logged in our design.

The data logged for a memory access consist of the instruction count of the memory operation that causes the cache miss, and the data of the cache block fetched. To simplify the design, we choose to not use any additional local log buffers. Instead, we directly write-back the cache block to the log space allocated in the main memory. Note that any read from an uncacheable memory-mapped location would always be logged as it will always result in a cache miss. Thus, non-deterministic input read from system devices such as network cards are correctly captured. Also, `RDTSC` (Read TimeStamp Counter) instruction in the x86 architecture is also treated as a uncacheable load, and its return value is recorded.

A checkpoint for a program is created first when logging is turned on for that program. Thereafter, a new checkpoint is created at regular intervals. The checkpoint interval length is defined based on the available memory space for logging similar to the original BugNet architecture [51]. To create a new checkpoint, the operating system flushes the data in the private caches of the processor, logs the checkpoint header, and then continues to log the data of every cache block fetched on a cache miss.

5.1.2 Handling System Events

The previous section assumes a uni-processor system, and also that there are no system events that affect a program's execution. We now relax the latter constraint. Unlike BugNet [51], we choose to record the execution of the full system including the operating system code. An interrupt, a system call, or another program can context switch a program executing on a processor core.

On a context switch, the operating system terminates the current log by logging the current instruction count for the processor core (so that the replayer would know when to context switch during replay). It then logs a context header for the new program that is context switched in.

Now the context header contains the initial register state of new context including the program counter, the process identifier, the processor timestamp, and the instruction count of the preivous context header is updated.

Recording initial register state and the cache miss data along with the instruction count of the memory operation is sufficient for reconstructing virtual memory address space. However, to support full system replay, we should be able to reconstruct physical address space faithfully. One possible solution would be recording physical address as well during cache miss logging. During replay of the full system, when a program accesses a virtual address for the first time, we can determine its equivalent physical address from the log. Thus, we can establish a map between the virtual and physical addresses for a program during replay and emulate the physical address space. However, there are two problems. First, the size of program input log would increase, because we have to save 32bit or 64bit physical address for each log entry. Second, logging physical address on cache miss is not sufficient for deterministic replay on physically tagged private caches. In this case, on a context switch, private cache blocks are not flushed, so the first access to a virtual memory location by the newly context switched in program might not result in a cache miss. This implies that we cannot reconstruct the physical address from the replayed virtual address. Moreover, the mapping between physical addresses and virtual addresses could change after a page fault.

We solve this problems by letting the operating system record page tables on context switch while recoring context header and update new mapping on page faults. Since we record the mapping at page granularity, this approach would generate much smaller log size than logging physical memory addresses on cache misses. We may record TLB misses to replay physical addresses, but this approach would increase hardware complexity.

Replaying by emulating physical address space also allows our offline analysis to correctly determine shared-memory dependencies between multiple processes (and of course threads) that concurrently run on different cores.

To replay a checkpoint interval, the replayer starts from the first context header and continues to emulate the register state and the physical memory state of the system. When we find a record for a memory access in the log during replay, the replayer determines the physical address of the memory state from the logged page tables. When the execution during replay reaches the next context header (determined by comparing the emulated instruction count with the instruction count that was logged on a context switch), the emulated register state is updated with the values from the next context header. Then the replay proceeds normally.

The above approach ensures replay of the full system execution on a processor core for an interval. We can replay on any operating system as long as we have a tool that emulates the ISA (Instruction Set Architecture) of the recorded processor. Using the process identifier logged in the context header, the replayer could provide the programmer with information about which

application is replayed at any instant.

5.1.3 Multi-Processor Replay

We now discuss support for recording a full system execution on a processor with multiple processor cores (which includes a DMA processor as well). Each processor core has log space allocated to it in the main memory by the operating system. To start recording for a checkpointing interval, the operating system first records the context header for each core, and then lets each core log their cache misses into the private log allocated to it. When a thread on a core is context switched out, the operating system performs the same tasks that we described earlier for a uni-processor system.

Consider the logs of two processors shown in Figure 5.1. All the memory operations shown in the figure access the same memory location. The memory operations marked with a cross are the ones that result in a cache miss, and therefore result in a log record. Notice that there are shared-memory dependencies between the two executions. In any cache coherent multiprocessor, before a node can write to a memory block, it has to first gain exclusive permission to that cache block. This results in invalidation of cache blocks privately cached in all the other nodes. As a result, when a processor core tries to read a value that was last written by another processor core, it triggers a cache miss. `W(`X_7`)` and `W(`X_5`)` shown in the figure are examples. Thus, our logging mechanism implicitly captures the new values produced by remote processors. This is the key property that allows us to replay the execution of a processor core independent of the other cores. We achieve this without any changes to the coherence protocol.

To replay the execution of `P1` in this two processor multi-core system, the replayer simply takes the log recorded by `P1`, initializes the register state, and starts the replay. The replay produces exactly the same sequence of instructions as in the recording phase, along with the input and output values of those instructions. For each memory operation, the replayer can determine its memory address. Also, it reproduces the value read or written by a memory operation, which we refer to as the *new* value for the memory operation. Finally, as we described earlier, for a write cache miss we log the value before it is modified by the write. Thus, the replayer can also reproduce the `old` value for a memory operation, which would be the value of the memory location before it was modified by the memory operation.

Thus, without any additional support for a multi-processor system, just by using the program input log for each processor core, the replayer reproduces the exact same sequence of memory operations that were executed during recording, along with their addresses, old and new values. The figure on the right in Figure 5.1 shows the information reproduced after replaying the execution of the two processor cores. The memory operations are labeled using the address location that they access (in this example, all the accesses are to the same location `X`). The left

super-script of a memory operation denotes its old value, and the left sub-script denotes its new value.

The operating system also records the final memory system state at the end of recording (similar to the core dump collected after a system crash). In the example, the final state of `X` is `3`. In Section 5.2, we discuss how all this information can be used to determine the shared-memory dependencies.

5.1.4 Discussion

We summarize the key additions to the operating system and hardware to support the logging approach that we discussed. The operating system needs to provide support for creating a checkpoint at regular intervals and recording page tables on context switch and a page fault. Creating a checkpoint requires logging the context header for each processor core in its local log (context header does not contain the memory state). Also, on a context-switch it needs to log the context header for the newly scheduled process or thread. The processor on the other hand needs to provide support for logging the data of the cache block fetched on a cache miss, and the instruction count. When compared to the system-dependent logging approach, we believe that this approach is a lot simpler.

5.2 Reproducing Shared-Memory Dependencies using Offline Analysis

In Section 5.1, we shows that a load-value based logging enables deterministic replay of each thread in isolation. However, to debug and understand parallel executions, we need the causal order, compliant to the memory consistency model, between shared-memory operations as well. In this section we present algorithms that calculate such causal order offline.

5.2.1 Overview of Offline Symbolic Analysis

The goal of our offline analysis is to find a valid *causal order* between all the memory accesses executed concurrently across all the threads. The algorithm takes the memory trace of each thread and the final memory state as input. The memory trace for a thread is produced by deterministically replaying using its cache miss log (Section 5.1). For each memory access, we have its effective address, old value, new value and information about whether it was a cache hit or a miss. To produce the causal order between the memory accesses, we generate the memory ordering constraints that need to be satisfied, encode them as a quantifier free first-order logic formula and use a Satisfiability Modulo Theory(SMT) solver called Yices [26] to find a solution.

The algorithm to encode all the necessary constraints in the first-order logic formula is presented in Algorithm 1, 2, and 3. A valid causal order should satisfy two types of constraints. First, any memory access M's old value should be same as the new value of the memory access to the same location that immediately precedes M in the derived causal order. We call this constraint as the *coherence constraint*(C_H), because it is the property of coherence that ensures that there exists a total global order between all memory accesses to a location under any memory model.

Second, the causal order should obey the memory ordering constraints specified by a particular memory model. We refer to these constraints as the *memory model*(C_M) constraints. In this paper we discuss the SC and TSO constraints and algorithms to encode them as the first-order logic formula.

5.2.2 Encoding Coherence Constraints

For each memory access M, there exists an order variable O. The values of the order variables determine the causal order for the memory operations. Lines `25-36` in Algorithm 1 presents the algorithm for encoding the coherence constraints. To encode coherence constraints, for each memory access M, the algorithm specifies the set of all memory operations that access the same location and can potentially be ordered immediately after M (which requires that their old values equals the new value of M). Special care is taken to account for the possibility that an access could be the last access to a memory location.

In the example shown in Figure 5.1(right), all accesses are to the same location `X`. X_5 is the only access that can immediately follow X_4, because only X_5 has the old value that matches the new value of X_4. On the other hand, X_5 could be followed by either X_6 or X_7, which leads to the possibility that there could be multiple solutions for a given execution trace. For this example, both $X_1 \rightarrow X_2 \rightarrow X_3 \rightarrow X_4 \rightarrow X_5 \rightarrow X_6 \rightarrow X_7$ and $X_1 \rightarrow X_2 \rightarrow X_3 \rightarrow X_4 \rightarrow X_5 \rightarrow X_7 \rightarrow X_6$ are valid causal orders as they both obey the coherence constraints.

5.2.3 Encoding Memory Model Constraints for SC

Memory model constraints are determined by the underlying consistency model, which can be classified according to whether it follows or relaxes the following two axioms: *Instruction Reordering* and *Store Atomicity* [4, 7]. Sequential consistency has the strictest requirement. It requires that program order between the memory operations of a processor is satisfied in the global total order observed between memory accesses executed by all the processors. Thus, it does not allow any type of instruction reordering. Moreover, SC requires store atomicity. That is, either all or none of the processors see a store's value.

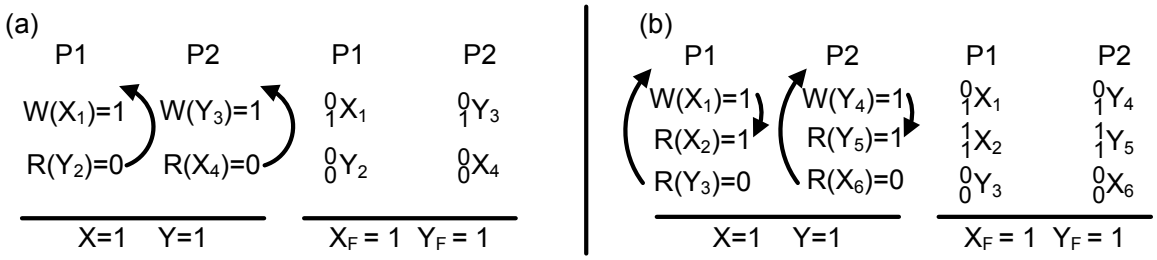

Figure 5.2: Two example TSO executions and their replayed memory traces with old/new values

Algorithm 2 describes how to encode the memory model constraints for SC. For each processor, we ensures that every memory operation should obey the strict program order, preventing any instruction reordering. Our algorithm encodes these constraints using the order variables O of memory operations. Under SC, order of each memory operation in a processor P is constrained to be greater than the order of all the earlier memory operations that appeared in the program order in P. Together with uniqueness constraints, this constraint in effect ensures store atomicity as well. For example shown in Figure 5.1(right), SC ensures $O_1 < O_2 < O_3 < O_4 < O_7$ for $P1$ and $O_5 < O_6$ for $P2$.

