Uniparallel execution and its uses

by

Kaushik Veeraraghavan

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
2011

Doctoral Committee:
Associate Professor Jason N. Flinn, Chair
Professor Peter M. Chen
Associate Professor Brian D. Noble
Assistant Professor Eytan Adar

To the women who have shaped my life:

My grandmother, Jayalakshmi Seshaier Chittur

My mother, Rukmini Veeraraghavan

My wife, Natasha Anna Chang

My daughter, Annika Jayalakshmi Veeraraghavan

ACKNOWLEDGEMENTS

I started working with Jason Flinn in September 2005, right after starting at Michigan. Over the past 6 years, I have benefited tremendously from Jason's guidance in taking my ideas from conception to realization. I have found Jason to be both incredibly patient, as he spent countless hours with me discussing research and hacking systems, and generous, as he permitted me to work remotely during my final year at Michigan. I have also been lucky in getting to know Jason outside of work as he introduced me to bridge; amongst the things I miss most about Ann Arbor are our Thursday evening bridge games. I am deeply indebted to Jason for taking me on as a graduate student and mentoring me over the years.

All the work described in this thesis was developed in collaboration with Peter Chen. Despite not serving as an official advisor, I have often turned to Pete for advice. I am thankful for the opportunity to work with him. Prior to serving on my thesis committee, Brian Noble also served on my qualifier committee and as a surrogate advisor while I was working on quFiles. Brian taught me the importance of seeking out the "big picture" in my work and in life, and for this I am grateful. I thank Eytan Adar for serving on my committee and for his insightful questions during my thesis proposal. I also thank Satish Narayanasamy who greatly contributed to the work in this thesis, and Venugopalan "Rama" Ramasubramanian and Doug Terry whose mentoring during my internship at MSR-Silicon Valley boosted my self-confidence.

My officemates in CSE 4929 have influenced my development both as a researcher and as an individual. Dan Peek and Benji Wester doubled up as both officemates and housemates, and are amongst my closest friends. Both have given freely of their time: Dan in answering my innumerable questions on everything from operating systems to

file systems and distributed systems, and Benji in helping me debug the Respec and Speculator kernels. Ed Nightingale has been an excellent mentor and friend; I cherish our bike rides on Huron River Road where we discussed everything from Speculator to the next triathlon we'd compete in. Mona Attariyan always made the time to listen to my wild-eyed ideas and provide encouragement when I needed it most. Dongyoon Lee pulled many all-nighters with me evaluating DoublePlay. I also thank Jessica Ouyang, Jie Hou, Ya-Yunn Su, Manish Anand, Evan Cooke, Sushant Sinha, Ashwini Kumar, Jon Oberheide and Andrew Myrick for helping make my work better through discussion and collaboration.

Life in Ann Arbor wouldn't have been fun without Amy Kao Wester, Arnab Nandi, Simone Rauscher, Nate Derbinsky and Mojtaba Mehrara who kept me sane after all those hours in the lab.

I would like to thank my family and friends for all their love and encouragement over the years. My mother, Rukmini, has always supported me in my decisions. My sister, Sneha, has been a constant source of excitement and cheer. My numerous cousins, aunts and uncles have provided me with perspective beyond life at school and vacations to look forward to.

I would like to thank my wife, Natasha, for her love and infinite patience in enduring the travails of the first year of our marriage which we spent apart, and my daughter Annika for providing me with the proverbial kick-in-the-pants so I'd graduate. Lastly, I would like to thank Natasha's parents, Susan and Luis—this dissertation wouldn't have seen the light of day without their help in juggling life with Annika.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER 1

Introduction

As the scaling of clock frequency has reached the boundary of physical limitations, to improve performance, the semiconductor industry has switched from developing uniprocessors with a single processing unit to multicores and multiprocessors that package multiple processing units. These multicore and multiprocessor designs are becoming ubiquitous in the servers, desktops, laptops and cellphones we use today, and the trend of increasing processor counts is expected to continue in the foreseeable future [88].

One common way to exploit the hardware parallelism of multicore and multiprocessor machines is to run multithreaded programs, which split their execution into distinct threads that communicate via shared memory. These threads run concurrently on the available processors and scale performance with increasing cores. Multithreaded programs are increasingly used in a wide range of domains including scientific computing, network servers, desktop applications, and mobile devices.

Unfortunately, it has proven very difficult to write correct multithreaded programs. For instance, there were several cases of radiation overdoses involving the Therac-25 radiation therapy machine in which a high-power electron beam was activated instead of a low-power beam due to a data race [47]. Similarly, the Northeast blackout of 2003 was partly blamed on a data race bug in GE's energy management software—this bug

affected 45 million residents in the US and 10 million in Canada [68].

The systems community has proposed many solutions to improve the reliability of multithreaded programs. These solutions include software testing frameworks [81, 56] and data race detectors [78, 26, 75, 82] that attempt to identify bugs during the software development process, and deterministic replay systems that record a programs execution for offline debugging [85, 41, 89] and forensic analysis [24]. Researchers have also proposed alternate approaches such as transactional memory [83], which attempts to simplify how developers write multithreaded programs and deterministic execution [7, 2], which tries to guarantee that a multithreaded program will always generate the same output for a given input.

While techniques like deterministic replay and data race detection can improve program reliability, they are often too expensive for practical use on multithreaded programs executing on a multiprocessor. The primary cause for the slowdown is the overhead of tracking shared memory dependencies. Specifically, on a multiprocessor, threads of the program execute concurrently on different processors and might simultaneously update shared memory. Not only are these shared memory accesses expensive to instrument and track, but they also happen very frequently. Deterministic replay solutions can add up to 9X overhead to program execution if they log and replay shared memory dependencies [25]. The state-of-the-art dynamic data race detection tools add about 8.5X overhead for managed code [29] and about 30X overhead for non-managed code (e.g., programs written in languages like C/C++) [82].

The key insight in this thesis is that there exist many techniques that are difficult or slow to achieve on a multiprocessor, but are easy and efficient on a uniprocessor. For instance, deterministic replay on uniprocessors imposes little overhead [74] and is available in commercial products by companies like VMware. When a multithreaded program runs on a uniprocessor, only one thread of the program executes at any given time. Hence, only the schedule with which threads are multiplexed on the processor needs to be tracked to recreate the order of shared memory accesses. As thread context-switches are much more infrequent than shared memory accesses, these techniques add significantly lesser overhead on a uniprocessor.

We exploit our insight by proposing *uniparallelism*: a new model of execution that achieves the benefits of executing on a uniprocessor, while still allowing application performance to scale with increasing processors. This dissertation demonstrates the utility of uniparallelism by addressing two challenging problems. First, we use uniparallelism to implement a software-only deterministic replay system that can record multithreaded execution on a commodity multiprocessor and guarantee offline replay. Next, we use uniparallelism to implement a replication system that can detect data races at low overhead and also increase system availability by masking the effects of harmful data races at runtime. Note that uniparallelism is implemented within the operating system so applications benefit without requiring any modification.

1.1 Uniparallel execution

We observe that there are (at least) two ways to run a multithreaded program on multiple processors, which we call *thread parallelism* and *epoch parallelism*. With thread parallelism, the threads of a multithreaded program run on multiple processors. This traditional method of parallelization can achieve good scalability. With epoch parallelism, multiple time intervals (*epochs*) of the program run concurrently. This style of parallelism has also been called Master/Slave Speculative Parallelism by Zilles [108] and Predictor/Executor by Süßkraut [87].

A *uniparallel execution* consists of a single thread-parallel execution and one or more epoch-parallel executions of the same program. Each epoch-parallel execution runs all threads of a given epoch on a single processor at a time; this enables the use of techniques that only run efficiently on a uniprocessor. Unlike a traditional thread-parallel execution that scales with the number of cores by running different threads on different cores, an epoch-parallel execution achieves scalability in a different way: by running different epochs of the execution concurrently on multiple cores. To run future epochs before prior epochs have completed, epoch-parallel execution requires the ability to predict future program states. These predictions are generated by running a thread-parallel execution concurrently.

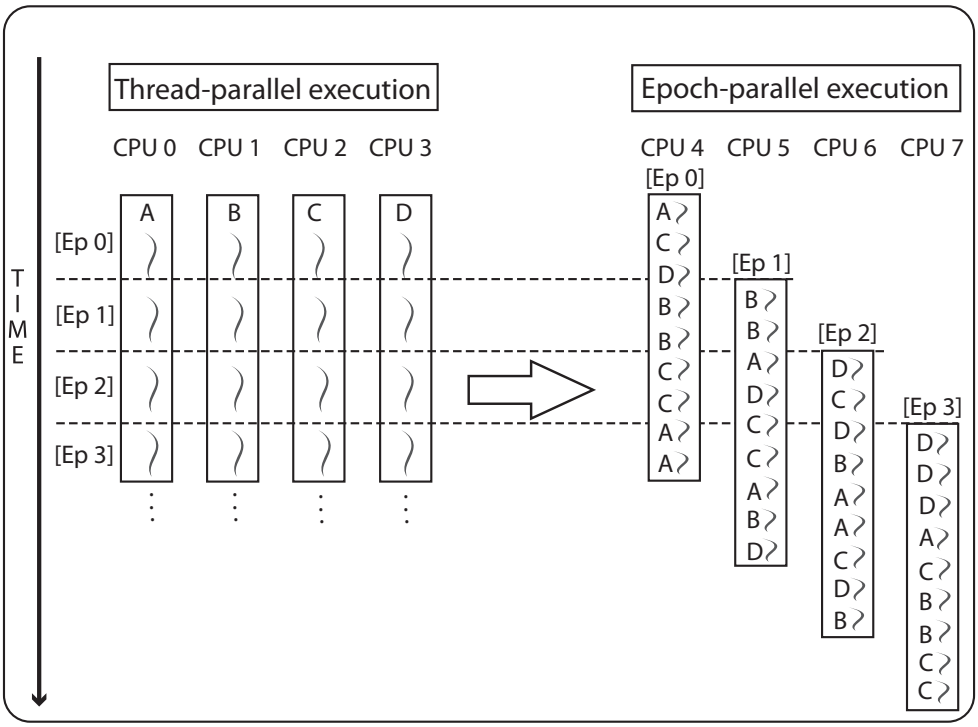


Figure 1.1: Uniparallel execution

Consequently, for CPU-bound workloads, uniparallel execution requires at least twice the number of cores as a traditional execution (for one thread-parallel execution and one epoch-parallel execution), leading to an approximately 100% increase in CPU utilization and energy usage (assuming energy-proportional hardware). If the technique requires additional epoch-parallel executions, the utilization cost of uniparallel execution for CPU-bound workloads increases proportionally (e.g., a 200% increase for two epoch-parallel executions).

1.1.1 Starting a uniparallel run

Figure 1.1 depicts the uniparallel execution of a program with 4 threads, with one thread-parallel and one epoch-parallel execution. Each execution uses multiple cores. The execution on the left is the thread-parallel execution that runs threads concurrently on the cores allocated to it. The thread-parallel execution is partitioned into time intervals that we call *epochs*. At the beginning of each epoch, a copy-on-

write checkpoint of the state of the thread-parallel execution is created.

The execution on the right is the epoch-parallel execution that runs each epoch on one of the cores allocated to it. All threads of a given epoch run on the same core. We achieve good performance by running *different* epochs of the same process *concurrently*. As shown in Figure 1.1, both the thread-parallel execution and the epoch-parallel execution start running the first epoch [Ep 0] simultaneously from the same initial state. However, the thread-parallel execution runs the epoch much faster because it uses more cores. When the thread-parallel execution reaches the start of the second epoch, we checkpoint the process state and use that state to start running the second epoch [Ep 1] in the epoch-parallel execution. By the time the thread-parallel execution is running the fourth epoch [Ep 3], the epoch-parallel execution is able to fully utilize the four cores allocated to it. From this point on, the epoch-parallel execution can utilize its allotment of cores to achieve speedup from parallelization roughly equivalent to that achieved by the thread-parallel execution.

1.1.2 Releasing program output

The two types of executions can be viewed as follows. Each epoch of the epoch-parallel executions runs on a single core, so we can apply any of the simple and efficient uniprocessor techniques to the execution. The thread-parallel execution allows the epoch-parallel executions to achieve good performance and scale with the number of cores. The thread-parallel execution provides a *hint* as to what the future state of the process execution will be at each epoch transition. As long as this hint is correct, the state of the process at the beginning of each epoch in the epoch-parallel execution will match the state of the process at the end of the previous epoch. This means that the epochs can be pieced together to form a single, natural execution of the process. This process is akin to splicing movie segments together to form a single coherent video.

But, what if the hint is incorrect? For instance, a data race could cause an epoch-parallel execution to write an incorrect value to a shared data structure. In such

an instance, two epochs of an epoch-parallel execution cannot be pieced together to form a single natural run; the logged run will contain unnatural transitions in program values at epoch boundaries akin to artifacts in a bad video splice. To detect such events, the process state (memory values and registers) of the thread-parallel and each of the epoch-parallel runs are compared at each epoch transition.

If the state of the executions mismatch, one of several recovery strategies is employed. One option is to declare the state of one of the executions as correct and accept its result, which we refer to as *committing* that execution. Another option is to mark this epoch as suspect and *roll back* the epoch to the last checkpoint before the divergence and run additional executions to learn more about why the epoch lead to divergences. We describe both recovery strategies in detail in Section 4.3.2.

Since the thread and epoch-parallel executions may produce different output, the uniparallel execution is prevented from externalizing any output (e.g., console or network messages) until which execution to commit is decided upon. Our implementation of uniparallelism uses speculative execution [61] to defer when output is released.

1.1.3 Protecting against divergences

A divergence between the thread-parallel and epoch-parallel executions can slow performance substantially because it may lead to squashing the execution of several subsequent epochs or the need to run additional executions to decide on an epoch outcome. Uniparallel execution uses *online replay* [46] to reduce the frequency of such divergences and their resulting slowdown.