5.2.4 Encoding Memory Model Constraints for TSO

The TSO model is widely used in the SPARC [67] and is also similar to the x86 memory model [36, 37]. It relaxes the SC memory order constraints between memory accesses executed in a processor core as follows:

- **Instruction Reordering:** A processor may re-order a load before a store if they access different locations.
- **Store Atomicity:** A store's value may be made visible to a following local load in the same processor before it is made visible to remote processors.

Algorithm 3 describes how we encode these memory ordering constraints. We relax this constraint for TSO to account for the above two relaxations (line 6).

5.2.4.1 Allowing Relaxed Instruction Reordering

Instruction re-ordering relaxation in TSO allows a processor to retire a store to a local store buffer and allows following (performance critical) loads to execute. The execution shown on the left in Figure 5.2 is not valid under SC but is a valid TSO execution due to the relaxed instruction ordering requirement. Under SC, either Y_2 or X_4 should be the last memory operation in any valid total order. But in this example, both loads Y_2 and X_4 are executed before the stores X_1 and Y_3 respectively, which leads to a non-SC order.

If a load can be re-ordered above a store in a processor, the ordering requirement between them is not specified in the first-order logic formula. The first clause in line `40` in Algorithm 1 checks whether the ordering between a store and a load can be relaxed, and if so no ordering constraint between those two instructions will be enforced (Line `42`).

5.2.4.2 Allowing Relaxed Store Atomicity

Store atomicity relaxation allows a processor to read its own store's value early. That is, a processor can forward the value of a store in the store buffer to a following load to the same location. Thereby, if a store results in a cache miss, a processor need not wait for it to resolve, and instead forward that store's value to a later load accessing the same location.

Relaxed store atomicity requirement can be accounted for in our memory model constraints. Under TSO, a load-to-load program order constraint generally cannot be broken. However, to accommodate relaxed store atomicity constraint, we make an important observation that loads can be allowed to reorder with respect to an older load that read its local store's value. Thus, if we can determine all loads that read its local store's value offline, then we can simply relax the ordering requirement between those loads and the loads that follow them in the program order. To identify the loads that read a local store's value offline, we log the memory count of the load that hit in the store buffer during recording.

The example shown on the right in Figure 5.2 is not valid under SC as it violates store atomicity but is valid under TSO. Under TSO, while the stores X_1 and Y_4 are temporarily held in P_1's and P_2's store-buffer respectively, the loads X_2 and Y_5 can read the value `1` written by locally buffered stores. Then, before the stores' new values become visible to the other processor, loads Y_3 and X_6 can read the old value `0` from their locally cached copies. Effectively, the load-load ordering between $X_2 \rightarrow Y_3$ and $Y_5 \rightarrow X_6$ appear to be relaxed.

Thus, we accommodate TSO's relaxed store atomicity constraint by allowing loads to be re-ordered with respect to older loads that resulted in store buffer hits during offline analysis. The second clause in line `40` in Algorithm 1 checks for the condition when a load-load order can be relaxed. The offline analysis should ensure that no remote load/store accesses are interleaved between the load that caused a store buffer hit and the previous store to the same location. This is taken care of by ensuring that, if a load results in a store buffer hit, its preceding store's immediate follower set (IFS) contains only the load that resulted in the store buffer hit (Line `11`).

Consider again the example trace in Figure 5.2(b). Our analyzer would relax the program order constraints $X_2 \rightarrow Y_3$ and $Y_5 \rightarrow X_6$ because X_2 and Y_5 are store buffer hits. This would allow our analysis to produce a valid causal order under TSO for this example: $Y_3 \rightarrow X_6 \rightarrow X_1 \rightarrow X_2 \rightarrow Y_4 \rightarrow Y_5$.

While relaxing the above constraints, we also specify that all memory accesses following a

fence or a lock-prefixed memory operation should obey the program order.

5.2.5 Replay Guarantees and Finding All Solutions

As we find a valid causal order that meets the above constraints, the order determined by the offline analysis might be different from the original order observed during recording. However, the replayed execution is guaranteed to be a valid execution that leads to the same final state. For example, in the case of data races, the racy accesses would have the same value, and any erroneous behavior would also be deterministically reproduced.

Our symbolic analyzer can produce all possible solutions by adding the negation of the previously derived solutions into the constraints. The procedure continues until the constraints become unsatisfiable. Finding all feasible solutions is valuable as it reveals all thread interleavings leading to the same erroneous state. In order to improve the performance we must avoid finding the solutions that are different but semantically equivalent. For example, two causal orders that have different orders on loads to the same location are equivalent. Two solutions are *equivalent* if the following two conditions are satisfied:

- The total orders of non-silent writes (which have different old and new values) are same.
- For each read and silent writes, its preceding and following non-silent writes (if exists) are same.

The definition allows us to filter out equivalent solutions due to reads and silent writes. The result of finding all possible (but non-equivalent) solutions can be found in Section 5.5.6

5.3 Bounding Search Space

The encoding algorithm presented in Section 5.2 is impractical, as an SMT solver cannot compute a satisfiable solution with limited time and resource for unbounded number memory accesses. We present a solution to bound the search space by logging hints that allows our offline analyzer to partition a multi-processor execution into smaller bounded intervals, and analyze each interval separately.

To bound the search space we log barrier-like hints called *Strata* at regular intervals [**?**]. Each processor keeps track of the length of interval by counting the number of cycles elapsed since the last Stratum log. Once a predetermined threshold is reached, all the processors simultaneously record their current memory counts. A memory count is the number of memory operations *committed* by a processor. The program execution between two Strata hints is referred to as a *Strata region*.

Each processor logs its memory counts at the same instant of time. Under SC, memory counts logged at a particular time t provide a barrier-like happens-before relation between all

Algorithm 1: ENCODING_ALGORITHM(STRATAREGION E, FINALSTATE F)

/*
Given: Memory events $E = \{e_1, e_2, \ldots, e_{|E|}\}$ and Final State F of a Strata region
Goal : Find a causal order between memory events satisfying
(1) uniqueness constraints $\mathcal{C}_U$, (2) coherence constraints $\mathcal{C}_H$, and
(3) memory model constraints $\mathcal{C}_M$
*/
let $E|_p$ be a set of all memory accesses in processor p
let O_i be an event order variable of memory access e_i
let $e_i.loc$ be the memory location of e_i
let $e_i.type$ be the access type (load or store) of e_i
let $e_i.succ$ be the memory access to $e_i.loc$ in the same processor, following e_i in program order
let $e_i.sbh$ specifies if e_i resulted in a store buffer hit
let $e_i.IFS$ (Immediate Follower Set) be the set of memory accesses which can immediately follow e_i. It contains only $e_i.succ$ if $e_i.succ.sbh$ is a hit. Otherwise, it contains $e_i.succ$ and remote memory accesses to $e_i.loc$, provided their old values are same as e_i's new value
let $e_i.IntS$ (Interference Set) be the set of memory accesses including $e_i.succ$ and remote accesses to $e_i.loc$
let $e_i.LAST$ be the set containing a memory access if it is the last access to $e_i.loc$ in a thread and its new value is same as final state of $e_i.loc$ in F
/* The SMT solver finds a solution that satisfies all the constraints */
$\mathcal{C}_{FINAL} = \mathcal{C}_U \wedge \mathcal{C}_H \wedge \mathcal{C}_M$
/* Uniqueness Constraints */
$\mathcal{C}_U$=true;
for all pairs of memory accesses $(e_i, e_j) \in E \times E$ where $i \neq j$ **do**
    $\mathcal{C}_U = \mathcal{C}_U \wedge (O_i \neq O_j)$;
end for
/* Coherence Constraints */
$\mathcal{C}_H = true$;
for all memory accesses $e_i \in E$ **do**
    for all $e_j \in e_i.IFS$ **do**
        //order one immediate follower e_j, prevent other accesses to $e_i.loc$ from being scheduled between e_i and e_j
        $\mathcal{C}_i = \mathcal{C}_i \vee ((O_i < O_j) \wedge \bigwedge_{e_k \in (e_i.IntS - \{e_j\})}((O_k < O_i) \vee (O_j < O_k)))$;
    end for
    if $e_i \in e_i.LAST$ **then**
        //schedule e_i to be the last memory access to $e_i.loc$ and prevent other accesses in $e_i.LAST$ from being the last access
        $\mathcal{C}_i = \mathcal{C}_i \vee (\bigwedge_{e_k \in (e_i.LAST - \{e_i\})}(O_k < O_i)$;
    end if
    $\mathcal{C}_H = \mathcal{C}_H \wedge \mathcal{C}_i$;
end for
/* Memory Model Constraints are shown in Algorithm 2 and 3*/

Algorithm 2: ENCODING_ALGORITHM FOR SC

1: **/* Memory Model Constraints for SC */**
2: $\mathcal{C}_M$=true;
3: **for all** $E|_p : \langle e_{p_1}, e_{p_2}, \ldots, e_{p_k} \rangle \subseteq E$ **do**
4: $\quad \mathcal{C}_M = \mathcal{C}_M \wedge (O_{p_1} < O_{p_2} < ... < O_{p_k})$;
5: **end for**

Algorithm 3: ENCODING_ALGORITHM FOR TSO

1: **/* Memory Model Constraints for TSO */**
2: $\mathcal{C}_M$=true;
3: **for all** $E|_p : \langle e_{p_1}, e_{p_2}, \ldots, e_{p_k} \rangle \subseteq E$ **do**
4: $\quad$ **for** $i = p_k; i > p_1; i--$ **do**
5: $\quad\quad$ **for** $j = i-1; j \geq p_1; j--$ **do**
6: $\quad\quad\quad$ **if** $\neg(e_i.type = load \wedge e_j.type = store \wedge e_i.loc \neq e_j.loc) \wedge$
//store-to-load reordering
$\neg(e_i.type = load \wedge e_j.type = load \wedge e_i.loc \neq e_j.loc \wedge e_j.sbh = hit)$
//store atomicity violation
then
7: $\quad\quad\quad\quad \mathcal{C}_M = \mathcal{C}_M \wedge (O_j < O_i)$;
8: $\quad\quad\quad$ **end if**
9: $\quad\quad$ **end for**
10: $\quad$ **end for**
11: **end for**

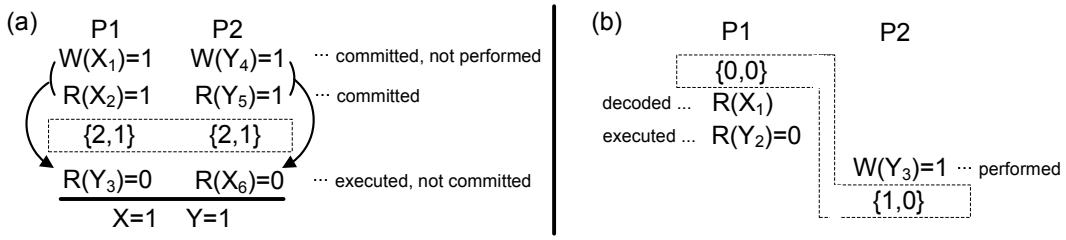

Figure 5.3: Two examples of recording Strata Hints under TSO

memory accesses committed before t and the accesses committed after t. Thus, memory accesses in different Strata regions are totally ordered. An SMT solver can solve one Strata region at a time, starting with the last Strata region and final state. Later, solutions found for all the regions are concatenated based on the total order for Strata.

The above approach, however, is not sufficient for recording execution under relaxed consistency models and out-of-order executions. We discuss how we can ensure the correctness of the happens-before relations specified by the count of committed memory operations.

5.3.1 Pending Stores in Store Buffer

For clarity, we distinguish between three states of a memory access' execution: (a) a load access is said to have *executed* if it has read the value, (b) a memory access is said to have *committed* when it is committed in-order and its entry removed from the Re-Order Buffer (ROB) and (c) a store access is said to have *performed* when its value is written to the cache block (made visible to remote processors) and its entry removed from the store buffer.

Stores committed from the ROB, but not yet removed from the store buffer could violate the happens-before specified by the Strata hints. Consider the example in Figure 5.3(a), which shows the same trace in Figure 5.2(b). Ignore the dashed box for now and let us assume the following state: stores X_1 and Y_4 are committed but not yet performed (temporarily buffered in the store-buffer of their respective processors), loads X_2 and Y_5 are committed, and loads Y_3 and X_6 have only been executed but have not been committed yet. Assume a Stratum is logged at this state. Each processor logs that they have committed two memory operations. This Stratum would provide happens-before relation $Y_4 \rightarrow Y_3$, but in reality load Y_3 executed before the store Y_4 was made visible to $P1$. Similarly, an incorrect happens-before order $X_1 \rightarrow X_6$ would be enforced by the offline analyzer. As a result, it will be impossible to find a satisfiable solution for the second Strata region containing $\{Y_3$,$X_6\}$ as they would conflicts with the final state.

We solve this problem by logging the number of in-flight stores ($IStore$) in addition to the number of committed instructions as Strata hints. Since the stores are retired in-order from the store buffer, the offline analyzer can determine that the last $IStore$ stores in a thread before the Stratum log were pending in the store buffer. Using this information, while constructing the

Strata regions, the offline analyzer moves the stores pending in the store buffer and its dependent loads (loads that read their value from the store buffer) to the following Strata region. Then they are analyzed with the memory accesses in that Strata region.

For the example in Figure 5.3(a), each processor logs both the number of committed instructions, which is two, and the number of in-fight stores, which is one. The Strata is represented as a dashed box, and the tuple inside the box shows the logged information in each processor. During offline analysis, while creating Strata regions, in-fight stores X_1 and its dependent load X_2 would be moved to the second region as the arrows indicates. Also, Y_3 and Y_4 would be moved to the second region. With this modification, now the SMT solver would be able to correctly analyze $\{X_1,X_2,Y_3,Y_4,Y_5,X_6\}$ together and arrive at valid TSO-compliant causal order.

5.3.2 In-flight Loads in Out-of-Order Execution

Most modern processor implementations have speculation support for breaking load-to-load memory ordering constraints to efficiently support TSO [31]. They execute a load out-of-order, and then re-execute them on commit to check if the out-of-order speculative execution was valid or not. The check would fail if there was a remote store that modified the value before the load commits. When a check fails, the load and its dependent operations are re-executed. However, recording Strata using committed memory counts is still sufficient even in the presence of out-of-order speculation.

Figure 5.3(b) presents an example. Say, the load Y_2 executes out-of-order returning a value of 0, but remains uncommitted. Then, the store Y_3 in P_2 executes, commits and retires from the store buffer by writing a value of 1 to memory. If Strata is created at this moment, then the loads X_1 and Y_2 in P_1 would be considered as part of the second Strata region, because those loads have not committed yet, whereas the store would be considered as part of the first Strata region. This would be an incorrect happens-before relation. However, before committing Y_2, the processor would re-execute the load and find that its value has changed, which would trigger a misspeculation recovery.

5.3.3 Bounding Search Space Effectively Using B-bound

Processors can determine the end of a Strata region in many ways. The simplest approach would be for each processor to count the number of processor cycles and determine the end of an interval when a threshold number of cycles had elapsed. This bounding mechanism is called **c**ycle bound (c-bound), and it requires no additional communication between processors. However, the interval size does not account for the degree of communication between concurrently executing threads which is a critical factor that determines the offline analysis time.

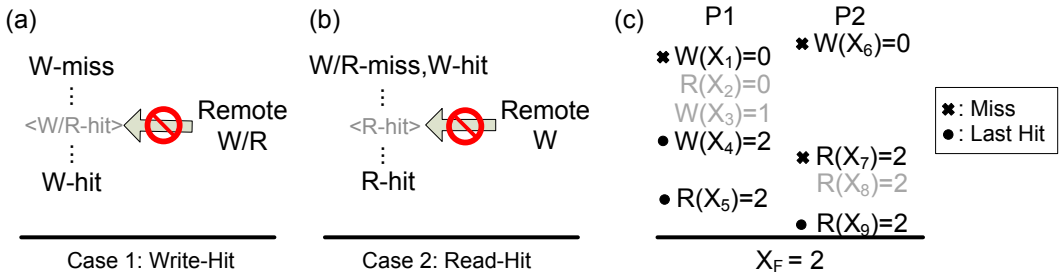

Figure 5.4: (a) Write-hit Property, (b) Read-hit Property, and (c) Cache Hit Filtering Example

An adaptive scheme that logs adjusts Strata region size based on the amount of inter-processor communication is preferable.

We evaluate two approaches that are aware of the degree of communication between processors. One is called **d**owngrade bound (d-bound). In d-bound, each processor counts the number of invalidated or downgraded cache blocks in an interval. If any processor observes downgrades more than a pre-configured threshold, it asks all the other processors to log a Stratum hint. Since the number of downgrades implicitly capture the amount of sharing, we expect the number of shared accesses to be analyzed across Strata regions to be similar. However, d-bound requires changes to the coherence mechanism as it requires additional inter-processor communication to create Strata.

We also evaluate a second approach that is suitable for a snoop-based architecture **b**roadcast-bound (**b**-bound). In snoop-based architecture, each processor snoops the coherence messages broadcasted on the bus. We leverage this property to determine the Strata interval length. Each processor simply counts the number of broadcasted messages and when a threshold is reached a Stratum is logged. Thus, b-bound does not require additional communication between processors in a snoop-based architecture, while it can also adapt the frequency of Strata logs according to the degree of communication between concurrent threads.

5.4 Reducing Offline Analysis Cost Using Cache Hit Filtering

While analyzing a Strata region, causal order for many memory operations can be trivially determined and therefore filtered out from the time consuming symbolic analysis. First, *local access filtering* eliminates all accesses to a location that are accessed in only one processor in a Strata region. Second, *read-only access filtering* removes all accesses to a location that are only read within a Strata region. This is because any causal order between these eliminated accesses is valid within a Strata region. Since filtering unnecessary memory operations can significantly reduce offline analysis time, we propose an additional non-trivial filtering method called *cache*

hit filtering (CHF).

Cache hit filtering is based on our observation that memory operations between a cache-miss and its last-cache-hit (a hit before losing the read or write permission) to a location can be filtered from the offline analysis. Our recorder logs only cache blocks fetched on a cache miss, and so after replaying each thread and obtaining its memory trace, the offline analyzer can determine which memory accesses had resulted in cache misses during recording. Using this cache hit and miss information, it is also trivial to determine last-cache-hits.

Our offline analysis filters out all cache hits except last-cache-hits. Following constraints (also illustrated in Figure 5.4) are added to the first-order-logic formula produced by Algorithm 1.

- Remote reads and writes cannot be interleaved between write-miss and a consecutive write-hit.
- Remote writes cannot be interleaved between {write-miss, read-miss, or write-hit} and a consecutive read-hit.

The SMT solver then finds a valid causal order among only unfiltered memory operations. The order for the filtered operations are inferred trivially according to the program order.

Figure 5.4(c) shows an example for the cache hit filtering optimization. Memory operations marked with crosses are cache misses. Memory operations marked with solid dots are last read-/write cache hits. Rest of the memory operations in gray are the memory operations eliminated by the cache hit filtering optimization. Following are the constraints added to correctly support this optimization. Write X_6 and reads X_7, X_9 cannot be interleaved between X_1 (write-miss) and X_4 (write-hit). Remote write X_6 cannot be interleaved between X_4 (write-hit) and X_5 (read-hit). Note that remote reads such as X_7 and X_9 are allowed to be scheduled between X_4 and X_5.

5.4.1 Implications of Cache Hit Filtering

In addition to enforcing the above additional constraints, cache hit filtering also requires several modifications to the offline analysis. First, cache miss/hit information is available only at the cache block granularity. Therefore, local and read-only accesses also need to be determined at the block granularity instead of at the word granularity. Otherwise, local and read-only filtering may incorrectly remove memory operations that are cache-misses or last-cache-hits which need to be preserved for enforcing cache hit filtering constraints. This may reduce the effectiveness of local and read-only filtering optimizations due to false sharing at the block granularity. However, we expect that cache hit filtering would effectively compensate for the cost.

Second, memory dependencies should also be determined at the block granularity. That

is, SMT solver should consider block address to determine whether two memory accesses are aliased. Similarly, old and new value comparison should also be performed at the block granularity. This reduces the aliasing between values of loads and stores, and thereby reduces the search space and offline analysis time.

Third, filtering out write hits may lead to a mismatch between old and new values of unfiltered accesses. We resolve this by patching up the old and new values at the cache block granularity before we feed the filtered traces to the SMT solver.

Finally, a Stratum log may be created between a cache miss and a cache hit. This implies that the first access to a location in a Strata region is a cache hit. We add additional constraints to take care of such tricky special boundary cases. For instance, if the first access to a location in a thread in a write-hit, then none of the remote accesses to that location is allowed to be scheduled before the write-hit.

5.5 Results

In this section, we begin with Section 5.5.1 by describing our evaluation methodology and then provide experimental results as follows. Section 5.5.2 evaluates the size of Strata hints log and offline analysis overhead for the TSO model and compare them to the sequentially consistent (SC) model. The following three sections provide detailed analysis on different bound schemes (c-bound, d-bound, and b-bound) and filtering optimizations. Then, Section 5.5.6 shows the number of all possible solutions. Section 5.5.7 and 5.5.8 evaluate the program input and store buffer hit log size respectively. Finally, we present recording performance in Section 5.5.9.

5.5.1 Evaluation Methodology

Our simulation framework is based on Simics [46] for full system functional simulation and modified FeS2 [1] for cycle-accurate TSO simulation. We model 2, 4, 8, and 16 cores, each with a 32 KB private L1 cache (32-byte block, 4-way associative, 3-cycle latency) and a shared L2 cache (64-byte block, 8-way associative, 30-cycle latency). We model the MESI coherence protocol and a store buffer (32 entry FIFO, 8-byte granularity). We also model speculation support for breaking load-to-load memory ordering constraints to efficiently support TSO [31].