During the thread-parallel execution, the ordering and results of all system calls and low-level synchronization operations in `libc` are logged. When an epoch-parallel execution executes a system call or synchronization operation, the logged result is returned instead of executing the operation. If needed, thread execution is also delayed to match the order of operations in the log. Further, signals are logged and delivered only on kernel entry or exit, making their effects deterministic. These actions are sufficient to ensure that the thread-parallel and epoch-parallel executions are iden-

tical (and, hence, do not diverge) for any epochs that are free of data races [73]. The only situation in which the thread-parallel and epoch-parallel executions might diverge is when the program contains a data race, causing two threads to execute unsynchronized, conflicting operations on a shared memory address.

1.1.4 When should one use uniparallelism?

The cost of uniparallelism is that there are multiple executions and therefore an increased utilization of hardware resources. Uniparallelism affects the throughput but not the latency as the uniprocessor execution of a time interval can be started as soon as a checkpoint is available. In our evaluation, we find that uniparallelism adds an average throughput overhead of less than 28% if there are spare cores available on the machine. If all cores can be productively used by the application, uniparallelism incurs a much higher overhead because it splits the cores between the multiple executions of the program. In this case, we find that deterministic replay imposes a 2X overhead, while data race detection requires 3X, proportional to the number of replicas. However, even in the absence of spare cores, both techniques still perform substantially better running in the uniparallel execution model than in other state-of-the-art systems.

We believe the advent of multicore processors makes the increased hardware utilization of uniparallelism a worthwhile tradeoff: the number of cores per computer is expected to grow exponentially, and scaling applications to use these extra cores is notoriously difficult.

1.2 Deterministic record and replay

Our first test of the uniparallel execution model was in a deterministic record and replay system. Our goal was to reduce the overhead of logging multithreaded execution on commodity multiprocessors while guaranteeing offline replay. To achieve this, we employ uniparallelism to run a thread-parallel and an epoch-parallel execution of the program concurrently. The epoch-parallel execution, which is the run being

recorded, constrains all threads for a given epoch to a single processor. This simplifies logging because threads in the same epoch never simultaneously access the same memory and because different epochs operate on different copies of the memory. Thus, rather than logging the order of shared memory accesses, we need only log the order in which threads in an epoch are timesliced on the processor. The main overhead of this approach is the use of more cores to run two instances of the program during logging.

We implement these ideas in *DoublePlay*, a new system that takes advantage of spare cores to log multithreaded programs on multiprocessors at low overhead and guarantees being able to replay them deterministically. We evaluate the performance of DoublePlay on a variety of parallel client, server, and scientific benchmarks. We find that, with spare cores, DoublePlay increases run time by an average of 15% with two worker threads and 28% with four worker threads, and that DoublePlay can replay the run later without much additional overhead. On computers without spare cores, DoublePlay adds approximately 100% overhead for CPU-bound applications that can scale to 8 cores. This compares favorably with other software solutions that provide guaranteed deterministic replay.

1.3 Detecting and surviving data races

A data race occurs when two threads concurrently access the same shared memory location without being ordered by a synchronization operation, and at least one of the accesses is a write. Data races are responsible for many concurrency bugs. As data races often only manifest during rare thread interleavings, they might be missed during development only to materialize in production.

In the second part of this thesis we describe Frost[1], a new system that protects a program from data race bugs at runtime by combining uniparallelism and *complementary schedules*. Frost uses uniparallelism to split the program into epochs of

[1]Our system is named after the author of the poem "The Road Not Taken". Like the character in the poem, our second replica deliberately chooses the schedule not taken by the first replica.

execution; multiple replicas of each epoch are run as distinct epoch-parallel executions that differ only in the complementary schedule they follow. Complementary schedules are a set of thread schedules constructed to ensure that replicas diverge only if a data race occurs and to make it very likely that harmful data races cause divergences.

Complementary schedules work by exploiting a sweet spot in the space of possible thread schedules. First, Frost runs an additional thread-parallel execution. Apart from generating checkpoints from which future epochs can be started (thus enabling Frost to scale performance with increasing processors by running multiple epochs simultaneously), the thread-parallel execution can also be logged to ensure that all replicas of an epoch see identical inputs and use thread schedules that obey the same program ordering constraints imposed by synchronization events and system calls. This guarantees that replicas that do not execute a pair of racing instructions will not diverge [73]. Second, while obeying the previous constraint, Frost attempts to make the thread schedules executed by two replicas as dissimilar as possible. Specifically, Frost tries to maximize the probability that any two instructions executed by different threads and not ordered by a synchronization operation or system call are executed in opposite orders by the replicas. For all harmful data races we have studied in actual applications, this strategy causes replica divergence.

Frost enforces complementary schedules by constraining each epoch-parallel execution to a single processor and switching between threads only at synchronization points (i.e., it uses non-preemptive scheduling). In addition to permitting tight control over the thread schedules, running threads on a single processor without preemptions prevents bugs that require preemptions (e.g., atomicity violations) from manifesting, thereby increasing availability.

To distinguish buggy replicas from correct ones, Frost introduces *outcome-based race detection* which compares the output and memory state of replicas executed with complementary schedules, to detect the occurrence of a data race. For production systems, Frost also helps diagnose the type of data race bug and select a recovery strategy that masks the failure and ensures forward progress. Our evaluation of Frost

on 11 real data race bugs shows that Frost both detects and survives all of these data races with a reasonable overhead of 3—12% if there are sufficient cores or idle CPU cycles to run all replicas.

1.4 Thesis

My thesis is:

Uniparallelism allows applications to benefit from the simplicity of uniprocessor execution while scaling performance with increasing processors. With operating system support for uniparallelism, techniques that are easy to achieve on a uniprocessor but difficult or slow on a multiprocessor, such as guaranteed deterministic replay, data race detection, and data race survival, can be deployed with reasonable overhead on production software systems running on commodity multiprocessors without any application modification.

1.5 Roadmap

The rest of this manuscript validates the thesis.

Chapter 2 provides background information about the design and implementation of the Respec online multiprocessor replay system [46]. Our implementation of uniparallelism uses Respec to log the thread-parallel execution. The epoch-parallel execution is treated as an online replay of the logged thread-parallel run to minimize the likelihood of divergence.

Chapter 3 describes the design, implementation and evaluation of DoublePlay, an efficient deterministic record and replay system for multithreaded programs running on multiprocessors. DoublePlay significantly lowers the overhead of logging by recording a epoch-parallel execution instead of the traditional thread-parallel execution. Additionally, since the epoch-parallel schedule is verified using online replay, later offline replays are guaranteed to succeed.

Chapter 4 describes the design, implementation and evaluation of Frost, an online data race detection and survival system for multithreaded programs running on multiprocessors. Frost uses uniparallel execution to run replicas with tightly controlled complementary schedules that detect data races and mask their harmful effects. Uniparallel execution ensures that the overhead of Frost is low enough for use in production environments.

Chapter 5 describes related work. Chapter 6 describes future work and summarizes the contribution of this thesis.

CHAPTER 2

Background

Uniparallelism leverages our prior work on the Respec online replay system to ensure that the thread-parallel and epoch-parallel executions remain in sync. This section motivates the need for online replay and summarizes the design and implementation of the Respec system, a full description of which can be found in the paper by Lee et al. [46].

2.1 Introduction

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 does not address deterministic replay of a process after the original execution completes; therefore, it cannot be used for offline debugging, intrusion analysis, and other activities that must take place after an execution finishes. Chapter 3 describes how DoublePlay builds upon Respec to provide this capability.

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. We argue in Section 2.2.1 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 [73], 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 [61] 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.

2.2 Replay guarantees

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

2.2.1 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 [15, 24, 102]. Deterministic replay can also be provided at higher levels of a system, such as a Java virtual machine [18] or a Unix process [85], or lower levels of a system, such as cycle accuracy for interconnected components of a computer [77]. 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 reproduce 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 conflicting 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 [73]. 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 [15], 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 [28, 41], 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 [62]. 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 guarantee 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.

2.2.2 Online versus offline replay

Different uses of deterministic replay place different constraints on replay speed. For some uses, such as debugging [28, 41] 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 [67].

For other uses of replay, such as fault tolerance and decoupled [20] or parallel checks [62], 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 [48].

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 [67]. 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 [1, 67, 45] could be used.

2.3 Design

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

2.3.1 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 2.1 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

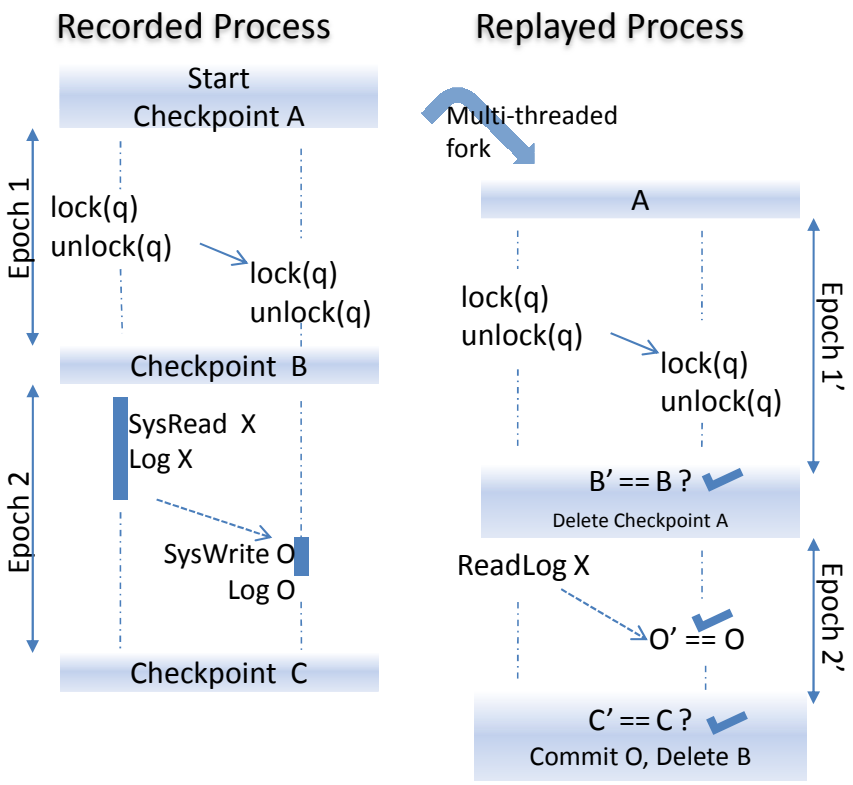


Figure 2.1: An execution in Respec with two epochs.

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 [61] 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.

2.3.2 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 2.2. 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 2.4.3.1), the replayed process is more likely to reproduce the same memory

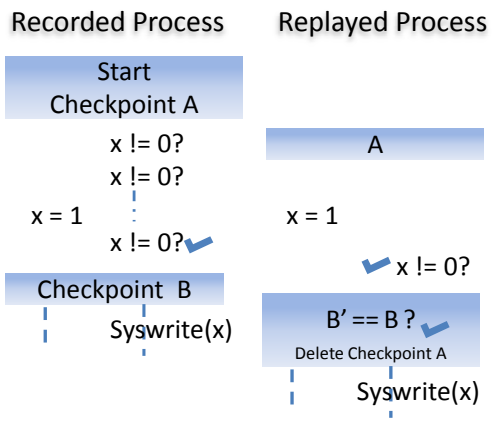


Figure 2.2: A data race that leads to convergent state

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.

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 [59]. 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 2.2 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.

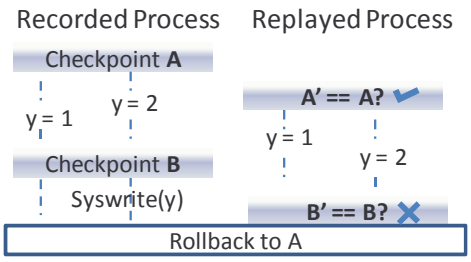


Figure 2.3: A data race that leads to divergent state

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.

However, for harmful races that should happen rarely, Respec's divergence check could trigger a rollback. Figure 2.3 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 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.

2.4 Implementation

2.4.1 Checkpoint and multithreaded fork

Rollback and recovery implementations often use the Unix copy-on-write `fork` primitive to create checkpoints efficiently [32, 61]. 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 *multithreaded 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: 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 frequency 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 [63] 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.

2.4.2 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 [61] 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.

2.4.3 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.

2.4.3.1 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.

However, for an application that heavily uses handcrafted synchronizations our approach might lead to frequent rollbacks. A simple solution would be to require programmers to annotate synchronization accesses so that we could instrument and log them. In fact, recently proposed Java [52] and C++0x [13] memory models already

require programmers to explicitly annotate synchronization accesses using `volatile` and `atomic` keywords.

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 2.3.2.

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.

2.4.3.2 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 [66].

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.

2.4.4 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 particu-

lar, 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.

2.4.5 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.

2.5 Summary

This chapter describes the Respec multiprocessor online replay system. Respec ensures that two concurrent executions of the same process are externally deterministic, which we define to mean that the two executions execute the same system calls in the same order, and that the program state (values in the address space and registers) of the two processes are identical at the end of each epoch of execution. As the next chapter describes, Respec provides the basic infrastructure that DoublePlay extends using uniparallelism to guarantee deterministic offline multiprocessor replay.

CHAPTER 3

DoublePlay: parallelizing sequential logging and replay

This chapter describes how uniparallel execution can be used to create a practical deterministic replay system for recording the execution of a production software system running on commodity multiprocessors while guaranteeing offline replay.

3.1 Introduction

Deterministic replay systems record the execution of a hardware or software system for later replay. As the execution is deterministic for the most part, deterministic replay systems need only log and reproduce the non-deterministic events encountered during recording.

The specific set of what non-deterministic events need to be recorded depends of the abstraction at which deterministic replay operates. In this thesis, we are interested in Operating System-level replay which requires that system level non-deterministic events be captured. As OS-level non-deterministic events such as interrupts, system calls and signals are rare, recording a uniprocessor execution is very efficient. If the program is multithreaded, it is still the case that only one thread runs on the processor at any instance so the order of shared memory updates can be recreated by logging the schedule with which threads are context switched on the processor.