We use four sets of benchmarks: SPLASH-2 [80], PARSEC 2.0 [14], SPEComp [68], and server applications. We evaluate our system with `barnes`, `fmm`, and `ocean` from SPLASH-2, `blacksholes`, `bodytrack`, and `x264` from PARSEC 2.0, `wupwise`, and `swim` from SPEComp, and two server applications `Apache` and `MySQL`. All applications are configured to have the same number of worker threads as the number of cores. We fast-forward up to a point where all the threads are spawned and the program starts its main computation (e.g. up

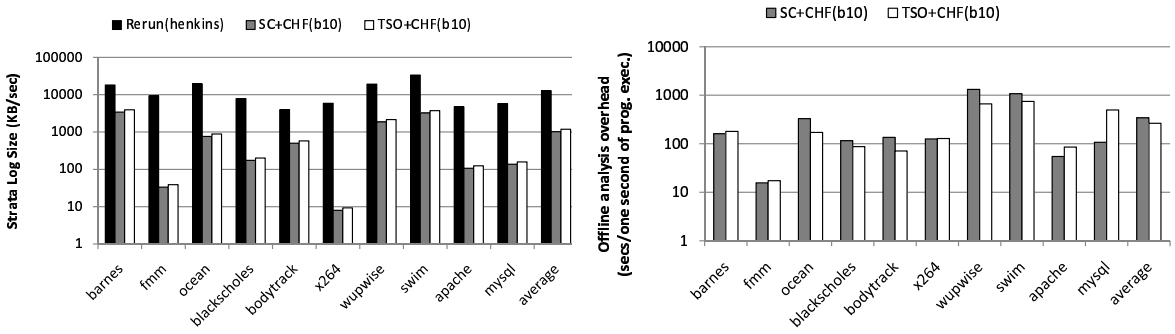

Figure 5.5: Strata log size and offline analysis overhead under SC and TSO memory models

to the second barrier synchronization point or OMP parallelization point). Then, we collect the multi-threaded workload traces for 500 million instructions. For Apache, we use SURGE [10] to generate web requests to a repository of 20000 files (totaling 480 MB) with 400 concurrent clients. For MySQL, we use SysBench [3] to send concurrent queries to a database containing one million records. We tested OLTP mode with 16 client threads. Except the scalability results, all other results are collected for 8-core configurations. Finally, for offline symbolic analysis, we used the Yices SMT solver [26].

5.5.2 Strata Log Size and Offline Analysis Time

Figure 5.5 compares Strata log size and offline analysis time between the SC and TSO models. For this experiment, we apply cache hit filtering and b-bound optimization with a threshold of 10. Sensitivity results on these optimizations are presented in the later three sections.

We logged 4 bytes to record a memory count in a processor core while logging a Stratum, but this could be optimized by recording only the different in memory counts in a processor between two Strata logs. In addition, we logged the number of in-flight stores to support TSO model (Section 5.3). For a 32 entry store buffer, we need 5 bits to log the number of in-flight stores. On average, we need 1025KB for SC and 1185KB for TSO (15% increase) to record Strata hints for one second of program execution on an 8-core configuration.

Figure 5.5(b) shows that offline analysis time for TSO surprisingly decreases by 30% when compared to that of SC. Relaxing constraints could have positive or negative effect on offline analysis time. Under TSO, search space increases. But when compared to SC, in TSO, it is possible that the proportion of legal solutions to the infeasible solutions that the SMT might explore increases. If so, then the offline analysis time could be better than SC. The variation in analysis time for different applications in Figure 5.5(b) is a consequence of this.

On average, it takes 260 seconds to analyze one second of an 8-threaded execution under TSO. `swim` is our worst case which takes 745 seconds. This offline analysis need to be per-

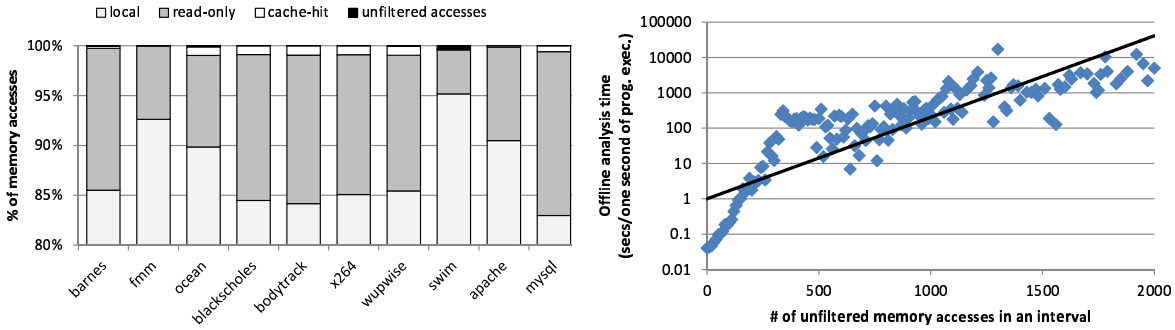

Figure 5.6: (a) Proportion of local, read-only, cache-hit and unfiltered accesses and (b) Scalability of offline analysis.

formed only once. Once shared memory dependencies are resolved, execution can be replayed with little overhead. Furthermore, we could reduce the analysis cost by parallelizing the offline analysis of different Strata regions and also improve our generic Yices solver by customizing it specifically for our problem.

For the SC model, we also compared the sizes of Strata logs to the precise race logs in one of the state-of-the-art hardware recorders, called Rerun [34]. The results show that we can save about 10 times memory race log size when compared to ReRun. However, our program input log could be larger than that of a copy-on-write based program input recorder assumed by ReRun. The program input log size is evaluated in Section 5.5.7.

5.5.3 Strata Region Length

This section provides comprehensive analysis on determining the appropriate length for the Strata regions to bound the search space of offline analysis. As discussed in Section 5.4, we eliminate local, read-only, and intermediate cache-hit memory accesses in each Strata region, and only analyze remaining *unfiltered* memory accesses that is left after filtering. Filtering the local accesses and read-only accesses within a Strata region eliminates over 99% of memory accesses from offline analysis. Figure 5.6(a) shows this result for a configuration where the Strata regions are constructed when 10 broadcasted coherence messages have been observed (b-bound of 10). Here, `swim` show the most portion of unfiltered accesses, which was 0.34%.

More memory events per Strata region would increase the cost of offline analysis. Therefore, we would like the unfiltered accesses per Strata region to be less than some threshold. Figure 5.6(b) shows the time taken (y-axis uses a log scale) to analyze Strata regions with different numbers of unfiltered accesses. This includes the execution of all test applications where the Strata regions are constructed with rough cycle bound. The result shows the exponential increase of offline analysis cost.

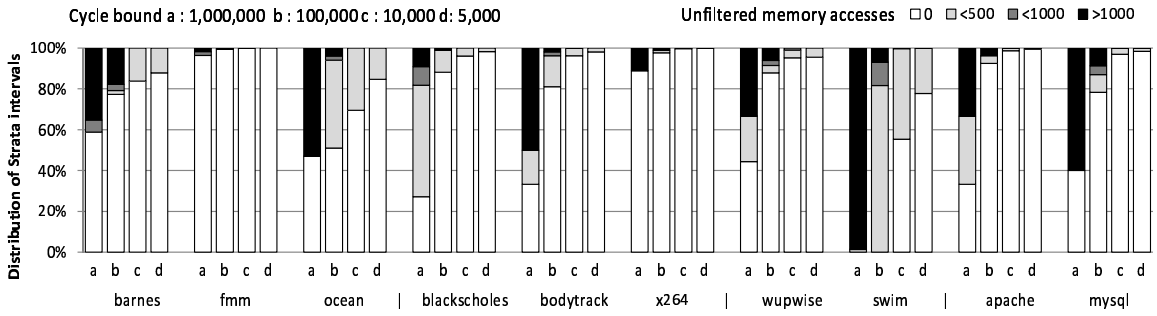

Figure 5.7: Distribution of unfiltered memory events in a Strata interval for cycle bounds

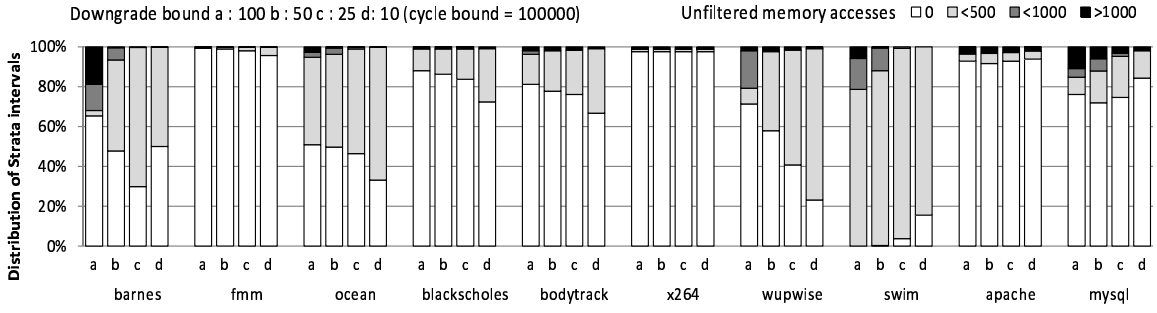

Figure 5.8: Distribution of unfiltered memory events in a Strata interval for downgrade bounds

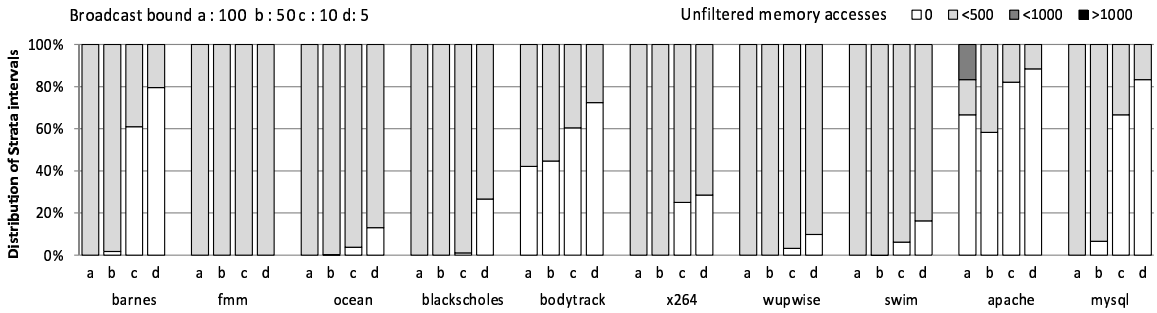

Figure 5.9: Distribution of unfiltered memory events in a Strata interval for broadcast bounds

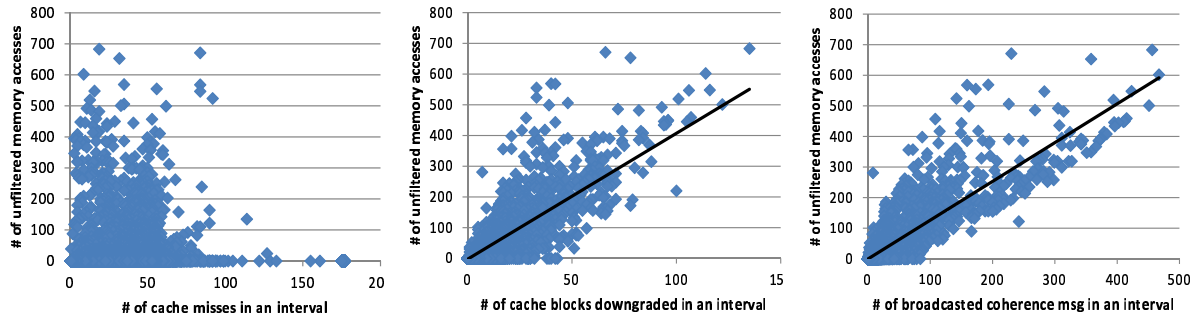

Figure 5.10: Correlation between the number of unfiltered accesses and (a) cache miss counts, (b) downgrade counts, and (c) broadcast counts

Based on this observation, we experimented with four different processor cycle bounds. Figure 5.7 shows the distribution of Strata intervals for each bound. Each Strata interval would have different number of unfiltered memory accesses depending on the program characteristics, and the intervals are classified over four ranges. Figure 5.7 shows that for programs like `fmm` even if we use a bound of a million cycles (a stratum is created after a million processor cycles has elapsed), most intervals would still be left with less than 500 unfiltered accesses. But for programs like `swim` we need a lower cycle bound, because we find many intervals with more than 1000 unfiltered accesses if we use a higher bound. Based on the application to be recorded, the operating system can set the cycle bound appropriately.