On multiprocessors, however, shared-memory accesses add a high-frequency source

of nondeterminism, and logging and replaying these accesses can drastically reduce performance. Many ideas have been proposed to reduce the overhead of logging and replaying shared-memory, multithreaded programs on multiprocessors, but all fall short in some way. Some approaches require custom hardware [100, 57, 55, 35]. Other approaches cannot replay programs with data races [73] or are prohibitively slow for applications with a high degree of sharing [25]. Some recent approaches provide the ability to replay only while the recording is in progress [46] or sacrifice the guarantee of being able to replay the recorded execution without the possibility of a prohibitively long search [67, 1, 98, 107].

In this chapter, we describe a new way to guarantee deterministic replay on commodity multiprocessors. Our method combines the simplicity and low recording overhead of logging a multithreaded program on a uniprocessor with the speed and scalability of executing that program on a multiprocessor.

Our insight is that one can use the simpler and faster mechanisms of single-processor record and replay, yet still achieve the scalability offered by multiple cores, by using an additional execution to parallelize the record and replay of an application. Our goal is for the single-processor execution to be as fast as a traditional parallel execution, but to retain the ease-of-logging of single-processor multithreaded execution.

We accomplish this goal by using uniparallelism to run a thread-parallel and an epoch-parallel execution of the program concurrently. Unlike traditional approaches which log the thread-parallel execution, we log the epoch-parallel execution. As the epoch-parallel execution is constrained so all threads for a given epoch execute on a single processor, we can reuse the well studied technique of uniprocessor replay. Thus, rather than logging the order of shared-memory accesses, we need only log the order in which threads are context-switched on the processor.

Uniparallelism scales effectively by running different epochs concurrently on separate processors. As described in Section 1.1.1, we treat the thread-parallel execution as a predictor of future state and checkpoint its execution at the start of each epoch. Each of the epoch-parallel executions is started from a checkpoint and operates on

its own copy of memory allowing multiple epochs to be logged in parallel.

This approach supports deterministic replay on commodity multiprocessors without needing to log shared-memory accesses. Replaying multithreaded programs on multiprocessors thus becomes as easy as replaying multithreaded programs on uniprocessors, while still preserving the speed and scalability of parallel execution. The main overhead of this approach is the use of more cores to run two instances of the program during logging.

To demonstrate these ideas, we implement a system called *DoublePlay* that can take advantage of spare cores to log multithreaded programs on multiprocessors at low overhead and guarantee being able to replay them deterministically.

The rest of this chapter is organized as follows. Sections 3.2 and 3.3 describe the design and implementation of DoublePlay. Section 3.4 reports on how DoublePlay performs for a range of client, server and scientific parallel benchmarks. Section **??** describes our future work, and Section 3.5 concludes.

3.2 Design

The goal of DoublePlay is to efficiently record the execution of a process or group of processes running on a multiprocessor such that the execution can later be deterministically replayed as many times as needed. DoublePlay is implemented inside the Linux operating system, and its boundary of record and replay is the process abstraction. The operating system itself is outside the boundary of record and replay, so DoublePlay records the results and order of system calls executed by the process and returns this data to the application during replay.

Figure 3.1 shows an overview of how DoublePlay records process execution. DoublePlay uses uniparallelism so it simultaneously runs two executions of the program being recorded. The execution on the left is the thread-parallel execution and the one on the right is the epoch-parallel execution. As described in Section 1.1, the thread-parallel execution is checkpointed at the start of each epoch and the checkpoint is used to start running the corresponding epoch in the epoch-parallel execution.

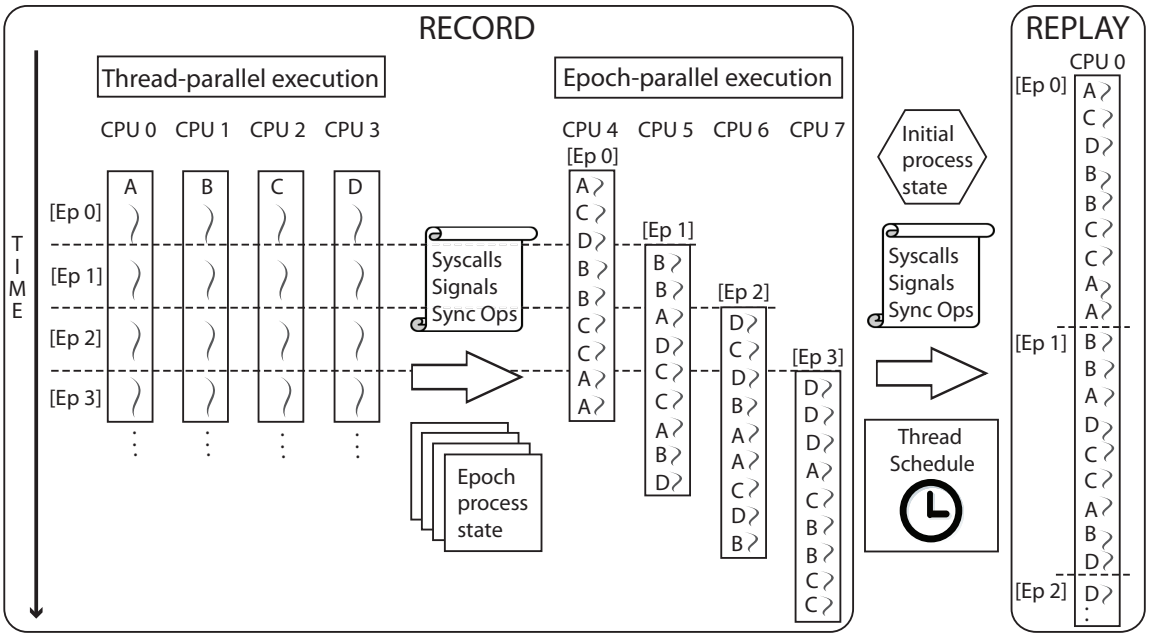


Figure 3.1: Overview of DoublePlay record and replay

During recording, DoublePlay saves three items that are sufficient to guarantee that the process execution can be replayed deterministically in the future. First, DoublePlay records the initial state of the process at the start of recording. Second, DoublePlay records the order and results of system calls, signals, and low-level synchronization operations in GNU `libc`. Finally, DoublePlay records the schedule of thread execution (i.e., when context switches between threads occur) of the epoch-parallel execution. Note that since each epoch is executed on a single core in the epoch-parallel execution, DoublePlay does not need to record the ordering or results of any shared memory operations performed by multiple threads; the three items above are sufficient to exactly recreate identical operations during replay.

The two executions can be viewed as follows. The epoch-parallel execution is the actual execution of the program that is being recorded. Because each epoch of the epoch-parallel execution runs on a single core, DoublePlay can use the simple and efficient mechanisms for uniprocessor deterministic replay to record and replay its execution. The thread-parallel execution allows the epoch-parallel execution to achieve good performance and scale with the number of cores.

Each checkpoint captures the state of the thread-parallel execution at an epoch transition. DoublePlay compares the process state (i.e., register and memory values) of the epoch-parallel run to the corresponding checkpoint at each epoch transition.

As described in Section 1.1, we reduce the change of divergence of the thread-parallel and epoch-parallel executions by combining speculative execution [61] and online replay [46]. The thread-parallel execution runs speculatively so all of its output is buffered. We log a subset of the non-deterministic events in the thread-parallel execution (e.g., the order of synchronization operations), and use this log to guide the execution of the epoch-parallel execution down a similar path. At the end of each epoch, we check that the epoch-parallel execution has arrived at a matching state as the subsequent thread-parallel checkpoint. If yes, we commit the speculation and release program output.

In case of divergence, we abort all epochs that started after the divergence and initiate rollback. In Section 3.3.4, we discuss two different implementations for rolling back execution state. The first is a simpler design that rolls both executions back to the start of the epoch that diverged and restarts execution. The second is more complicated, but guarantees forward progress; it rolls both executions back to the state of the epoch-parallel execution at the divergence and restarts both executions from that state.

At any subsequent time, DoublePlay can replay a recorded execution by (1) restoring the initial state of the recorded process, (2) replaying it on a single core using the logged system calls, signals, and synchronization operations, and (3) using the same schedule of thread execution that was used during the epoch-parallel execution.

3.3 Implementation

DoublePlay uses Respec (see Chapter 2 during recording to coordinate the thread-parallel and epoch-parallel executions. From the point of view of Respec, the thread-parallel execution is the original execution and the epoch-parallel execution is the cloned execution. DoublePlay makes several enhancements to the basic Respec in-

frastructure in order to support offline replay, which we describe in the following sections.

3.3.1 Enabling concurrent epoch execution

DoublePlay needs to run multiple epochs concurrently, while Respec runs only a single epoch at a time. DoublePlay therefore makes multiple copies of the thread-parallel execution by calling the multi-threaded fork primitive before starting the execution of each individual epoch. This primitive creates a new process whose state is identical to that of the thread-parallel execution at that point in its execution. Each time a new process is created, DoublePlay does not let it begin execution, but instead places it in an *epoch queue* ordered by process creation time.

The DoublePlay scheduler is responsible for deciding when and where each process will run. Currently, the scheduler uses a simple policy that reserves half of the available cores for the thread-parallel execution and half for the epoch-parallel execution. It uses the Linux *sched_setaffinity* system call to constrain process execution to specific cores. As long as cores remain available, the scheduler pulls the next process from the epoch queue, allocates a core to it, constrains it to execute on only that core, and wakes up the process. When the process completes executing the epoch, it informs the scheduler that the core is now free, and the scheduler allocates the core to the next process in the epoch queue.

Even though DoublePlay starts execution of the epochs in the epoch-parallel execution in sequential order, there is no guarantee that epoch execution will *finish* in order, since some epochs are much shorter than others. DoublePlay uses an adaptive algorithm to vary epoch lengths. It sets the epoch length to 50 ms after a rollback. Each epoch without a rollback increases the epoch length by 50 ms up to a maximum of one second. Further, by leveraging output-triggered commits [63], DoublePlay ends an epoch immediately if a system call requires synchronous external output and no later than 50 ms after asynchronous external output. Network applications have much external output, so they have many short epochs even with few rollbacks.

Even though epochs may finish out-of-order, DoublePlay ensures that they commit or rollback in sequential order. When a epoch completes execution, DoublePlay performs a divergence check by comparing its memory and register state to that of the checkpoint associated with the next epoch. Once this check passes, DoublePlay allows the process to exit and allocates its processor to another epoch. If all prior epochs have been committed, DoublePlay also commits the epoch and discards the checkpoint for that epoch. Otherwise, it simply marks the epoch as completed. After all prior epochs commit, DoublePlay will commit that epoch.

In its strictest form of verification, DoublePlay considers a divergence check to fail if (1) a thread in the epoch-parallel execution calls a different system call from the one called by the corresponding thread in the thread-parallel execution, (2) the two threads call different `libc` synchronization operations, (3) the two threads call the same system call or synchronization operation, but with different arguments, or (4) the registers or memory state of the two executions differ at the end of the epoch. In Section 3.3.5, we describe how we loosen these restrictions slightly to reduce unnecessary rollbacks.

When a divergence check fails, DoublePlay terminates all threads executing the current epoch and any future epochs in the epoch-parallel execution, as well as all threads of the thread-parallel execution, by sending them kill signals. However, epochs started prior to the one that failed the divergence check may still be executing for the epoch-parallel execution. DoublePlay allows these epochs to finish and complete their divergence checks. If these checks succeed, DoublePlay restarts the thread-parallel execution from the failed epoch. If a check for one of the prior epochs fails, it restarts execution from the earliest epoch that failed.

3.3.2 Replaying thread schedules

DoublePlay guarantees deterministic offline replay by executing each epoch on a single core using the same thread schedule that was used during recording by the epoch-parallel execution. There are two basic strategies for providing this property.

One strategy is to use the same *deterministic scheduler* during epoch-parallel execution and offline replay. The second strategy is to *log the scheduling decisions* made during epoch-parallel execution and replay those decisions deterministically during offline replay.

To implement these strategies, we added a custom scheduler layer that chooses exactly one thread at a time to be run by the Linux scheduler and blocks all other threads on a wait queue. When the DoublePlay scheduler decides to execute a new thread, it blocks the previously-executing thread on the wait queue and unblocks only the thread that it chooses to run. It would also have been possible to implement these strategies by modifying the Linux scheduler, but this would have required instrumenting all scheduling decisions made in Linux.

To implement the first strategy (the deterministic scheduler), DoublePlay assigns a strict priority to each thread based on the order in which the threads are created. DoublePlay always chooses to run the highest-priority thread eligible to run that would preserve the total ordering of system calls and the partial ordering of synchronization operations.

To preserve system call ordering, the thread-parallel execution assigns a sequence number to every system call entry or exit when it is added to the circular buffer. A thread in the epoch-parallel execution only consumes an entry if it has a sequence number one greater than the entry last consumed. Thus, a high-priority thread whose next entry is several sequence numbers in the future must block until low-priority threads consume the intervening entries. Similarly, DoublePlay uses multiple sequence numbers to represent the partial order of user-level synchronization operations. Specifically, the address of each lock or futex accessed in a synchronization operation is hashed to one of 512 separate counters, each of which represents a separate sequence number. Note that once a low-priority thread consumes an entry in either buffer, a higher-priority thread may become eligible to run. In this instance, DoublePlay immediately blocks the low-priority thread and unblocks the high-priority thread. For any given set of system calls and synchronization operations, this algorithm produces a deterministic schedule. That is, context switches always occur at

the same point in each thread's execution.

This first strategy is relatively easy to implement and requires no additional logging. However, it is difficult to allow preemptions in this strategy, because doing so would require inserting preemptions deterministically [64]. Allowing preemptions has two benefits: it allows the uniparallel execution to reproduce any bug that manifests on sequentially-consistent hardware, and it maintains liveness in the presence of spin locks.