We also performed similar experiments on different downgrade bounds (Figure 5.8) and broadcast bounds (Figure 5.9). For downgrade bounds, each core counts the number of downgrade requests (invalidation or downgrade exclusive permission), and if any core reaches a predefined bound, then it sends a message to all the nodes to log a stratum. For a snoop-based architecture, we can exclude this additional communication by leveraging the property that each processor snoops the coherence messages broadcasted on the bus. Each processor simply counts the number of broadcasted messages and when a threshold is reached a Stratum is logged. These approaches are more complex than using a cycle-bound approach, but could reduce the offline analysis overhead. This tradeoff is discussed later in Section 5.5.4.

Our system does not use the number of cache misses as another metric to form Strata regions as the capcity misses are not directly correlated to the degree of communication between concurrent threads. Figure 5.10 plot the number of unfiltered accesses versus the cache miss count, downgrade count and broadcast count respectively for all applications with the cycle bound of 10000. As one can see, there exists significant correlation between the downgrade/broadcast counts and unfiltered accesses, because they are a better indicator of the sharing behavior in an application (more sharing would result in more memory accesses in an interval as we filter out local and read-only accesses).

5.5.4 Effects of Cache Hit Filtering and B-Bound Optimization

In this section, we evaluate the effectiveness of cache hit filtering (CHF) and also compare different bounding schems. We compare four configurations under SC: SC(c-bound), SC(d-bound), SC+CHF(d-bound), and SC+CHF(b-bound). The first SC(c10000) configuration represents cycle-bound scheme where each processor forms Strata after 10000 **c**ycles have elapsed (c-bound). The second SC(d10.c10000) configuration is for downgrade-bound approach where each processor creates Strata either after more than ten cache blocks have been **d**owngraded (d-bound) or after 10000 **c**ycles have elapsed. The third configuration, SC+CHF (d10.c10000), employs our cache hit filtering optimization over the earlier design. The forth configuration,

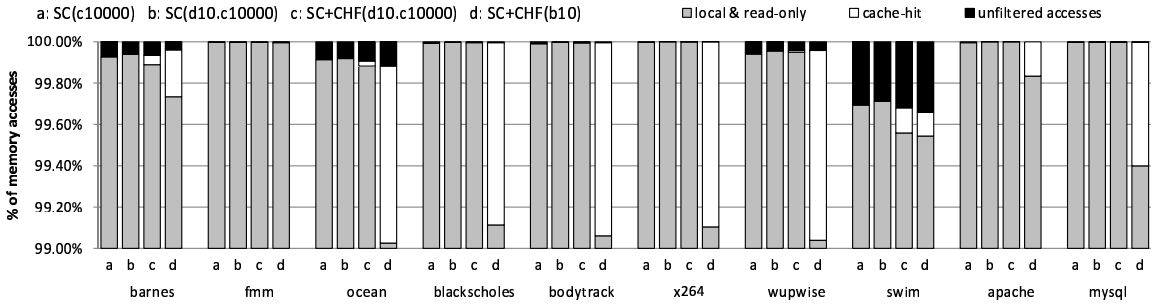

Figure 5.11: Effectiveness of local, read-only, and cache-hit filtering

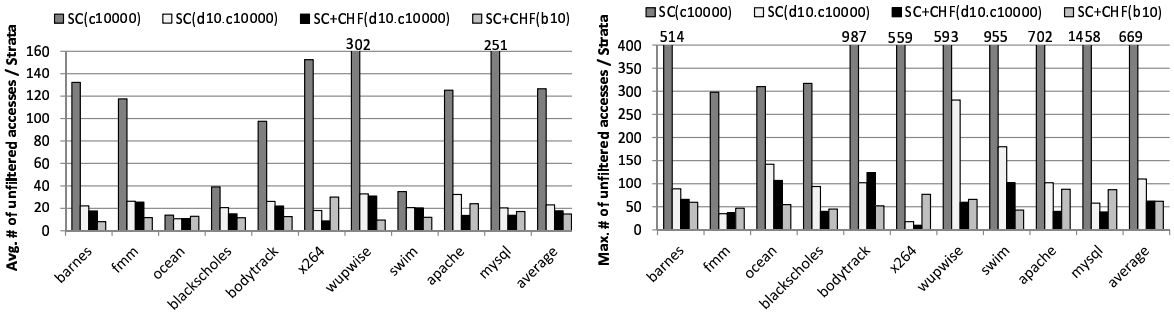

Figure 5.12: Average and maximum number of unfiltered accesses per Strata region

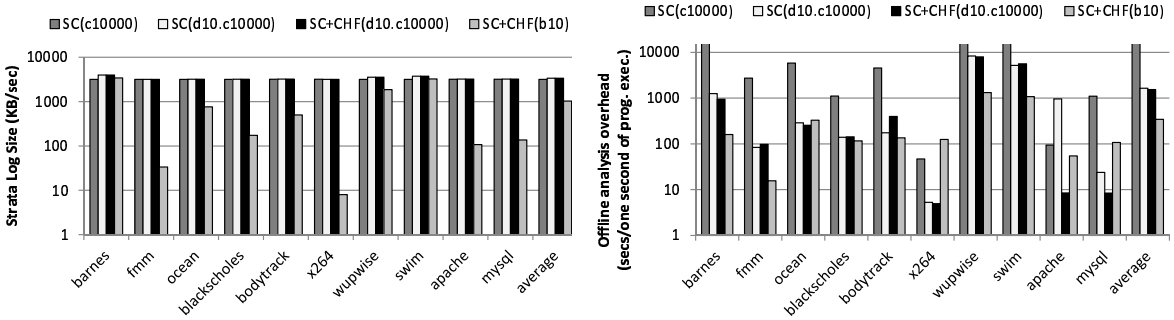

Figure 5.13: Effectiveness of b-bound and Cache Hit Filtering (CHF) in reducing Strata log size and offline analysis overhead

SC+CHF(b10), is a design with cache hit filtering and with b-bound. Each core creates Strata if there have been more than ten coherence **b**roadcast messages (b-bound). Without cache hit filtering, b-bound optimization alone is not effective because there could be a Strata region with high hit rate and less sharing, leading to a large number of unfiltered accesses.

Figure 5.11 shows differences in local, read-only, and cache-hit filtering ratio among different bounding schems. For the first two configurations without cache-hit filtering, we need to analyze less than 0.4% of total memory operations. Applying cache hit filtering (SC+CHF (d10.c10000)) reduces the effectiveness of local or read-only filters due to false sharing (we have to analyze at the block granularity if we employ cache hit filtering). However, cache-hit filtering effectively compensates for the loss. Figure 5.12 shows the average and maximum number of unfiltered accesses per Strata region. This shows that the cache-hit filtering is effective especially when we combine it with the b-bound optimization represented as SC+CHF(b10). When the two optimizations are applied together the average number of memory operations per Strata reduces by more than 10x compared to c-bound (SC(c10000)) and nearly 40% compared to the d-bound (SC(d10.c10000)). Programs like `ocean` with a high cache miss rate does not benefit from CHF optimization.

Also, the maximum number of operations per Strata region, which determines the worst case analysis time, reduces significantly in most cases. Maximum number of memory operations per Strata region is an important measure given that the offline analysis time grows exponentially with the number of memory operations to be analyzed together. The maximum number of memory operations across all Strata regions in all programs is less than 90 after employing CHF and b-bound optimizations.

One interesting property of b-bound is that the standard deviation of the maximum number of unfiltered accesses across different benchmarks is a lot smaller than d-bound (16.9 vs 76.6). This indicates that b-bound with CHF is a better application-independent predictor of how large each Strata region should be. Furthermore, b-bound is simpler than d-bound in terms of hardware implementation for a snoop-based architecture, because d-bound requires additional communication among cores but b-bound does not.

Figure 5.13 shows the result of cache hit filtering and b-bound optimizations on Strata log size and offline analysis overhead. The b-bound optimization reduces the Strata log size by nearly three times. Cache hit filtering does not affect the size of Strata log, because it is an offline filter employed to reduce the number of memory operations that need to be analyzed within a Strata region.

Cache hit filtering reduces offline analysis time by reducing the number of memory operations that need to be analyzed. Also, to support CHF optimization, we perform analysis at the cache block granularity. While this may reduce the effectiveness of local and read-only filters as discussed before, it could reduce the search space by reducing the amount of aliasing be-

tween the old and new values of memory operations which in turn reduces the legal follower set for a memory operation. On average, compared to SC(d10.c10000), SC+CHF(d10.c10000) shows 6% of improvement on offline analysis overhead. However, together with b-bound, SC+CHF(b10) shows impressive improvements: 3x less Strata log size and 4.8x less offline analysis time on average.

5.5.5 Sensitivity Studies

In this section, we present sensitivity studies varing the threshold on each bounding scheme. Figure 5.14, 5.15, and 5.15 shows the tradeoff between Strata log size versus offline analysis time for c-bound, d-bound and b-bound respectively. For example, Figure 5.16 shows that on average, Strata log size increase approximately linearly as the b-bound decreases from 100 to 5: 122KB/sec, 242KB/sec, 1185KB/sec, and 2333KB/sec, respectively. On the other hand, it takes 54261 seconds with b-bound of 50 to analyze one second of 8-threaded execution, whereas it only takes 258 seconds with b-bound of 10, which is about 210 times of improvement. The user or the operating system can specify the bound based on the trade-off that one is willing to pay.

We also present scalability results with different number of processors for a constant b-bound of 10. Figure 5.17(a) shows that on average Strata log size increases by 3.1x, 1.6x, and 2.4x as the number of cores doubles from 2 to 16. Similarly, Figure 5.17(b) shows that offline analysis cost increases by 1.8x, 3.3x, and 2.5x respectively.