To allow preemptions, we implemented a second strategy for replaying thread schedules deterministically, which is to log the preemptions that occur during epoch-parallel execution and replay those decisions deterministically during offline replay. In order to deterministically reproduce preemptions, we record the instruction pointer and branch count of a thread when it is preempted during epoch-parallel execution. The branch count is necessary because the instruction could be inside a loop, and we must replay the preemption on the correct iteration [54]. These branch counts are maintained per thread and obtained using hardware performance monitoring counters configured to count branches executed in user-space [15]. We compensate for return from interrupt (`iret`) branches, which would otherwise cause interrupts to perturb the branch count non-deterministically. After recording the instruction pointer and branch count, we unblock the next thread. In offline replay, we preempt a thread when it reaches the recorded instruction pointer and branch count and allow the next thread to run.

All deterministic replay systems perturb the execution that is being recorded, and DoublePlay is no exception. Most deterministic replay systems perturb the execution by significantly slowing events that may represent interprocessor communication (such as shared memory accesses) and thereby changing the thread interleaving relative to an unrecorded execution. DoublePlay avoids this type of perturbation; instead, it perturbs the execution by timeslicing threads onto a single processor and controlling preemptions. Being able to control and limit preemptions makes it easy to replay the epoch-parallel execution; it also makes possible new strategies for detecting and avoiding races [90]. However, DoublePlay's execution strategy makes executions with

numerous, fine-grained interleaving impossible without injecting numerous preemptions. Thus, while DoublePlay with preemptive scheduling is *able* to produce any execution that is possible with an unlogged, thread-parallel run [1], a particular execution may be more or less *likely* in DoublePlay than in an unlogged, thread-parallel run. However, this does not limit applications of DoublePlay in production systems where reproducibility is the required guarantee. When using deterministic replay for debugging, DoublePlay and other deterministic replay systems change the likelihood of encountering particular bugs. Users who want to systematically explore the entire space of legal thread interleavings may want to combine deterministic replay with a system for controlling preemptions, such as CHESS [56]. In fact, to orchestrate a thread interleaving, CHESS executes each test run on a uniprocessor, which can be accelerated by using DoublePlay's thread-parallel execution.

3.3.3 Offline replay

To support offline replay, DoublePlay records the system calls and synchronization operations executed during an epoch in a set of log files (for simplicity, DoublePlay uses a separate log for each thread). After committing each epoch, DoublePlay marks the entries belonging to that epoch as eligible to be written to disk. It then writes the marked records out asynchronously while other epochs are executing. Note that DoublePlay only has to record the results of synchronization operations and system calls, since the arguments to those calls will be deterministically reproduced by any offline replay process. This reduces log size considerably for system calls such as `write`. Signals are logged with the system calls after which they are delivered.

If a rollback occurs, DoublePlay deallocates any records in the circular buffer that occurred after the point in the thread-parallel execution to which it is rolling back (these records cannot have been written to disk since the epoch has not yet committed). It also ensures that all entries that precede the rollback point are written

[1] More precisely, DoublePlay with preemptive schedule can produce any execution that is possible with an unlogged, thread-parallel run *on a sequentially consistent memory system.* Executions that require a weaker consistency model between processors cannot occur in DoublePlay, since the epoch-parallel execution runs all threads for an epoch on a single processor.

to disk before restoring the checkpoint. The checkpoint includes the sequence numbers at the point in execution where the checkpoint was taken, so subsequent entries will have the correct sequence number. Thus, there is no indication of the divergence or the rollback in the logs on disk.

DoublePlay saves the initial state of the process when recording began. To perform an offline replay, it starts from this initial copy. DoublePlay currently runs the offline replay on a single processor and uses the scheduling algorithm described in Section 3.3.2 to constrain the order of thread execution for the offline replay process. When an offline replay thread executes a system call or synchronization operation, DoublePlay returns the results recorded in its log file. The only system calls that DoublePlay actually executes are ones that modify the process address space such as `mmap` and `clone`. DoublePlay delivers recorded signals at the same point in process execution that they were delivered during recording.

3.3.4 Forward recovery

We implemented two different rollback strategies in DoublePlay. Initially, we decided to roll both executions back to the checkpoint at the beginning of the epoch that failed the divergence check. Both executions would restart from this point. If the divergence check again failed, we would roll back and try again. However, we saw some executions in which a given epoch would roll back several times in a row before the divergence check succeeded, presumably because it contained one or more frequently diverging data races. Frequent rollbacks imposed a substantial performance overhead for some applications; in the worst case, a program with many frequent races could potentially even fail to make forward progress.

After consideration, we realized that this strategy reflected the incorrect view that the thread-parallel execution was the run being recorded. In fact, the epoch-parallel execution is the "real" execution being recorded — it is after all the one that is being executed on a single core with a known thread schedule. The epoch-parallel execution is also a perfectly legal execution that could have occurred on the thread-

parallel execution with a particular set of relative speeds among the processors (since we verify the ending state of the prior epoch matches the starting state of the next epoch). The thread-parallel execution exists merely as a means for generating hints about future process state so that multiple epochs can be executed in parallel.

Once we viewed the epoch-parallel execution as the one being recorded, it was clear that the state of its execution at the time the divergence check fails is a valid execution state that can be deterministically reproduced offline. Therefore, DoublePlay can use the epoch-parallel process state at the time the divergence check fails as a checkpoint from which to restart execution. We call this process *forward recovery*.

A complication arises because the kernel state is associated only with the thread-parallel execution (because it is the process that actually executes all system calls), while the correct process state is associated with the epoch-parallel execution. DoublePlay detects when the thread-parallel and epoch-parallel executions diverge by comparing the order and arguments of system calls, and the memory and registers after each epoch. At the point of divergence, the (logical) kernel states of the two executions are guaranteed to be identical because the executions have issued the same sequence of system calls up to that point. Because the logical kernel states are identical, it is correct to merge the kernel state of the thread-parallel execution with the memory and register state of the epoch-parallel execution.

DoublePlay logs an undo operation for each system call that modifies kernel state, so that forward (and regular) recovery can roll back kernel state by applying the undo operations. It can thus roll back the thread-parallel execution's kernel state to the system call at which a divergence was detected. It then makes the contents of the address space of the thread-parallel execution equal to the contents of the address space of the epoch-parallel execution at the point where the divergence check failed.

For instance, consider a process with a data race that causes the memory states of the two executions to differ at the end of an epoch. DoublePlay's divergence check already compares the address space of the epoch-parallel execution with a checkpoint of the thread-parallel execution. If any bytes differ, DoublePlay simply copies the corresponding bytes from the epoch-parallel address space to the checkpoint's. Dou-

blePlay then restores the checkpoint to restart the thread-parallel execution; new epoch-parallel executions will be spawned as it proceeds.

Forward recovery guarantees that DoublePlay makes forward progress, even when a divergence check fails. All work done by the epoch-parallel execution up until the divergence check, including at least one data race (the one that triggered the divergence) will be preserved. In the worst case, every epoch may contain frequent data races, and divergence checks might fail. However, even in this case, DoublePlay should be able to approach the speed of uniprocessor replay since a single core can always make progress. In the expected scenario where data races are relatively infrequent, DoublePlay runs much faster.

3.3.5 Looser divergence checks

In our initial design for DoublePlay, we made the divergence check very strict because we wanted to detect a divergence as soon as possible in order to minimize the amount of work thrown away on a rollback. However, once we implemented forward recovery, we decided that strict divergence checks might no longer be best. As long as the epoch-parallel execution can continue its execution, any work that it completes will be preserved after a rollback.

Even with forward recovery, it is still necessary to check memory and register state at each epoch boundary. If the process state at the end of an epoch of the epoch-parallel execution does not match the state from which the next epoch execution starts, the epoch-parallel executions of all subsequent epochs are invalid and must be squashed. However, strict divergence checks are not necessarily required *within* an epoch.

As long as the external output of the epoch-parallel execution continues to match the external output of the thread-parallel execution, then the system calls and synchronization operations performed by the two executions can be allowed to diverge slightly. On the other hand, if one of these operations can affect state external to the process, then the state changes cannot be made visible to an external entity

and all subsequent results from system calls and synchronization operations must be consistent with the divergence.

Based on this observation, we modified DoublePlay to support three slightly looser forms of divergence checks within an epoch. First, if a thread's circular buffer contains a system call with no effect outside that thread, but the epoch-parallel execution omits that system call, we allow it to skip over the extra record in the buffer. Examples of such system calls are `getpid` and `nanosleep`.

Second, if a thread executes a system call that produces output for an external device such as the screen or network, the epoch-parallel execution is allowed to execute the same system call with different output. Because DoublePlay buffers such output in the kernel until after both executions have completed the system call, no external entity observes this output prior to the execution of the call by the epoch-parallel execution. Because that execution is logically the "real" execution, we simply release its output, rather than the output from the thread-parallel execution, to external observers. The observers therefore see the same output during recording and offline replay. The output produced by the thread-parallel execution can be viewed as an incorrect hint that is later corrected.

Third, if a thread executes a set of self-canceling synchronization operations or system calls during the thread-parallel execution, but omits them in the epoch-parallel execution, then all members of the set can be skipped. By self-canceling, we mean two or more operations that when executed together have no effect on state external to the thread. For example, lock and unlock operations for the same low-level lock are a self-canceling pair. If the thread-parallel execution performed both operations, then the epoch-parallel execution can achieve the same effect by performing neither.

Looser divergence checks have two benefits. First, the epoch parallel execution can run longer before a rollback is needed. Second, a rollback can sometimes be avoided all together when the divergence in process state is transitory. For example, one application we tested (pbzip2) has a benign race in which one thread spins waiting for another to set a value. Since the thread calls `nanosleep` periodically, the spin loop executes different numbers of iterations in different executions. With strict

divergence checks, such behavior leads to rollbacks. However, with looser divergence checks, the epoch-parallel execution continues past this loop. While different calls to `nanosleep` leave temporary differences on the thread's stack, these differences are soon overwritten by subsequent system calls. Thus, unless the epoch boundary happens immediately after the spin loop, the divergence in program state heals and no rollback is needed.

3.4 Evaluation

Our evaluation answers the following questions:

- What is the overhead of DoublePlay record and replay for common applications and benchmarks?
- How often do applications roll back, and what is the effect of rollback on replay time?
- Do our optimizations, namely forward recovery and loose replay, reduce overhead for applications with data races?

3.4.1 Methodology

We used two different 8-core computers to parallelize our evaluation. The first has a 2 GHz 8-core Xeon processor with 3 GB of RAM, while the second has a 2.66 GHz 8-core Xeon with 4 GB of RAM. Both computers run CentOS Linux version 5.3. The kernel is a stock Linux 2.6.26 kernel, modified to include DoublePlay. We also modified the GNU glibc library version 2.5.1 to support DoublePlay. We use our first strategy for replaying thread schedules, i.e. the deterministic scheduler that runs threads according to a strict priority.

We evaluated DoublePlay with 9 benchmarks: five parallel applications (pfscan, pbzip2, aget, the Apache web server, and the mysql database server) and 4 SPLASH-2 [99] benchmarks (fft, radix, ocean, and water). We report three values for each

experiment: the original execution time of the application running on a stock system, the execution time during DoublePlay recording, and the execution time for DoublePlay offline replay. The record time measures the time for both the thread-parallel and epoch-parallel executions to finish.

DoublePlay periodically writes out the kernel and user-level replay logs to a file to disk so they can be used for offline replay. We report the size of the log and include the time taken to write the log out to disk in our recording cost.

We evaluate the benchmarks in the following manner. We use pbzip2 to compress a 311 MB log file in parallel. We use pfscan to search in parallel for a string in a directory with 952 MB of log files. We extended the benchmark to perform 100 iterations of the search so that we could measure the overhead of deterministic replay over a longer run while ensuring that data is in the file cache (otherwise, our benchmark would be disk-bound). Aget is a bandwidth-stealing application that takes advantage of I/O parallelism by opening multiple connections to a remote server. We used aget to retrieve a 21 MB file over a local network from a server configured to limit each connection to a maximum download bit rate of 1MB/sec. We tested Apache using ab (Apache Bench) to simultaneously send 100 requests from eight concurrent clients over a local network. We evaluate mysql using sysbench version 0.4.12. This benchmark generates 3000 total database queries on a 9 GB myISAM database; 2000 queries are read-only, 600 are updates, and 400 are other types such as table lock and unlock. Since mysql uses a separate worker thread to handle each client, we vary the number of concurrent clients depending on the number of worker threads we are evaluating in each trial.

For each benchmark, we vary the number of worker threads from one to eight for the original execution. DoublePlay uses two cores per worker thread, so we measure its performance with up to four worker threads. Many benchmarks 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 five trials.

3.4.2 Record and replay performance

Several factors may lead to performance overhead in DoublePlay. The dominant potential source of overhead is DoublePlay's use of two replicas, which will increase runtime if there are insufficient spare processors to absorb this increased utilization. In addition to processor utilization, the extra replica may contend for cache or memory bandwidth. Other sources of overhead include the need to log and replay synchronization operations and system calls, wait for epochs to finish on all processors, compare memory pages, and rollback in case of data races. DoublePlay also increases the amount of memory used by the application. Each processor used in the epoch-parallel execution operates on a copy-on-write replica of the address space. Thus, the average instantaneous memory overhead of DoublePlay is the number of pages written to during an epoch multiplied by the number of epochs running in parallel.

Table 3.1 shows the overall performance results for DoublePlay. The first three columns show the application or benchmark executed, the number of worker threads used, and the execution time of the application without DoublePlay. The next seven columns give statistics about DoublePlay execution: the number of user-level synchronization operations logged, system calls logged, epochs executed, memory pages compared, size of the DoublePlay logs written to disk, the average number of rollbacks that occurred per execution, and the time to record an execution. The next column shows the overhead added by DoublePlay during recording, compared to two configurations of the original application. The first overhead is relative to the original application with the same number of application threads; this shows the overhead of DoublePlay when it can take advantage of unused cores. The second overhead is relative to the original application with twice as many application threads; this shows the overhead of DoublePlay relative to an application configured to use the same number of cores as DoublePlay. The last column shows the offline replay execution time.

The availability of unused cores significantly impacts the overhead added by DoublePlay during recording. If there are sufficient unused cores, DoublePlay adds little overhead to the recorded application execution time.

app	worker threads	original time (s) & stdev.	synch. ops.	system calls	epochs	pages compared	log size (MB)	average rollbacks per run	record time (s) & stdev.	recording overhead	offline time (s) & stdev.
pfscan	1	193.93 (0.11)									
	2	100.99 (0.89)	23302	7528	101	1325	1.60	0	108.38 (0.87)	7% / 105%	193.73 (0.26)
	3	69.01 (0.26)	22250	7531	60	868	1.55	0	78.91 (1.01)	14% / 112%	190.23 (2.88)
	4	52.97 (0.18)	21575	7534	50	760	1.52	0	59.20 (0.92)	12% / 102%	188.19 (1.16)
	6	37.18 (0.04)									
	8	29.26 (0.03)									
pbzip2	1	91.65 (0.06)									
	2	46.08 (0.03)	26182	5481	46	573243	1.30	0	50.42 (0.41)	9% / 111%	92.88 (0.11)
	3	31.45 (0.77)	26025	5207	31	574131	1.28	0	35.92 (0.09)	14% / 119%	92.94 (0.29)
	4	23.94 (0.38)	26393	5066	23	562705	1.29	0	29.38 (0.43)	23% / 128%	92.85 (0.10)
	6	16.38 (0.07)									
	8	12.90 (0.03)									
aget	1	21.19 (0.04)									
	2	10.73 (0.02)	25243	33495	22	267	27.41	0.4	10.80 (0.06)	1% / 99%	0.28 (0.01)
	3	7.22 (0.01)	21564	29022	20	263	26.54	0.2	7.31 (0.08)	1% / 102%	0.26 (0.01)
	4	5.42 (0.01)	19618	27372	20	277	26.13	0.2	5.54 (0.15)	2% / 104%	0.25 (0.02)
	6	3.62 (0.01)									
	8	2.71 (0.00)									
apache	1	44.36 (0.13)									
	2	43.59 (0.27)	3756	3944	393	5264	0.14	0.1	43.95 (0.33)	1% / 10%	0.04 (0.00)
	3	41.67 (0.35)	3682	3923	386	5285	0.14	1.0	42.74 (0.59)	3% / 11%	0.04 (0.00)
	4	40.13 (0.27)	3636	3904	389	5383	0.14	1.7	40.75 (0.65)	2% / 9%	0.04 (0.00)
	6	38.39 (0.41)									
	8	37.35 (0.61)									
mysql	1	29.97 (0.08)									
	2	29.25 (0.09)	195903	46691	3035	195903	13.86	0	34.89 (0.23)	19% / 19%	0.53 (0.00)
	3	29.28 (0.08)	199952	46792	3052	199952	14.07	0	34.98 (0.21)	19% / 19%	0.54 (0.00)
	4	29.20 (0.08)	200598	46951	3069	200598	14.10	0.2	34.25 (1.39)	17% / 17%	0.55 (0.00)
	6	29.33 (0.12)									
	8	29.35 (0.14)									
fft	1	117.88 (0.19)									
	2	58.12 (0.06)	15689	3011	67	547619	0.90	0	68.72 (0.25)	18% / 106%	115.65 (0.11)
	4	33.33 (1.08)	32242	3041	41	333837	1.77	0	43.47 (0.17)	30% / 131%	106.61 (0.15)
	8	18.79 (0.15)									
radix	1	177.84 (0.96)									
	2	89.10 (0.39)	4571	471	41	1295817	0.22	0	96.88 (0.15)	9% / 114%	177.58 (0.23)
	4	45.28 (0.28)	11140	607	41	1313664	0.53	0	53.43 (0.23)	18% / 127%	177.73 (0.24)
	8	23.50 (0.03)									
ocean	1	56.67 (0.03)									
	2	28.19 (0.67)	108808	149	32	914104	4.88	0	46.09 (0.06)	63% / 222%	56.45 (0.03)
	4	14.31 (1.35)	218788	222	27	745697	10.44	0	31.75 (0.26)	121% / 278%	53.24 (0.25)
	8	8.39 (0.09)									
water	1	154.57 (1.97)									
	2	81.70 (1.95)	5376008	21404	88	275484	207.33	0	89.00 (3.78)	9% / 106%	160.57 (3.63)
	3	57.78 (1.25)	6756393	21741	65	279701	260.64	0	63.10 (1.45)	9% / 89%	160.60 (2.13)
	4	43.15 (0.03)	8131526	21537	54	334339	313.81	0	56.32 (1.96)	31% / 92%	162.99 (1.05)
	6	33.39 (0.52)									
	8	29.33 (0.22)									

Results are the mean of five trials. Values in parentheses show standard deviations. Note that DoublePlay uses more cores than the original execution during recording since it executes two copies of the application. The overhead column shows the overhead of DoublePlay with respect to two configurations of the original application. The first overhead is relative to the original application with the same number of application threads; this shows the overhead of DoublePlay when it can take advantage of unused cores. The second overhead is relative to the original application with twice as many application threads; this shows the overhead of DoublePlay relative to an application configured to use the same number of cores as DoublePlay.

Table 3.1: DoublePlay performance

On average, 2 worker threads add about 15% overhead to the application run time. The overhead gradually increases to 28% with 4 threads. For the five real applications, the maximum overhead of any benchmark is 23% (for pbzip2 with 4 worker threads) — the average overhead is only 7% with 2 worker threads and 11% with 4 worker threads. Apache and aget are limited by the speed of our local network; mysql is limited by disk I/O, and the remaining benchmarks are CPU bound. DoublePlay shows more overhead for the SPLASH-2 benchmarks, generally because they perform many more synchronization operations per second and dirty memory pages more rapidly, which increases the number of pages compared. As shown in the Respec paper [46], Ocean is a challenging benchmark as most of its overhead derives from sharing the limited memory bandwidth and processor caches between the recorded and online replay processes. As expected, Ocean incurs approximately the same overhead with DoublePlay as it incurs with Respec.

If all cores can be productively used by the application, then DoublePlay incurs much higher overhead because it uses twice as many cores, executing each benchmark twice during recording. When compared to an application configured to use the same number of cores as DoublePlay, DoublePlay adds approximately 100% overhead to CPU-bound applications that can scale to eight cores (this does not affect Apache, which is network bound, or mysql, which is disk bound). For comparison, iDNA adds an average of 1100% overhead [11], and SMP-ReVirt adds 10-600% overhead [25]. SMP-ReVirt does not incur DoublePlay's overhead of executing the application twice, but SMP-ReVirt does incur high overhead for applications that share data frequently (including false sharing due to tracking ownership at page granularity) because of the cost of memory protection faults. DoublePlay also scales well up to 8 cores (e.g., its average overhead across all benchmarks without spare cores is 99% with 2 threads and 110% with 4 threads), while SMP-ReVirt does not scale well for some applications even up to 4 cores.

Thus, DoublePlay is well suited for three settings that require deterministic replay (e.g., for forensic or auditing purposes): (1) applications which cannot scale effectively to use all cores on the machine, (2) sites that are willing to dedicate extra cores to

provide deterministic replay, and (3) applications that share data frequently between cores (for which DoublePlay is the lowest overhead solution). In particular, we believe that many machines will have unused cores that DoublePlay can take advantage of because the number of cores per computer is expected to grow exponentially, and scaling applications to use these extra cores is notoriously difficult. In such settings with spare cores, DoublePlay incurs only modest overhead.

For CPU-bound benchmarks, DoublePlay's offline replay takes approximately the same amount of time as a single-threaded execution of the application. This is due to our current implementation, which limits offline replay to executing on one core. If we used two executions to accelerate replay in the same way we accelerate recording, the offline replay time should be approximately the same as the record time for these benchmarks. Aget runs much faster during offline replay than during recording because it obtains its data from sequential disk reads rather than from network receives. Apache runs even faster because it uses Linux's zero-copy `sendfile` system call: thus, no data is copied into or out of its address space on replay. Mysql also benefits from sequential disk I/O.

DoublePlay's offline replay performance is a substantial improvement over system such as PRES and ODR that log partial information during recording, then search during replay for an execution that matches the original execution. In its low-recording overhead mode (SI-DRI), ODR's replay time is reported as ranging from 300 to over 39000 times the original application time, with some replays not completing at all. During replay, PRES records the global order of accesses to shared variables from different threads (called the RW scheme) to guide its search for an execution that matches the recorded run. This is reported to have an overhead from 28% (for network applications to several hundred times (for CPU-bound applications and benchmarks) the original execution time. This additional work is necessary to guarantee that the produced replay can be reproduced at will. Further, PRES may try several executions before finding a matching one. When PRES records synchronization operations and system calls (similar to DoublePlay), it takes 1-28 tries to replay most runs, and is unable to replay one run within 1000 tries. Thus, Double-

app	threads	rollbacks w/o opt.	rollbacks with opt.	execution time(s) & stdev. w/o opt.	execution time(s) & stdev. with opt.	relative reduction in exec. time
pbzip2	2	2.2	0	53.32 (2.61)	50.42 (0.41)	6%
	3	1.2	0	38.54 (3.77)	35.92 (0.09)	8%
	4	0.8	0	32.59 (3.04)	29.38 (0.43)	13%
aget	2	0.4	0.4	10.80 (0.06)	10.84 (0.12)	0%
	3	0.2	0.2	7.31 (0.08)	7.30 (0.02)	0%
	4	0.2	0.2	5.54 (0.15)	5.57 (0.03)	0%
apache	2	0.1	0.1	43.95 (0.33)	43.81 (0.31)	0%
	3	1.0	1.0	42.74 (0.59)	41.78 (0.48)	2%
	4	1.7	1.7	40.75 (0.65)	41.09 (1.28)	-1%
mysql	4	0.2	0.2	34.25 (1.39)	34.25 (1.39)	0%

Table 3.2: Benefit of forward recovery and loose replay.

Play's contribution compared to these prior systems is (1) to guarantee that a replay can be produced in a bounded amount of time, and (2) to substantially lower relative replay time. The main cost of these contributions is that DoublePlay uses twice as many cores during recording to run two executions of CPU-bound applications.

3.4.3 Forward recovery and loose replay

Of the nine benchmarks we used in our evaluation, four (pbzip, aget, Apache, and mysql) experience application-level benign data races. For instance, pbzip2 has a benign data race in which an output thread spins waiting for a worker thread to set a value. Aget has a benign race where a thread reads and displays a progress counter without grabbing a lock. Apache has a benign race in which worker threads increment an idle counter in a spin loop with an atomic compare-and-swap implemented using low-level locks. If two worker threads contend, one may experience an additional iteration of the spin loop, which leads to an additional paired lock-unlock sequence.

As shown in Table 3.2, DoublePlay's loose replay optimization reduces the frequency of rollbacks for pbzip2 — in fact, all rollbacks were eliminated during our evaluation. Eliminating rollbacks improves execution time from 6% of the original execution time with two threads to 13% with four threads. In all runs during our evaluation, the loose replay optimization was able to continue the execution of the

epoch-parallel execution past the spin loop. While extra system calls create a transient difference in stack values, the process states soon converge when the thread overwrites those values during subsequent function calls.

For aget and Apache, no rollbacks were avoided. Although loose replay allows epoch-parallel execution to proceed when aget outputs a different progress value to the console, the output values usually remain in a buffer in `libc` at the end of the epoch, which is detected as a divergence. The data races in Apache lead to complicated divergences in synchronization operations and system calls for which loose replay cannot be used. Further, for these two applications, forward recovery does not have a measurable performance impact. For aget, the reason is that DoublePlay saves a copy of data received over the network until the epoch that received the data is committed (otherwise, the received data would be lost). On a rollback, the subsequent execution reads the copied data from memory rather than the network, so those system calls are much faster. Apache also benefits from this behavior, but it also takes frequent epochs because it sends external output over the network quite often. Consequently, the amount of work preserved by a forward recovery is very small. Since divergence check failures are relatively infrequent, the effect of forward recovery on the execution time of the two applications is negligible. We observed only one rollback during the mysql benchmarks; as with Apache, this rollback could not be avoided due to a complicated divergence, but the performance impact was negligible due to frequent epochs. Thus, while forward recovery and loose replay are often successful in reducing rollbacks and preserving work done during a failed epoch, they appear to be most beneficial for CPU-bound applications with longer epoch durations.

3.5 Conclusion

Providing efficient deterministic replay for multithreaded programs running on multiprocessors is challenging. While many prior solutions have been proposed, all fall short in some way. For instance, they may require custom hardware support, be prohibitively slow for many applications, or not guarantee that a replayed execution

can be produced in a reasonable amount of time. Compared to these prior systems, DoublePlay's contribution is to provide guaranteed software-only record and replay with a minimal overhead to execution time, at the cost of using more cores. The key insight in DoublePlay is that one can use the simpler and faster mechanisms of single-processor record and replay, yet still achieve the scalability offered by multiple cores, by using an additional execution to parallelize the record and replay of an application. On machines with spare cores, this insight allows DoublePlay to record application execution with only an average of 15% overhead with 2 threads and 28% with 4 threads.

CHAPTER 4

Detecting and surviving data races using complementary schedules

This chapter describes how uniparallel execution can be used to build a system that can detect data races in multithreaded programs running on commodity multiprocessors and also mask the harmful effects of most data race bugs and ensure the forward progress of production software systems.

4.1 Introduction

Developing reliable multithreaded programs that run reliably on commodity multiprocessors is a challenging goal. One fundamental problem that exacerbates this challenge is the occurrence of bugs due to data races. A data race occurs when two threads concurrently access the same memory location and one of them performs a write, without being ordered by synchronization operations. As most data race bugs result from rare thread interleavings, they are difficult to identify during development and can cause crashes, data loss and other program errors at runtime.

To help address the problem of data race bugs, we propose running multiple replicas of a program using uniparallelism and forcing two of these replicas to follow *complementary schedules*. Complementary schedules are a set of replica thread schedules crafted to ensure that replicas diverge only if a data race occurs and to make it very likely that harmful data races cause divergences. First, we use uniparallelism

to split a program into epochs and run each epoch twice as distinct epoch-parallel replicas. Next, we run a single thread-parallel execution that provides a log of non-deterministic events (e.g., system calls) and a log of the happens-before ordering of synchronization operations. The two epoch-parallel executions of an epoch are identical to the thread-parallel execution in that all three replicas operate on the same program input. Additionally, the epoch-parallel replays are constrained to a single processor and observe the same happens-before ordering and non-deterministic events logged by the thread-parallel execution.