5.5.6 The Number of Satisfiable Solutions

In this section, we evaluate the number of feasible solutions. Our offline analysis can find all satisfiable solutions that obey all the required constraints compilant to the given memory consistency model. We also applied the notion of *equivalent* solutions (discussed in Section 5.2.5) to prevent the solver from finding a solution that has indeed the same causal dependencies as the previous solutions.

Figure 5.18(a) shows the statistics on the number of possible solutions for each application. On average 62% of Strata regions have a single solution; 29% have less than 10 solutions; 8% have less than 100 solutions. The remaining 1% of Strata regions have more than 100 solutions. In further investigation, we find that four applications (`barnes`, `wupwise`, `swim`, and `ocean`) have Strata regions with more than 100 solutions, but all of those Strata regions actually have relatively small number of memory operations accessing a single or few different memory locations. For example, `barnes` has a Strata region that includes 300 different solutions with only 12 unfiltered memory operations accessing a single location; and `wupwise` includes a Strata region that produces 216 satisfiable solutions with 12 memory operations accessing a single memory location.

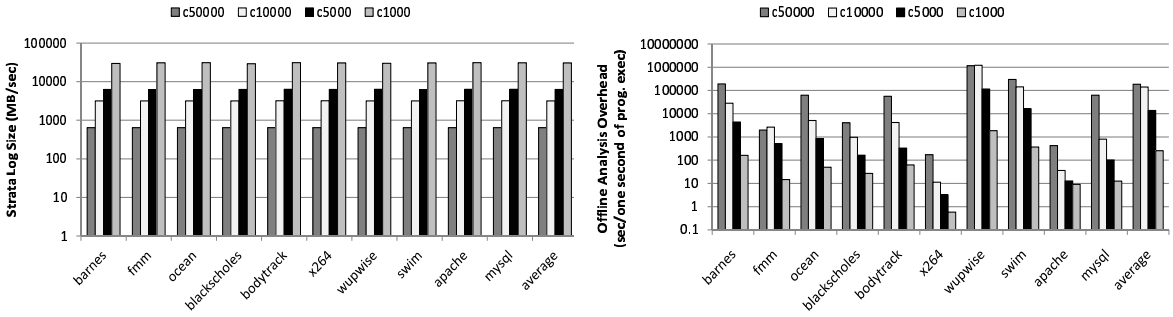

Figure 5.14: Strata log size and offline analysis overhead for different c-bounds

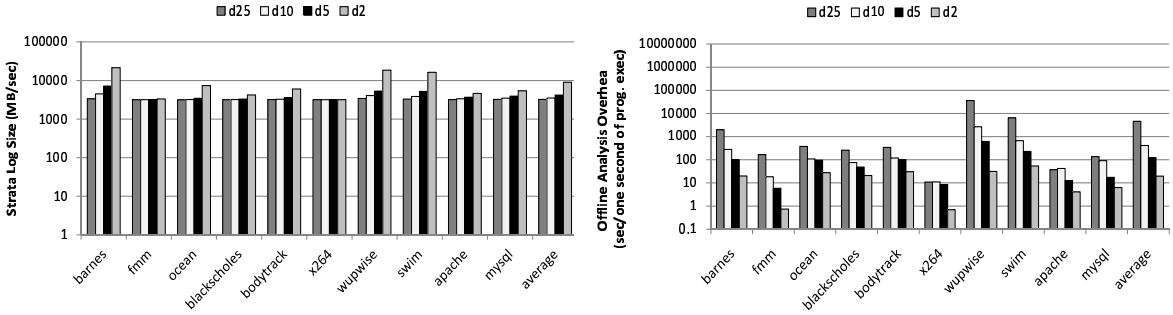

Figure 5.15: Strata log size and offline analysis overhead for different d-bounds

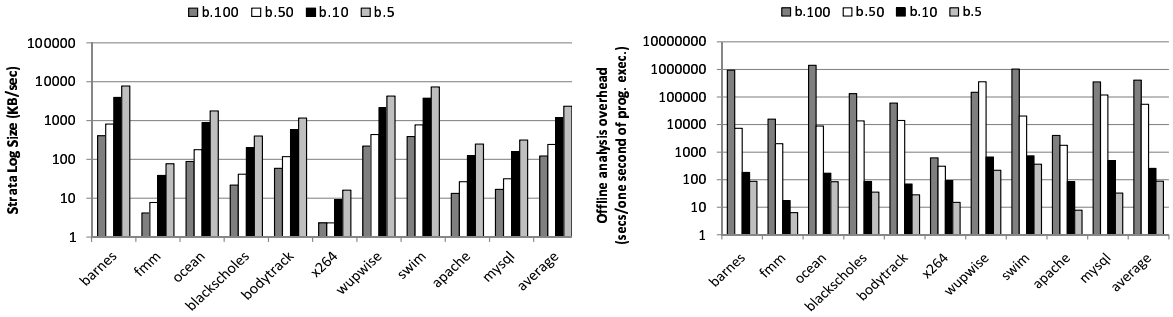

Figure 5.16: Strata log size and offline analysis overhead for different b-bounds

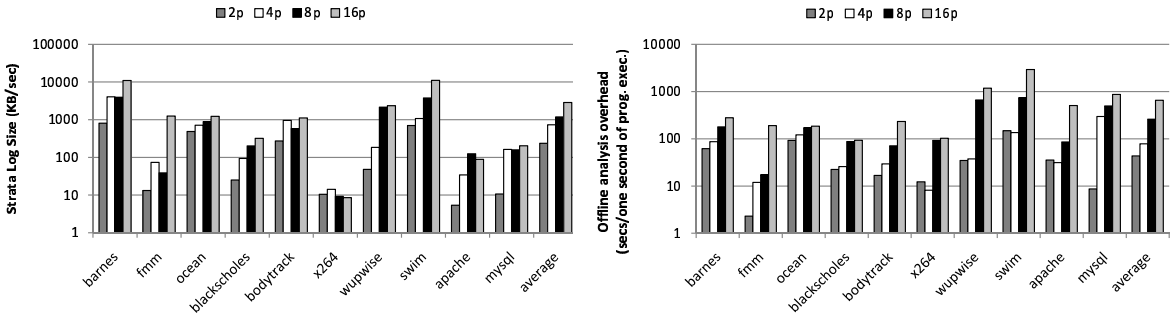

Figure 5.17: Strata log size and offline analysis overhead for different number of processors

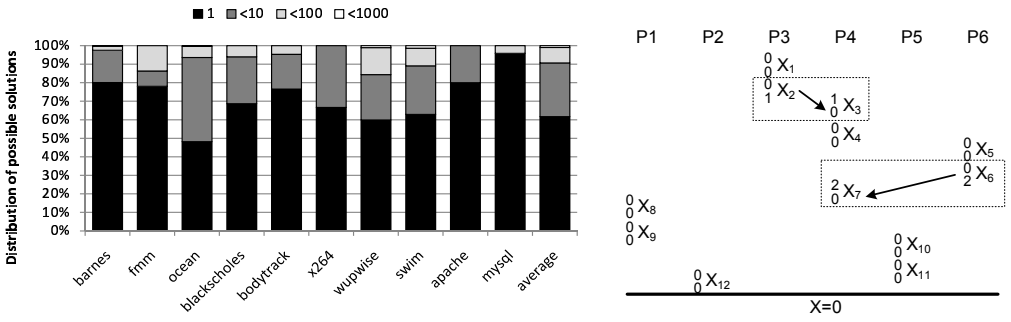

Figure 5.18: The number of satisfiable solutions. (a) Distribution of possible solutions (b) Example of many solutions in `wupwise` due to silent stores

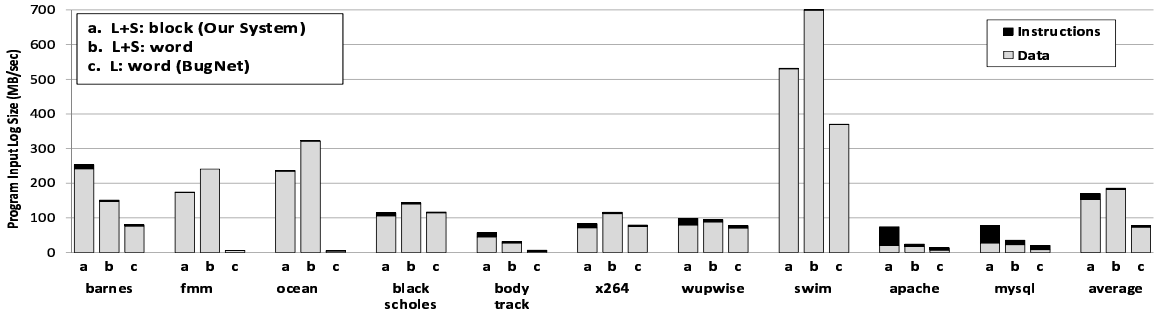

Figure 5.19: Program input log size with 16bit counters

It turns out that these many solutions are mainly due to a set of writes that together act as silent writes, which have the same old and new value. Figure 5.18(b) is an example of a Strata region that has many (216) possible solutions in `wupwise`. In this example, X_2 and X_3 (similarly X_6 and X_7) form silent write pairs that together has the same old and new value of 0. As long as they are ordered ($X_2 < X_3$, $X_6 < X_7$), all the rest operations can be ordered in any place, leading to many solutions. For instance in P1, X_8 and X_9 can be placed before $X_2 < X_3$ pair, or between X_3 and X_6, or after $X_6 < X_7$ pair, leading to 6 possible combinations. Similarly X_12 in P2 have 3 choices, X_10 and X_11 in P5 have 6 choices, and X_5 in P6 has 2 choices. As a result, total valid solutions become 6*3*6*2 = 216.

5.5.7 Program Input (Cache Miss) Log Size

As discussed in Section 5.1, our system logs cache miss data to implicitly capture non-deterministic program input such as I/O, interrupt, DNA, etc. and replay each thread's memory operations in isolation. On any cache miss, the cache block fetched is directly written back to the main-memory along with the current instruction count of the processor core. Thus, the total size of program input log can be calculated by (Size of Cache Block + Size of dyanmic counter) * (Number of cache misses).