The only difference in the epoch-parallel replicas is that each executes a unique complementary schedule. Specifically, the current thread in each epoch-parallel replica runs non-preemptively until it blocks on a synchronization operation or system call; the next thread to run on the processor is carefully selected from the set of eligible threads with the goal that the epoch-parallel replicas execute two instructions in different threads that are not ordered by a synchronization operation or system call, in opposite orders. By having two replicas execute racing instructions in opposite order, we hope to cause at least one replica to trigger the data race bug while the other replica avoids the harmful ordering.

We realize our design in Frost, a new system that combines uniparallelism and complementary schedules to achieve two goals: detecting data races at low overhead and increasing availability by masking the effects of harmful data races at runtime. Frost introduces a new method to detect races: *outcome-based data-race detection*. While traditional dynamic data race detectors work by analyzing the events executed by a program, outcome-based race detection works by detecting the *effects* of a data race by comparing the states of different replicas executed with complementary schedules. Outcome-based race detection achieves lower overhead than traditional dynamic data race detectors, but it can fail to detect some races, e.g., data races that require a preemption to cause a failure and that also generate identical correct outcomes using multiple non-preemptive schedules (see Section 4.3.4 for a full discussion). However, in our evaluation of real programs, Frost detects all potentially harmful data races detected by a traditional data race detector. While prior work [59] compared the

outcomes of multiple orderings of instructions for *known* data races in order to classify those races as either benign or potentially malign, Frost is the first system to construct multiple schedules specifically to detect and survive *unknown* data races. Frost thus faces the additional challenge of constructing useful schedules without first knowing which instructions race. A benefit that Frost inherits from the prior classification work is that it automatically filters out most benign races that are reported by a traditional dynamic race detector.

For production systems, Frost moves beyond detection to also diagnose and survive harmful data races. Since a concurrency bug due to a data race manifests only under a specific order of racing memory accesses, executing complementary schedules makes it extremely likely that one of the replicas survives the ill effects of a data race. Thus, once Frost detects a data race, it analyzes the outcomes of the various replicas and chooses a strategy that is likely to mask the failure, such as identifying and resuming execution from the correct replica or creating additional replicas to help identify a correct replica.

Note that Frost's approach of constraining epoch-parallel replicas to run threads on a single processor without preemptions has an implicit benefit: it prevents bugs that require preemptions (e.g., atomicity violations) from manifesting, thereby increasing availability. However, since running all threads on a single processor prevents a replica from scaling to take advantage of multiple cores, Frost also uses the thread-parallel execution to generate checkpoints of future states at each epoch boundary so multiple epochs can be run on separate cores simultaneously.

Frost helps address the problem of data races in several scenarios. During testing, it can serve as a fast dynamic data race detector that also classifies races as benign or potentially harmful in the observed execution. For a beta or production system with active users, both detection and availability are important goals. Frost masks many data race failures while providing developers with reports of data races that could lead to potential failures.

Frost makes the following contributions. First, it proposes the idea of complementary schedules, which guarantees that replicas do not diverge in the absence of

data races and makes it very likely that replicas do diverge in the presence of harmful data races. Second, it shows a practical and low-latency way to run two replicas with complementary thread schedules by using a third replica to accelerate the execution of the two complementary replicas. Third, it shows how to analyze the outcomes of the three replicas to craft a strategy for surviving data races. Fourth, it introduces a new way to detect data races that has lower overhead than traditional dynamic data race detectors.

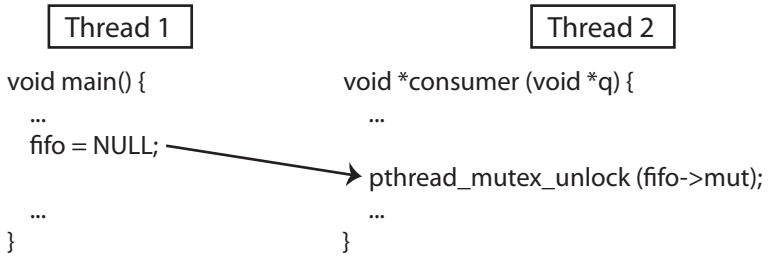
We evaluate the effectiveness of complementary thread schedules on 11 real data race bugs in desktop and server applications. Frost detects and survives all these bugs in every trial. Frost's overhead is at worst 3x utilization to run three replicas, but it has only 3–12% overhead given spare cores or idle CPU cycles to run all replicas.

4.2 Complementary schedules

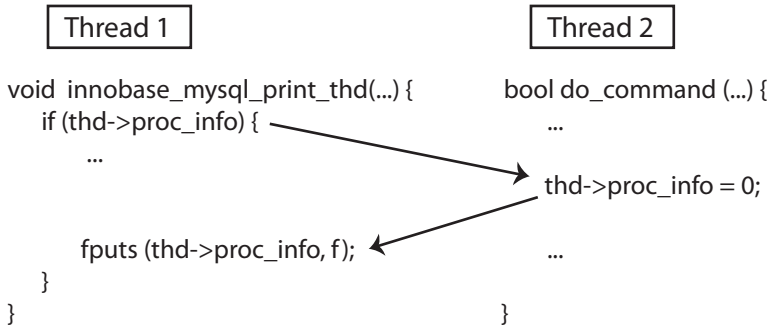
The key idea in Frost is to execute two replicas with complementary schedules in order to detect and survive data race bugs. A data race is comprised of two instructions (at least one of which is a write) that access the same data, such that the application's synchronization constraints allow the instructions to execute in either order. For harmful data races, one of these orders leads to a program error (if both orders lead to an error, then the root cause of the error is not the lack of synchronization). We say that the order of two instructions that leads to an error is a *failure requirement*. In general, a data race bug may involve multiple failure requirements, all of which must be met for the program to fail.

As an example, consider the simple bug in Figure 4.1(a). If thread 1 sets `fifo` to NULL before thread 2 dereferences the pointer, the program fails. If thread 2 accesses the pointer first, the program executes correctly. The arrow in the figure shows the failure requirement. Figure 4.1(b) shows a slightly more complex atomicity violation. This bug has two failure requirements; i.e., both data races must execute in a certain order for the failure to occur.

To explain the principles underlying the idea of complementary schedules, we first



(a) Type I bug



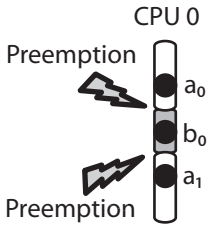
(b) MySQL #3596: Type II bug

Figure 4.1: Data race examples

consider an interval of execution in which at most one data race bug occurs and which contains no synchronization operations that induce an ordering constraint on the instructions (we call such an interval *synchronization-free*). For such regions, complementary schedules provide hard guarantees for data race detection and survival. We discuss these guarantees in this section. However, real programs do not consist solely of such regions, so in Section 4.3.4, we discuss how generalizing the range of scenarios affects Frost's guarantees.

The goal of executing two replicas with complementary schedules is to ensure that one replica avoids the data race bug. We say that two replicas have perfectly complementary schedules if and only if, for every pair of instructions a and b executed by different threads that are not ordered by application synchronization, one replica executes a before b, and the other executes b before a.

Since a failure requirement orders two such instructions, use of perfectly com-



Execution of two threads is interleaved on a single core. It is not feasible to construct a completely complementary schedule for this particular execution.

Figure 4.2: Preemption scenario

plementary schedules guarantees that for any failure requirement, one replica will execute a schedule that fulfills the requirement and one will execute a schedule that does not. This guarantees that one of the two replicas does not experience the failure.

Eliminating preemptions is essential to achieving perfectly complementary schedules. To understand why this is so, consider a canonical schedule with a preemption as shown in Figure 4.2: thread A executes an instruction a_0, then thread B preempts thread A and executes an instruction b_0, then thread A resumes and executes a_1. It is impossible to generate a perfectly complementary schedule for this schedule. In such a schedule, a_1 would precede b_0, and b_0 would precede a_0. However, this would require a_1 to precede a_0, which would violate the sequential order of executing instructions within a thread.

In contrast, without preemptions, constructing a perfectly complementary schedule for two threads in a synchronization-free interval is trivial—one schedule simply executes all of thread A's instructions before thread B's instructions; the other schedule executes all of thread B's instructions before thread A's instructions. For more than two threads, one schedule executes the threads in some order, and the other executes the threads in the reverse order.

Thus, to guarantee that at least one replica avoids a particular data race bug in a synchronization-free interval, we execute two replicas, use non-preemptive scheduling for these replicas, and reverse the scheduling order of thread execution between the two replicas.

The above algorithm provides even stronger properties for some common classes of data race bugs. For instance, consider the atomicity violation in Figure 4.1(b). Because the failure requirements point in opposite directions, each of the two replica schedules will fulfill one constraint but not the other. Since failure requires that both requirements be fulfilled, the proposed algorithm guarantees that *both* replicas avoid this failure.

In general, given n threads, we must choose an arbitrary order of those threads for one schedule and reverse that order in the complementary schedule. Visualizing the threads arrayed according to the order chosen, if all failure requirements point in the same direction, then the proposed algorithm guarantees that one replica avoids the failure. In the rest of the paper, we will refer to bugs in this category as *Type I*. If any two failure requirements point in the opposite direction, the proposed algorithm provides the stronger guarantee that both replicas avoid the failure. We will refer to bugs in this category as *Type II*.

4.3 Frost: Design and implementation

Frost uses complementary schedules to detect and survive data races. This section describes several challenges, including approximating the ideal behavior for intervals of execution with synchronization operations and multiple bugs, scaling performance via multicore execution, implementing heuristics for identifying correct and faulty replicas. It concludes with a discussion of specific scenarios in which Frost can fail to detect or survive races and the steps Frost takes to minimize the effect of those scenarios.

4.3.1 Constructing complementary schedules

Frost divides the execution of a program into time-slices called *epochs*. For each epoch, it runs multiple replicas and controls the thread schedule of each to achieve certain properties.

The first property that Frost enforces is that each replica follows the same partial

order of system calls and synchronization operations. In other words, certain pairs of events such as `lock` and `unlock` on a mutex lock, `signal` and `wait` on a condition variable, or `read` and `write` on a pipe represent a happens-before order of events in the two threads; e.g., events following the `lock` in one thread cannot occur until after the other thread calls `unlock`. By ensuring that all threads have the same happens-before order of such events, Frost guarantees that two replicas can diverge in output or final memory and register state only if a data race occurs within the epoch [73]. Further, all replicas will encounter the same pair of racing instructions.

The second property that Frost tries to achieve is that two replicas have thread schedules that are as complementary as possible, given that the first property has to be upheld. As discussed in Section 4.2, this property is intended to ensure that at least one of the two replicas with complementary thread schedules does not fail due to a particular data race.

Frost must execute a replica to observe the happens-before order of synchronization operations and system calls before it can enforce an identical order over the same operations in other replicas. Frost observes this order by using a modified glibc and Linux kernel that maintain a vector clock for each thread and for synchronization entities such as locks and condition variables. Each value in the vector represents a thread's virtual time. Synchronization events and system calls increment the calling thread's value in its local vector clock. Operations such as `unlock` set the vector clock of the lock to the maximum of its previous value and the vector clock of the unlocking thread. Operations such as `lock` set the vector clock of the locking thread to the maximum of its previous value and the vector clock of the lock. Similarly, we modified kernel entities to contain vector clocks and propagate this information on relevant system calls. For instance, since system calls such as `map` and `munmap` do not commute, Frost associates a vector clock with the address space of the process to enforce a total order over address-space-modifying system calls such as `mmap`.

When the first replica executes, Frost logs the vector clocks of all system calls and synchronizations in a log. Other replicas read the logged values and use them to follow the same happens-before order. Each replica maintains a *replay vector clock*

that is updated when a thread performs a synchronization operation or system call. A thread may not proceed with its next operation until the replay vector clock matches or exceeds the value logged for the matching operation by the first replica. This ensures, for example, that one thread does not return from `lock` until after another thread calls `unlock` if there was a happens-before order between the two operations in the original replica. More than one replica can execute an epoch concurrently; however, all other replicas typically are slightly behind the first replica since they cannot execute a synchronization operation or system call until the same operation is completed by the first replica.

Given a happens-before order, Frost uses the following algorithm to construct schedules for two replicas that complement each other as much as possible without modifying the application. Frost chooses an order over all threads within a replica and assigns the reverse order to those threads in a second replica. For example, if three threads are ordered [A, B, C] in one replica, they are ordered [C, B, A] in the other. Frost executes all threads within each replica on a single core so that two threads do not run simultaneously. A thread is eligible to run as long as it is not waiting to satisfy a happens-before constraint and it has not yet completed the current epoch. The Frost kernel always runs the eligible thread that occurs first in its replica's scheduling order. A thread runs until it reaches the end of the epoch, it blocks to enforce a happens-before constraint, or a thread earlier in the replica's scheduling order becomes eligible to run.

4.3.2 Scaling via uniparallelism

As described so far, the use of complementary schedules does not allow a program to scale to use multiple cores because all threads of a replica must run sequentially on a single core. If multiple threads from a replica were to concurrently execute two instructions on different cores, those two instructions cannot be ordered by a happens-before constraint and are thus potentially racing. In this case, the two replicas should execute these instructions in different orders. However, determining the order of these

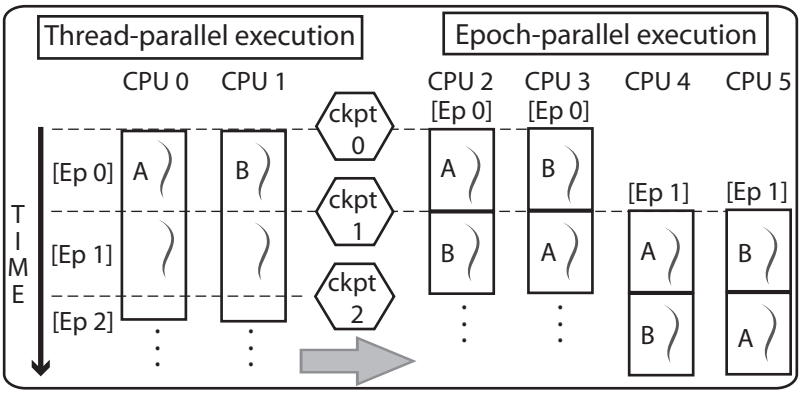


Figure 4.3: Frost: Overview

instructions and enforcing the opposite order on the other replica implies that the instructions execute sequentially, not concurrently.

Frost uses uniparallelism [91] to achieve scalability. Uniparallelism is based on the observation there exist at least two methods to scale a multithreaded program to run on multiple cores. The first method, termed *thread parallelism*, runs multiple threads on different cores — this is the traditional method for exploiting parallelism. The second method, termed *epoch parallelism*, runs multiple time-slices of the application concurrently.

Uniparallel execution runs a thread-parallel and one or more epoch-parallel executions of a program concurrently. It further constrains each epoch-parallel execution so that all its threads execute on a single core. This strategy allows the epoch-parallel execution to take advantage of the properties that come with running on a uniprocessor. Our original use of uniparallelism in a system called DoublePlay provided efficient software-only deterministic replay [91]. Frost is built on a modified version of the DoublePlay infrastructure, but it uses uniparallelism for a different purpose, namely the execution of replicas with complementary schedules and identical happens-before constraints.

As shown in Figure 4.3, to run epochs in parallel, a uniparallel execution generates checkpoints from which to start each epoch. It must generate these checkpoints early enough to start future epochs before prior ones finish. Thus, the thread-parallel

execution runs ahead of the epoch-parallel execution and generates checkpoints from which to start future epochs. Multiple epochs execute in parallel, in a manner similar to a processor pipeline — this allows an epoch-parallel execution to scale with the number of available cores.

In summary, Frost executes three replicas for each epoch: a thread-parallel replica that is used to record the happens-before constraints for the epoch and generate checkpoints to speculatively parallelize the other two replicas, and two epoch-parallel replicas with complementary schedules. Replicas use copy-on-write sharing to reduce overall memory usage. Frost uses online deterministic replay [46] to ensure that all replicas receive the same non-deterministic input and to enforce the same happens-before constraints in all replicas. It logs the result of all system calls and synchronization operations as the thread-parallel replica executes. When the epoch-parallel replicas later execute the same operations, Frost's modified kernel and glibc library do not re-execute the operations but rather return the logged values. Signals are also logged during the thread-parallel execution and delivered at the same point in the epoch-parallel execution. Because Frost logs and replays all forms of non-determinism except data races, only data races can cause replicas to diverge. Online replay has an additional performance benefit — the epoch-parallel execution does not block on I/O since the results have already been obtained by the thread-parallel execution.

When replicas diverge during an epoch, Frost chooses one of several actions. First, it may decide to accept the results of one of the replica executions, which we refer to as *committing* that replica. If it chooses to commit the thread-parallel replica, it simply discards the checkpoint taken at the beginning of the epoch. If it chooses to commit an epoch-parallel replica, and the memory and register state of that replica is different from that of the thread-parallel replica, then subsequent epochs in the pipeline are invalid. Effectively, the checkpoint from which the execution of the next epoch began was an incorrect hint about the future state of the application. Frost first discards the checkpoint taken at the beginning of the committed epoch. Then, it quashes all epochs subsequent to the one just committed and begins anew with fresh thread-parallel and epoch-parallel replicas using the state of the committed replica.

Frost may also choose to execute additional replicas to learn more about the epoch that led to the divergence. It starts additional thread-parallel and/or epoch-parallel executions from the checkpoint taken at the beginning of the epoch that led to the divergence — we refer to this process as *rolling back* the epoch. Frost could later decide to commit one of the original replicas or one of the new ones, though currently only new replicas are ever committed.

Since replicas may produce different output, Frost does not externalize any output until it decides which replica to commit. It uses speculative execution (implemented via Speculator [61]) to defer the output. This leads to a tradeoff among correctness, overhead, and output latency when choosing how long an epoch should last. Longer epochs offer better correctness properties, as discussed in Section 4.3.4.3, and also lower overhead. Shorter epochs yield lower latency for output. Frost balances these constraints by using an adaptive epoch length. For CPU-bound applications that issue no external output, checkpoint length grows up to one second. However, when the application executes a system call that produces external output, Frost immediately starts to create a new epoch. Thus, server applications we have evaluated often see the creation of hundreds of epochs per second. Additionally, as will be discussed in Section 4.3.3.1, the epoch length is varied depending on the number of data races observed during execution — epochs without a data race gradually increase the epoch length (by 50 ms at a time), while epochs with a data race decrease the epoch length (by up to a factor of 20). After Frost decides to start an epoch, it waits for all threads to reach a system call or synchronization operation. It then checkpoints the process and allows all threads to proceed.

4.3.3 Analyzing epoch outcomes

After all three replicas finish executing an epoch, the Frost kernel compares their executions to detect and survive data races. Since the Frost control code and data are in the kernel, the following logic cannot be corrupted by application-level bugs.

First, Frost determines if a replica has crashed or entered an infinite loop. We

call these *self-evident* failures, because Frost can declare such a replica to have failed without considering the results of other replicas. Frost detects if a replica crashes or aborts by interposing on the appropriate kernel signal-handling routines. It detects if a replica has entered an infinite loop via a timeout-based heuristic (we have not yet had the need to implement more sophisticated detection).

Other classes of failures are not self-evident; e.g., a replica may produce incorrect output or internal state. One way to detect this type of failure is to require a detailed specification of the correct output. Yet, for complex programs such as databases and Web servers, composing such a specification is quite daunting. Addressing this challenge in practice requires a method of detecting incorrect output that does not rely on program semantics or hand-crafted specifications.

Frost infers the potential presence of failures that are not self-evident by comparing the output and program state of the three replicas. While an epoch executes, Frost compares the sequence and arguments of the system calls produced by each replica. Frost also compares the memory and register state of all replicas at the end of epoch execution. To reduce the performance impact of comparing memory state, Frost only compares pages dirtied or newly allocated during the epoch. Frost declares two replicas to have different outcomes if either their output during the epoch or their final states at the end of the epoch differ.

To detect and survive data races, Frost must infer whether a data race has occurred and which replica(s) failed. Frost first considers whether each replica has experienced a self-evident failure. If the replica has not experienced a self-evident failure, Frost considers the memory and register state of the replica at the end of an epoch, and the output produced by the replica during that epoch.

There are 11 combinations of results among the three replicas, which are shown in the left column of Table 4.1. The result of each replica is denoted by a letter: `F` means the replica experienced a self-evident failure; `A-C` refer to a particular value for the final state and output produced by the replica for the epoch. We use the same letter, `A, B`, or `C`, for replicas that produced the same state and output. To simplify the explanation, we do not distinguish between different types of failures in

Epoch Results	Likely Bug	Survival Strategy
AAA	None	Commit A
FFF	Non-Race Bug	Rollback
AAB/ABA	Type I	Rollback
AAF/AFA	Type I	Commit A
FFA/FAF	Type I	Commit A
ABB	Type II	Commit B
ABC	Type II	Commit B or C
FAA	Type II	Commit A
FAB	Type II	Commit A or B
ABF/AFB	Multiple	Rollback
AFF	Multiple	Rollback

The left column shows the possible combination of results for three replicas; the first letter denotes the result of the thread-parallel run, and the other two letters denote the results of the epoch-parallel replicas. F denotes a self-evident failure; A, B, or C denote the result of a replica with no self-evident failure, where we use the same letter when replicas produce identical output and state.

Table 4.1: A taxonomy of epoch outcomes

this exposition. The first letter shows the result of the thread-parallel execution; the next two letters show the outcomes of the epoch-parallel executions. For example, the combination `F-AA` indicates that the thread-parallel execution experienced a self-evident failure, but the two epoch-parallel executions did not experience a self-evident failure and produced the same state and output.

As an aside, two replicas may produce the same output and reach the same final state, yet take different execution paths during the epoch due to a data race. Due to Frost's complementary scheduling algorithm, it is highly likely that the data race was benign, meaning that both replicas are correct. Allowing minor divergences during an epoch is thus a useful optimization for filtering out benign races. Frost lets an epoch-parallel replica execute a different system call (if the call does not have side effects) or

a different synchronization operation when it can supply a reasonable result for the operation. For instance, it allows an epoch-parallel replica to perform a `nanosleep` or a `getpid` system call not performed by the thread-parallel replica. It also allows self-canceling pairs of operations such as a `lock` followed by an `unlock`. While further optimizations are possible, the total number of benign races currently filtered through current optimizations is relatively small. Thus, adding more optimizations may not be worth the implementation effort. Consequently, when a divergence cannot be handled through any of the above optimizations, Frost declares the two replicas to have different output.

4.3.3.1 Using the epoch outcome for survival

Frost diagnoses results by relying on Occam's razor: it chooses the simplest explanation that could produce the observed results. Specifically, Frost chooses the explanation that requires the fewest data race bugs in an epoch. Among explanations with the same number of bugs, Frost chooses the explanation with the fewest failure requirements. The middle column in Table 4.1 shows the explanation that Frost associates with each combination of results, and the right column shows the action that Frost takes based on that explanation.

The simplest possible explanation is that the epoch was free of data race bugs. Because all replicas obey the same happens-before constraints, an epoch that is free of data races must produce the same results in all replicas, so this explanation can apply only to the combinations `A-AA` and `F-FF`. For `A-AA` epochs, Frost concludes that the epoch executed correctly on all replicas and commits it. For `F-FF` epochs, Frost concludes that the epoch failed on all replicas due to a non-race bug. In this case, Frost rolls back and retries execution from the beginning of the epoch in the hope that the failure is non-deterministic and might be avoided in a different execution, e.g., due to different happens-before constraints.

The next simplest explanation is that the epoch experienced a single Type I data race bug. A single Type I bug can produce at most two different outcomes (one for each order of the racing instructions) and the outcome of the two epoch-parallel

executions should differ because they execute the racing instructions in different order. For a Type I bug, one of these orders will not meet the failure requirement and will thereby work correctly. The other order will meet the failure requirement and may lead to a self-evident failure, incorrect state, or incorrect output. The following combinations have two different outcomes among the two epoch-parallel replicas (one of which is correct) and at most two outcomes among all three replicas: `A-AB` (and the isomorphic `A-BA`), `A-AF` (and the isomorphic `A-FA`), and `F-AF` (and the isomorphic `F-FA`).

For epochs that result in `A-AF` and `F-AF`, a replica that does not experience the self-evident failure is likely correct, so Frost commits that replica. For epochs that produce `A-AB`, it is unclear which replica is correct (or if both are correct due to a benign race), so Frost gathers additional information by executing an additional set of three replicas starting from the checkpoint at the beginning of the epoch. In this manner, Frost first tries to find a execution in which a happens-before constraint prevents the race from occurring; our hypothesis is that for a reasonably well-tested program, such an execution is likely to be correct. For the data races we have tested so far, Frost typically encounters such a constraint after one or two rollbacks. This results in a different combination of results (e.g., `A-AA`, in which case Frost can commit the epoch and proceed). If Frost encounters the same data race on every execution, we plan to use the heuristic that most natural executions are likely to be correct and have Frost choose the thread-parallel execution that occurs most often in such executions. Note that because epoch-parallel executions use artificially-perturbed schedules, they should not be given much weight; for this reason, we would not consider an `A-BA` to be two votes for `A` and one vote for `B`, but rather would consider it to be a single vote for `A`.

If a combination of results cannot be explained by a single Type I bug, the next simplest explanation is a single Type II bug. A Type II bug can produce the following combination of results: `A-BB`, `A-BC`, `F-AA`, and `F-AB`. None of these should be produced by a single Type I bug because the epoch-parallel replicas generate the same answer (`A-BB` or `F-AA`) or because there are three outcomes (`A-BC` or `F-AB`). In the latter case,

it is impossible for the outcome to have been produced by a single Type I bug, whereas in the first case, the outcome is merely unlikely. Any epoch-parallel execution should avoid a Type II bug because its non-preemptive execution invalidates one of the bug's failure requirements. For instance, atomicity violation bugs are Type II bugs that are triggered when one thread interposes between two events in another thread. Because threads are not preempted in the epoch-parallel replicas, both replicas avoid such bugs.

We have found that it is common for a single type II bug to result in three different outcomes (e.g., `A-BC` or `F-AB`). For example, consider two threads both logging outputs in an unsynchronized manner. The thread-parallel replica incorrectly garbles the outputs by mixing them together (outcome `A`), one epoch-parallel replica correctly outputs the first value in its entirety before the second (outcome `B`), and the remaining epoch-parallel replica outputs the second value in its entirety before the first (outcome `C`), which is also correct. Similar situations arise when inserting or removing elements from an unsynchronized shared data structure. Thus, when Frost sees an `A-BC` outcome, it commits one of the epoch-parallel replicas.

The remaining combinations (`A-BF`, the isomorphic `A-FB`, and `A-FF`) cannot be explained by a single data race bug. `A-BF` has more than two outcomes, which rules out a single Type I bug. `A-BF` also includes a failing epoch-parallel run, which rules out a single Type II bug. Both epoch-parallel replicas fail in `A-FF`, and this is also not possible from a single Type I or Type II bug. We conclude that these combinations are caused by multiple data race bugs in a single epoch. Frost rolls back to the checkpoint at the beginning of the epoch and executes with a shorter epoch length (trying to encounter only one bug at a time during re-execution).

4.3.3.2 Using the epoch outcome for race detection

Using epoch outcomes for data race detection is more straightforward than using those outcomes to survive races. Any outcome that shows a divergence in systems calls executed (which includes all external output), synchronization operations executed, or final state at the end of the epoch indicates that a data race occurred during the

epoch. Because all three replicas obey the same happens-before order, a data race is the only cause of replica divergence. Further, that data race must have occurred during the epoch being checked because all replicas start from the same initial memory and register state.