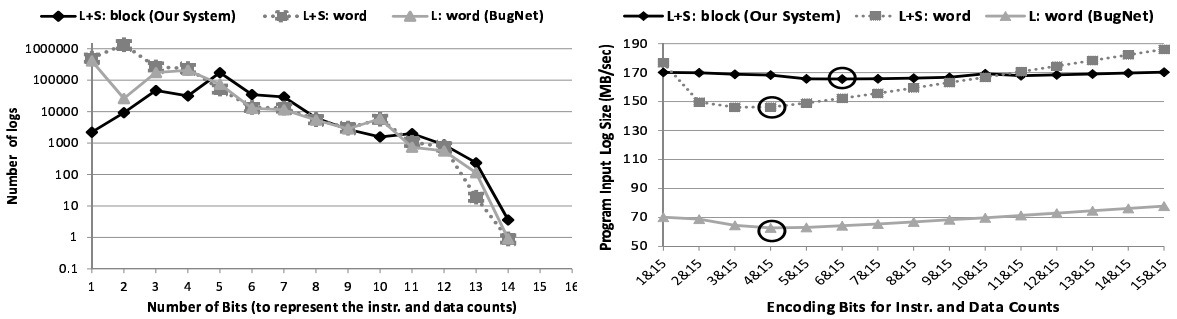

Figure 5.20: Distribution of instruction counts and program input log size with compressed format

Figure 5.19 shows program input log size for different logging schemes. All the logging scheme assume 16bit(2bytes) of counter. Logging scheme (a) is the baseline system which was described in Section 5.1. It records both load and store misses at the block granularity. Logging scheme (b) shows the log size for logging at word-size granularity, assuming that we track depedencies with additonal log-bit per word in cache like BugNet [51]. We expected that fine-grained logging should outperform cache-block-size logging due to less false sharing. However, the result shows that word-granularity logging (b) is not better than block-granularity logging (a). It turns out that we have to record more dynamic instruction operation counts between two log records for (b). That is, even though (b) captures shared memory dependency at fine-grained level, each record should include 16bit counters, which offsets the benefits.

To further understand the situations, we collected the distribution of dynamic instrunction counts between two log records. In Figure 5.20(a), the y-axis shows the number of logs, and x-axis shows the number of bits that is required to represent the log strides. As can be seen, most counts in the program input log for (b) is biased on lower bits. This implies that we can represent the majority of instruction counts with small number of bits, for example with 4 bits. Based on this observation, we can reduce the size of each log record by using two forms of dynamic counts. Let's say we use the first prefix bit to distinguish whether the log record includes compressed count format (4bits) or full count format (15 bits) for large counts. Figure 5.20(b) shows the result of sensitivity test on different encoding formats which use 1 to 15 bits for compressed format. The figure shows that (a) can be optimized with 6+15 format and shows 165MB/sec, whereas (b) is optimized with 4+15 format and shows 146MB/sec. Note that tracking dependencies at word granularity (b) requires additional hardware support like log-bit per cache word. Therefore, there is a tradeoff between log size and hardware complexity. We also add the input log size for BugNet in (c) which only logs the first load to each memory location and assumes additional hardware support for multiprocessor replay.

App.	Store Buffer Hit(%)
barnes	1.22
fmm	4.89
ocean	6.65
black.	1.95
body.	0.63
x264	2.33
wupwise	0.92
swim	0.35
apache	3.23
mysql	2.28
average	2.45

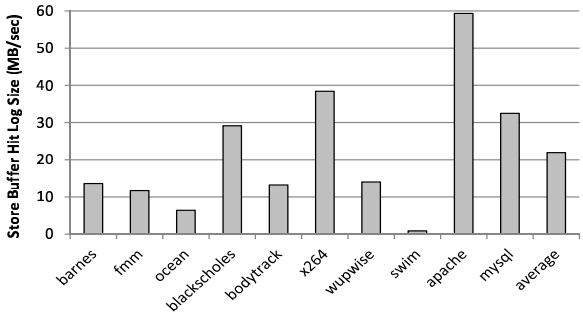

Figure 5.21: (a) Store buffer hit ratio and (b) Store buffer hit log size

application	baseline average IPC	slowdown without priority	slowdown with priority
barnes	10.44	0.43%	0.33%
fmm	13.04	0.01%	0.01%
ocean	4.31	0.21%	0.2%
blackscholes	12.61	0.93%	0.59%
bodytrack	11.56	0.17%	0.15%
x264	13.71	0.03%	0.03%
wupwise	13.48	0.03%	0.02%
swim	1.74	1.57%	1.48%
Apache	13.34	0.04%	0.03%
MySQL	13.59	0.03%	0.02%
average	10.78	0.35%	0.29%

Table 5.1: Recording performance

5.5.8 Store-buffer Hit Log Size

In addition to logging cache blocks feteched on cache miesses, our system also records memory counts of load instructions that hit in the store buffer to handle violations of store atomicity under TSO (Section 5.2.4.2). Figure 5.5.8 shows the stor buffer hit ratio on our simulation for each application, and Figure 5.21 shows corresponding log size. On average, about 2.45% of load instructions read their values from the processor-local store buffer, and thus store buffer hit log requires 21.9MB of log space to record one second of 8-threaded execution under TSO.

5.5.9 Recording Performance

As the last experiment, we analyze the performance overhead on recording for the processor configuration described in Section 5.5.1. On a cache miss, the fetched cache block is directly written back to the main-memory along with the current instruction count of the processor core. We also evaluated an optimization which the packets that write-back recorded logs are given a lower priority in the routers. Table 5.1 shows input log size and the performance degradation on

8 core configurations. `swim` shows the highest cache miss rates and requires largest input log size. The worst degradation is also for `swim` (1.57% slowdown). The priority optimization reduces the overhead to 1.48%. On average, the non-prioritized scheme incurs 0.35% slowdown, whereas prioritized scheme incurs 0.29% overhead.

5.6 Conclusion

Deterministic replay can significantly help programmers understand a multi-threaded program execution. Over the past few years, the architecture community has made significant progress is developing hardware designs that are both performance and space efficient. In this paper, we focused on reducing the hardware complexity of a recorder and supporting TSO memory model. We discussed a solution, where a program input log consisting mainly of the initial register state and cache miss data was sufficient for ensuring replay of the execution in multi-processor system. Much of the complexity is off-loaded to a novel symbolic analysis algorithm, which uses a SMT solver and determines the causal order between shared operations under the TSO model. We also discussed complexity-effective solutions for logging Strata hints that allowed us to bound the offline analysis, and cache-hit filtering optimization to reduce the number of memory operations that need to be analyzed to determine the causal order. These optimizations reduced the Strata log size by 3x and offline analysis time by 4.8x on average. We showed that offline analysis could in fact be 30% more efficient for a TSO execution that an SC execution.

CHAPTER 6

Conclusion

This chapter compares our proposed deterministic replay solutions, discusses lessons learned about holistic systems design for deterministic replay, presents future works, and concludes the thesis.

6.1 Comparisons and Contributions

Respec, an operating-system-based software-only system, is the first system that enables online replay on commodity hardware. Respec speculates that a program is data-race-free, and optimistically records synchronization operations and program input only. For data races, instead of logging the precise order of memory operations, Respec compares system call outputs and memory/register states of recorded and replayed processes at a semi-regular interval to check divergence while guaranteeing external determinism. When the check fails, Respec rolls back and retries the failed interval by serializing the threads and logging the schedule order. This technique results in low recording and replay overhead for the common case of data-race-free execution intervals and still ensures correct replay for execution intervals that have data races. The key strength of Respec is that it incurs very small latency overhead compared to other software-only solutions as it does not monitor individual memory operations, and thus Respec can provide online replay, in which both recording and replaying must be efficient. However, Respec performs best when extra cores are available since it needs an extra execution to detect divergence.

Chimera is the first system that leverages static program analysis in a deterministic replay field. Chimera records synchronization operations and solely potential data races reported by a static data race analysis. Chimera further reduces logging costs by leveraging profiled information and symbolic bounds analysis so that it can reduce the cost of logging false warnings produced by a conservative static analysis. Chimera does not monitor code regions running non-concurrently or accessing disjoint memory addresses. Instead, Chimera checks whether the profiled assumptions are satisfied at runtime. When source codes are available, and static

analysis is adequately precise, Chimera would be the best solution for efficient multiprocessor replay. Chimera can provide a stronger guarantee than Respec in that Chimera records the precise order of shared memory accesses by logging the order of newly added locks, but the performance overhead is similar to that of Respec. However, coarsened instrumentation granularity (e.g., loop-level or function-level locks) consequently does not allow the runtime system to observe fine-grained interleaving, implying that Chimera has limitations for the purpose of debugging. Moreover, the overall performance of Chimera is bounded by the precision of profiling and symbolic bounds analysis. In our experiments, we observe that compared to desktop and server applications, Chimera doesn't scale well to data-parallel scientific programs in which symbolic bounds analysis for multi-dimensional array accesses turns out to be very imprecise, and end up serializing many loop accesses. In such cases, Respec (or Doubleplay) will be a better solution and profiling could help identify the characteristic of applications. Nevertheless, Chimera shows that performance overhead is only 2.4 percent on average for Apache and desktop applications. In addition, we believe that the techniques in Chimera could also prove quite useful for enabling stronger semantics for concurrent languages such as sequential consistency and for enabling deterministic execution.

Rosa considerably reduces the complexity of hardware support in processor-based replay systems by not recording shared memory dependencies at all. Instead, Rosa determines a valid casual order which is compliant to an underlying memory model, using Satisfiability Modulo Theories (SMT) solver offline. Rosa provides solutions to support Sequential Consistency (SC) and Total Store Order (TSO) memory models. TSO is the most common consistency model implemented in modern processors, but not supported in many previous solutions. Rosa also includes a mechanism for bounding the search space of offline analysis. Compared to Respec and Chimera, Rosa incurs negligible performance overhead (less than one percent) at the cost of custom hardware support.

6.2 Holistic System Design

One of the key lessons that we learned from Respec and Rosa is that defining the right replay guarantee is very important in designing efficient replay systems. We observed that completely identical replay is often unnecessary. By relaxing replay guarantee, we could propose more complexity-efficient solutions. Respec only guarantees the same output and final state (external determinism) between two executions. Similarly, Rosa does not record shared-memory dependencies at all. Thread schedules reconstructed by SMT solver in Rosa could be different from the original execution. However, Rosa guarantees that each thread reproduces the same sequence of instructions and that each instruction reads and writes the same value as the recorded execution, which is sufficient for many replay uses, such as debugging.

The lessons we learned from Chimera include that 1) static analysis can help design efficient multiprocessor replay, and 2) a constrained runtime system also can help reduce the cost of monitoring and logging shared-memory dependencies. In addition to the lock-set based data race detection algorithm used in Chimera, we believe that there would be a lot of potential in another static analysis including static may-happen-in-parallel analysis. Chimera also takes advantage of a constrained runtime system in which thread interleaving happens at a larger granularity so that shared-memory dependencies can also be detected and logged at a coarse granularity, reducing the runtime overhead during recording.

Going forward, we believe that future multiprocessor replay systems should exploit the strengths of each approach. We imagine that the static analysis and constrained thread interleaving used in Chimera could improve other systems as well. Static analysis could be used to identify which objects need to be monitored in cases of misspeculation in Respec (instead of comparing whole memory states). If static analysis can figure out that a certain code region is data-race-free (though it does not look trivial for large code regions), then this ability can be used to turn on/off redundant executions to save twice the throughput cost. Moreover, by providing a constrained runtime system that prevents certain fine-grained interleaving while still preserving parallelism in common cases such as Chimera, Respec can reduce the chance of divergence between recorded and replayed processes. Similarly, Rosa can also reduce the cost of offline SMT analysis by restricting the search space. We see a deterministic execution system that guarantees the same thread interleaving for a given input (and thus does not need to log the thread schedule at all) as an extreme form of constrained runtime system. On the other hand, the current design of Chimera blindly records and replays all potential data races. However, as shown in Respec and Rosa, recording all data races is not necessary. One may identify a set of data races that may affect system outputs using dynamic taint analysis techniques, and design a system that guarantees the same happens before order only for those data races. Lastly, the offline search based approach used in Rosa looks promising when users want to have a very small recording overhead, but are willing to pay for more overhead during replay.

6.3 Future Work

For future work, we envision that the hybrid program analysis used for Chimera to build a deterministic replay system can be generalized and applied to other applications. We plan on developing a new speculation-based approach that leverages static program analysis to build efficient dynamic analysis tools such as data race detection.

Chimera, our second proposal, leverages static program analysis on deterministic replay. This work stems from the insightful observation that most memory accesses can be proved to be data-race-free by static data race analysis. However, existing static analysis for data race

detection is too imprecise and suffers from excessive false warnings due to 1) lack of runtime happened-before information and 2) conservative pointer analysis. We resolve this problem by leveraging profiled assumptions which can help prune out false positives and by performing light-weight runtime checks to detect misprofiling. We also discuss symbolic bounds analysis used for deriving memory regions that a code region may access, which can be used for testing disjointness at runtime. By monitoring only a subset of suspicious memory operations, we could build an efficient deterministic replay solution.

We envision that the proposed hybrid program analysis can be further generalized so that it can be applied to other types of dynamic analysis such as data-race detectors and memory safety checking. Previous works showed that information obtained from static analysis can help reduce the cost of dynamic tools. However, as shown in the work on Chimera, the main problems with static analysis, such as pointer alias analysis, are that they do not scale well or they are imprecise. Here, hybrid program analysis can play an important role to improve scalability or precision of static analysis, and eventually can be used to reduce the runtime cost of dynamic tools. Hybrid program analysis proposes to make assumptions about the common case (mostly via profiling) to guide the static analysis. Those mostly-true assumptions can help static analysis to be less conservative during analysis so that it can produce more precise results. With better static analysis, we may either reduce the number of dynamic checks or convert expensive checks into cheap ones, leading to improved runtime performance of dynamic tools. At runtime, we need to ensure that the assumptions we made during static analysis are satisfied. In case they fail, we would have to perform slower regular dynamic analysis.

Therefore, the performance benefits from hybrid program analysis depend on 1) how much we can improve static analysis, 2) how expensive the additional runtime checks to verify assumptions will be and 3) how frequently those assumptions will fail. Our future work will be to find appropriate assumptions that are mostly true, help static analysis, and are easy to check at runtime. We also want to find a proper dynamic analysis which can take advantage of profile guided static analysis.

6.4 Conclusion

We are now at the cusp of a major technological shift. Programmers should write parallel programs to extract performance from the next generation of multicore processors. However, today's tools provide very little support for developing parallel programs and programmers are left facing subtle and intermittent concurrency bugs. We believe that our research on deterministic replay can lead to significant improvements in programmability, security and reliability of parallel programs.

BIBLIOGRAPHY

[1] Fes2 simulator. http://fes2.cs.uiuc.edu.

[2] lpsolve, mixed integer linear programming solver. http://lpsolve.sourceforge.net/5.5.

[3] Sysbench. http://sysbench.sourceforge.net.

[4] ADVE, S. V., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. *IEEE Computer 29* (1995), 66–76.

[5] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 193–206.

[6] ANDERSEN, L. O. Program analysis and specialization for the c programming language. In *PhD thesis, DIKU, University of Copenhagen* (1994).

[7] ARVIND, A., AND MAESSEN, J.-W. Memory model = instruction reordering + store atomicity. *SIGARCH Comput. Archit. News 34*, 2 (2006), 29–40.

[8] AVIRAM, A., WENG, S.-C., HU, S., AND FORD, B. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, 2010).

[9] BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *In CC* (2004), Springer-Verlag, pp. 5–23.

[10] BARFORD, P., AND CROVELLA, M. Generating representative web workloads for network and server performance evaluation. In *In Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (1998).

[11] BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., AND GROSSMAN, D. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, 2010), pp. 53–64.

[12] BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications* (Orlando, FL, October 2009), pp. 81–96.

[13] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (2006), pp. 154–163.

[14] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (October 2008).

[15] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel Java. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications* (Orlando, FL, October 2009), pp. 97–116.

[16] BOOTHE, B. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (2000), pp. 299–310.

[17] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 80–107.

[18] CEZE, L., TUCK, J., MONTESINOS, P., AND TORRELLAS, J. Bulksc: bulk enforcement of sequential consistency. In *ISCA* (2007), pp. 278–289.

[19] CHEN, Y., HU, W., CHEN, T., AND WU, R. Lreplay: A pending period based deterministic replay scheme. In *ISCA* (Saint-Malo, France, June 2010).

[20] CHOI, J. D., ALPERN, B., NGO, T., AND SRIDHARAN, M. A perturbation free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium* (April 2001).

[21] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008), pp. 1–14.

[22] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009), pp. 85–96.

[23] DEVIETTI, J., NELSON, J., BERGAN, T., CEZE, L., AND GROSSMAN, D. Rcdc: a relaxed consistency deterministic computer. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 67–78.

[24] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[25] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M., AND CHEN, P. M. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2008), pp. 121–130.

[26] DUTERTRE, B., AND DE MOURA, L. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference* (2006), vol. 4144 of *LNCS*, Springer-Verlag, pp. 81–94.

[27] FELDMAN, S. I., AND BROWN, C. B. IGOR: A system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (1988), pp. 112–123.

[28] FLANAGAN, C., AND FREUND, S. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 2009), pp. 121–133.

[29] FRASER, K., AND CHANG, F. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003), pp. 325–338.

[30] GEORGES, A., CHRISTIAENS, M., RONSSE, M., AND BOSSCHERE, K. D. Jarec: A portable record/replay environment for multi-threaded java applications. In *Software: Practice and Experience* (2004).

[31] GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing* (1991), pp. 355–364.

[32] GWENDOLYN VOSKUILEN, FARAZ AHMAD, T. V. Timetraveler: Exploiting acyclic races for optimizing memory race recording. In *ISCA* (Saint-Malo, France, June 2010).

[33] HOWER, D., DUDNIK, P., HILL, M. D., AND WOOD, D. A. Calvin: Deterministic or not? Free will to choose. In *17th International Conference on High-Performance Computer Architecture* (February 2011), pp. 333–334.

[34] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th International Symposium on Computer Architecture* (June 2008), pp. 265–276.

[35] HUANG, J., LIU, P., AND ZHANG, C. Leap: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2010), pp. 207–216.

[36] INTEL CORPORATION. Intel 64 architectures memory ordering white paper. Tech. rep., 2007.

[37] INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developer's Manual (rev.30). Tech. rep., 2009.

[38] KAHLON, V., SINHA, N., KRUUS, E., AND ZHANG, Y. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2009), pp. 13–22.

[39] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference* (April 2005), pp. 1–15.

[40] LEBLANC, T. J., AND MELLOR-CRUMMEY, J. M. Debugging parallel programs with instant replay. *IEEE Transaction on Computers 36*, 4 (1987), 471–482.

[41] LEE, D., SAID, M., NARAYANASAMY, S., YANG, Z. J., AND PEREIRA, C. Offline symbolic analysis for multi-processor execution replay. In *International Symposium on Microarchitecture (MICRO)* (2009).

[42] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 327–336.

[43] LOWELL, D. E., CHANDRA, S., AND CHEN, P. M. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

[44] LUCIA, B., DEVIETTI, J., STRAUSS, K., AND CEZE, L. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th International Symposium on Computer Architecture* (Beijing, China, 2008), pp. 277–288.

[45] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[46] MAGNUSSON, S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÅLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. Simics: A full system simulation platform. *IEEE Computer 35*, 2 (2002), 50–58.

[47] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture* (June 2008), pp. 289–300.

[48] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 73–84.

[49] NARAYANASAMY, S., PEREIRA, C., AND CALDER, B. Recording shared memory dependencies using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 229–240.

[50] NARAYANASAMY, S., PEREIRA, C., PATIL, H., COHN, R., AND CALDER, B. Automatic logging of operating system effects to guide application-level architecture simulation. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)* (June 2006), pp. 216–227.

[51] NARAYANASAMY, S., POKAM, G., AND CALDER, B. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture* (June 2005), pp. 284–295.

[52] NARAYANASAMY, S., WANG, Z., TIGANI, J., EDWARDS, A., AND CALDER, B. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (San Diego, CA, June 2007).

[53] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), pp. 213–228.

[54] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[55] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, March 2008), pp. 308–318.

[56] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 2006), pp. 1–14.

[57] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009), pp. 97–108.

[58] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 361–376.

[59] OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. ...and region serializability for all. In *Proceedings of the fifth USENIX Workshop on Hot Topics in Parallelism* (2013), HotPar '13.

[60] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 177–191.

[61] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. PinPlay: A framework for determinisrtic replay and reproducible analysis of parallel programs. In *Proceedings of the 2010 IEEE/ACM International Symposium on Code Generation and Optimization* (March 2010).

[62] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating bugs as allergies—a safe method to survive software failures. In *ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 235–248.

[63] RONSSE, M., AND DE BOSSCHERE, K. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems 17*, 2 (May 1999), 133–152.

[64] RUGINA, R., AND RINARD, M. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), pp. 182–195.

[65] RUGINA, R., AND RINARD, M. C. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst. 27*, 2 (2005), 185–235.

[66] SARANGI, S., NARAYANASAMY, S., CARNEAL, B., TIWARI, A., CALDER, B., AND TORRELLAS, J. Patching processor design errors with programmable hardware. *IEEE Micro Top Picks 27*, 1 (2007), 12–25.

[67] SPARC INTERNATIONAL, INC., C. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[68] SPEC. Standard performance evaluation corporation - http://www.spec.org.

[69] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND ZHOU, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 29–44.

[70] STEENSGAARD, B. Points-to analysis in almost linear time. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (1996), pp. 32–41.

[71] STEVEN, J., CHANDRA, P., FLECK, B., AND PODGURSKI, A. jrapture: A capture replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis* (2000).

[72] TUCEK, J., LU, S., HUANG, C., XANTHOS, S., AND ZHOU, Y. Triage: Diagnosing production run failures at the user's site. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (October 2007), pp. 131–144.

[73] UCLIB.ORG. uClibc, a C library for embedded Linux. http://uClibc.org.

[74] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 369–384.

[75] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Doubleplay: parallelizing sequential logging and replay. In *ASPLOS* (Newport Beach, CA, 2011).

[76] VOUNG, J. W., JHALA, R., AND LERNER, S. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (Dubrovnik, Croatia, 2007), pp. 205–214.

[77] WESTER, B., DEVECSERY, D., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Parallelizing data race detection. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 27–38.

[78] WILSON, R. P., AND LAM, M. S. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1995), pp. 1–12.

[79] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June 1995), pp. 24–36.

[80] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The splash-2 programs: Characterization and methodological considerations. In *ISCA* (1995), pp. 24–36.

[81] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture* (June 2003), pp. 122–135.

[82] XU, M., BODIK, R., AND HILL, M. D. A regulated transitive reduction (RTR) for longer memory race recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 49–60.

[83] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *MOBS07* (June 2007).