Because Frost's data race detection is outcome-based, not all data races that occur during the epoch will be reported. This is a useful way to filter out benign races, which are sometimes intentionally inserted by programmers to improve performance. In particular, an ad-hoc synchronization may never cause a memory or output divergence, or a race may lead to a temporary divergence, such as in values in the stack that are soon overwritten. If Frost explores both orders for a pair of racing instructions and does not report a race, then the race is almost certainly benign, at least in this execution of the program. The only exception, discussed in Section 4.3.4.4, occurs when multiple bugs produce identical-but-incorrect program state or output.

Since Frost allows replicas to diverge slightly during an epoch, it sometimes observes a difference between replicas in system calls or synchronization operations executed, but it does not observe a difference in output or final replica state. Such races are also benign. Frost reports the presence of such races but adds an annotation that the race had no observable effect on program behavior. A developer can choose whether or not to deal with such races.

Because Frost is implemented on top of the DoublePlay framework for deterministic record and replay, it inherits DoublePlay's ability to reproduce any execution of an epoch-parallel replica [91]. Thus, in addition to reporting the existence of each race, Frost also can reproduce on demand an entire execution of the program that leads to each reported race, allowing a developer to employ his or her favorite debugging tools. For instance, we have implemented a traditional dynamic data race detector based on the design of DJIT+ [69] that replays a divergent epoch to precisely identify the set of racing instructions.

4.3.3.3 Sampling

Some recent race detection tools use sampling to reduce overhead at the cost of missing some data races [14, 27, 53]. We added a similar option to Frost. When the user specifies a target sampling rate, Frost creates epoch-parallel replicas for only some epochs; we call these the *sampled epochs*. Frost does not execute epoch-parallel replicas for other epochs, meaning that it neither detects nor survives races during those epochs. Frost dynamically chooses which epochs are sampled such that the ratio of the execution time of the sampled epochs to the overall execution time of the program is equal to the sampling rate. While it is possible to use more sophisticated heuristics to choose which epochs to sample, this strategy has the property that the relative decrease in Frost's ability to survive and detect dynamic data races will be roughly proportional to the sampling rate.

4.3.4 Limitations

Section 4.2 discussed the guarantees that complementary scheduling provides for data race survival and detection in synchronization-free code regions that contain no more than one data race. We now describe the limitations on these guarantees for epochs that do not conform to those properties. We also describe the steps that Frost takes to mitigate these limitations. As the results in Section 4.4.1.2 show, these limitations did not compromise Frost's survival or detection properties in practice when we evaluated Frost with real application bugs.

4.3.4.1 Multiple bugs in an epoch

Although we posit that data race bugs are rare, an epoch could contain more than one bug. If multiple bugs occur in one epoch, Frost could assign an explanation that explained the outcome, but which is incorrect. This would affect both survival and detection guarantees.

Survival requires that at least one replica execute correctly. Adding any number of Type II bugs to an epoch does not affect survival since neither epoch-parallel replica

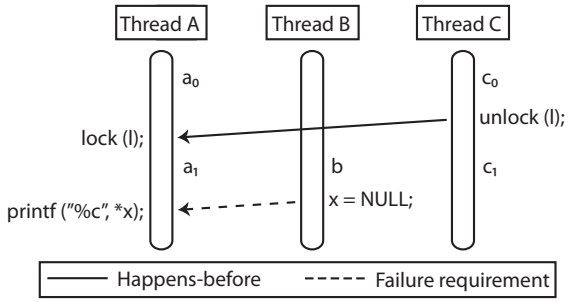


Figure 4.4: Priority inversion scenario

will fail due to such bugs. Thus, one replica will be correct for a synchronization-free region that contains zero or one Type I bugs and any number of Type II bugs. However, the presence of multiple Type I bugs can cause both replicas to fail. Typically, different bugs will cause the program to fail in different ways. The symptom of failure (e.g., crash or abort) might be different, or the memory and register state may be different at the time of failure. Thus, Frost can still take corrective action such as rolling back and executing additional replicas, especially if such failures are self-evident. When Frost rolls back, it substantially reduces the epoch length to separate out different bugs during re-execution. This is a form of search.

It is conceivable, though unlikely, that two different Type I bugs have the same effect on program state, in which case the replicas with complementary schedules could reach the same final state. If the failure is not self-evident, Frost will misclassify the epoch and commit faulty state.

For the purpose of data race detection, multiple data races are only a problem if all races have an identical effect on program state. Otherwise, replicas will diverge and Frost will report a race for the epoch. The presence of multiple data races will subsequently be discovered by the developer when replaying the epoch in question.

4.3.4.2 Priority inversion

The presence of happens-before constraints within an epoch may cause Frost to fail to survive or detect a data race within that epoch. For epochs in which only pairs of threads interact with one another, Frost's algorithm for complementary schedule

generation will construct schedules in which the order of all potentially racing instructions differ. Non-racing instructions may execute in the same order, but, by definition, this does not affect any guarantees about data races.

When more than two threads interact in an epoch, a situation similar to priority inversion may arise and prevent Frost from constructing schedules that change the order of all non-racing instructions. For instance, consider Figure 4.4. The epoch contains three threads, a happens-before constraint due to application synchronization, and a failure requirement caused by two racing instructions. If Frost's assigns the order `ABC` to threads in one replica, the serial order of execution in the two schedules is { a_0, b, c_0, a_1, c_1 } in one replica and { c_0, c_1, b, a_0, a_1 } in the other. All pairs of code segments that occur in different threads execute in a different order in the two schedules, with two exceptions. c_0 executes before a_1 in both schedules. However, this order is required by the application synchronization, and that synchronization prevents these instructions from racing. Additionally, b executes before a_1 in both schedules. If the Type I bug shown in the figure occurs, then both replicas will fail. This may prevent Frost from surviving or detecting the race if the failure does not occur in the thread-parallel execution and it is not self-evident.

Note that Frost could have guaranteed both survival and detection by choosing another set of priorities for the three threads, such as `BAC`. Based on this observation, we have implemented a heuristic that helps choose thread priorities that avoid priority inversion. As a program executes, Frost counts the number of happens-before constraints between each pair of threads. It uses a greedy algorithm to place the two threads with the most frequent constraints in adjacent slots in the priority order, then place the thread with the most frequent constraints with one of those two threads adjacent to the thread with which it shares the most constraints, and so on. Since priority inversion can happen only when a constraint occurs between two non-adjacent threads in the priority order, this heuristic reduces the possibility of priority inversion happening as long as the constraints seen earlier in a program are a good predictor of future constraints.

In some cases, the thread-parallel execution of an epoch may complete before

the epoch-parallel executions begin. In such cases, Frost can observe the exact set of happens-before constraints during that epoch and choose thread priorities accordingly. We have not yet implemented this further optimization.

4.3.4.3 Epoch boundaries

Frost separates execution into epochs to achieve scalability via multicore execution. Each epoch represents an ordering constraint (a barrier) that was not present in the original program. If a failure requirement crosses an epoch barrier (i.e., one of the instructions occurs in a prior epoch and one occurs in a subsequent epoch), the order of these two instructions is fixed in all replicas. For a Type I bug, all replicas will fail together or all will avoid failure.

Frost takes two steps to mitigate this limitation. First, it creates epochs infrequently. Second, it creates an epoch such that all threads are executing a system call at the point the epoch is created. For a data race such as the atomicity violation in Figure 4.1(b), this guarantees that no replica will fail unless the program issues a system call in the region that must be atomic.

All systems (including Frost) that are used to survive harmful races must commit state before externalizing output, and externalizing output is often required for forward progress. To avoid a bug due to a harmful race, such systems must also roll back to some committed state that precedes the race. This committed state may artificially order instructions before and after the commit point, and this ordering constraint may force the program to experience a harmful ordering of racing instructions [48].

When Frost is used only to detect races and not to survive them (e.g., during testing), there may be no need to keep the external output consistent after a data race occurs. Thus, we have implemented an optimization when Frost is used for data race detection in which external output does not cause the creation of a new epoch. This optimization is used only in Section 4.4.2 and not elsewhere in the evaluation.

4.3.4.4 Detection of Type II bugs

Frost's outcome-based race detection may not detect certain Type II bugs. Detection requires that any two replicas differ in system calls or synchronization operations executed, or that two replicas have a different memory or register state at the end of the epoch. As previously mentioned, certain benign races may have this property — filtering out such races is an advantage of outcome-based race detection. In addition, a code region may exhibit this property if the effects of two or more sets of racing instructions are identical. This is most likely to happen for a Type II bug in which both epoch-parallel replicas are correct and finish the epoch in identical states. However, in our experience so far with actual programs, Type II bugs have always led to some difference in program state or output.

4.4 Evaluation

Our evaluation answers the following questions:

- How effectively does Frost survive data race bugs?
- How effectively does Frost detect such bugs?
- What is Frost's overhead?

4.4.1 Detecting and surviving races

4.4.1.1 Methodology

We evaluated Frost's ability to survive and detect data races using a 8-core server with two 2.66 GHz quad-core Xeon processors and 4 GB of DRAM. The server ran CentOS Linux 5.3, with a Linux 2.6.26 kernel and GNU library version 2.5.1, both modified to support Frost.

We used 11 actual concurrency bugs in our evaluation. We started by reproducing all data race bugs from an online collection of concurrency bugs [103] in Apache, MySQL, and pbzip2 compiled form several academic sources [49, 101, 104]

Application	Bug number	Bug manifestation	Outcome	% survived	% detected	Recovery time (sec)
pbzip2	N/A	crash	F-AA	100%	100%	0.01 (0.00)
apache	21287	double free	A-BB or A-AB	100%	100%	0.00 (0.00)
apache	25520	corrupted output	A-BC	100%	100%	0.00 (0.00)
apache	45605	assertion	A-AB	100%	100%	0.00 (0.00)
MySQL	644	crash	A-BC	100%	100%	0.02 (0.01)
MySQL	791	missing output	A-BC	100%	100%	0.00 (0.00)
MySQL	2011	corrupted output	A-BC	100%	100%	0.22 (0.09)
MySQL	3596	crash	F-BC	100%	100%	0.00 (0.00)
MySQL	12848	crash	F-FA	100%	100%	0.29 (0.13)
pfscan	N/A	infinite loop	F-FA	100%	100%	0.00 (0.00)
glibc	12486	assertion	F-AA	100%	100%	0.01 (0.00)

Results are the mean of five trials. Values in parentheses show standard deviations.

Table 4.2: Data race detection and survival

and BugZilla databases. Out of the 12 concurrency bugs in the collection, we reproduced all 9 data race bugs. In addition, we reproduced a data race bug in the application pfscan that has been previously used in academic literature [104]. Finally, during our tests, Frost detected a previously unknown, potentially malign data race in glibc, which we added to our test suite. Table 4.2 lists the bugs and describes their effects.

For each bug, we ran 5 trials in which the bug manifests while the application executes under Frost's shepherding. The fourth column in Table 4.2 shows the replica outcomes for the epoch containing the bug. The fifth column shows the percentage of trials in which Frost survives the bug by producing output equivalent to a failure-free bug. The next column shows the percentage of trials in which Frost detects the bug via divergence in replica output or state. The final column shows how long Frost takes to recover from the bug — this includes the cost of rolling back and executing new replicas.

4.4.1.2 Results

The main result of these experiments is that Frost both survives and detects all 11 bugs in all 5 trials for each bug. For these applications, surviving a bug adds little overhead to application execution time, mostly because epochs are short for server applications such as MySQL and Apache, and the bugs in other applications occurred close to the end of execution, so little work was lost due to quashing future epochs. We next provide more detail about each bug.

The pbzip2 data race can trigger a `SIGSEGV` when a worker thread dereferences a pointer that the main thread has freed. This is a Type II bug because the dereference must occur after the deallocation but before the main thread exits. This failure is self-evident, leading to the `F-AA` epoch outcome.

Apache bug #21287 is caused by lack of atomicity in updating and checking the reference count on cache objects, typically leading to a double free. This is a latent bug: the data race leads to an incorrect value for the reference count, which typically manifests later as an application fault. Frost detects this bug via a memory divergence at the end of the epoch in which the data race occurs, which is typically much earlier than when the fault is exhibited. Early detection allows Frost to avoid externalizing any output corrupted by the data race. The bug may manifest as either a Type I or Type II bug, depending on the order of cache operations.

Apache bug #25520 is a Type II atomicity violation in which two threads concurrently modify a shared variable in an unsafe manner. This leads to garbled output in Apache's access log. Frost detects a memory divergence since the log data is buffered before it is written to the log. The epoch classification is `A-BC` because the failure is not self-evident and the two epoch-parallel executions produce a different order of log messages (both orders are correct since the logged operations execute concurrently).

Apache bug #45605 is an atomicity violation that occurs when the dispatcher thread fails to recheck a condition after waiting on a condition variable. For this bug to manifest, the dispatcher thread must spin multiple times through a loop and accept multiple connections. Frost prevents this bug from manifesting in any replica

because of its requirement that output not be released prior to the end of an epoch. Since `accept` is a synchronous network operation, two `accepts` cannot occur in the same epoch. Thus, Frost converts the bug to a benign data race, which it detects. Even when the requirement for multiple `accepts` is removed, Frost detects the bug as a Type II race and survives it.

MySQL bug #644 is a Type II atomicity violation that leads to an incorrect loop termination condition. This causes memory corruption that eventually may cause MySQL to crash. Frost detects this bug as a memory divergence at the end of the buggy epoch. Thus, it recovers before memory corruption causes incorrect output.

MySQL bug #791 is a Type II atomicity violation that causes MySQL to fail to log operations. In a manner similar to Apache bug #25520, Frost sees an `A-BC` outcome for the buggy epoch, although the difference occurs in external output rather than memory state. Like the Apache bug, the order of output in the two epoch-parallel replicas are different, but both orders are correct.

MySQL bug #2011 is a Type II multi-variable atomicity violation that occurs when MySQL rotates its relay logs. This leads MySQL to fail an error check, leading to incorrect behavior. Frost detects the bug as an `A-BC` outcome.

MySQL bug #3596 is the Type II bug shown in Figure 4.1(b). The NULL pointer dereference generates a self-evident failure. The two epoch-parallel replicas avoid the race and take correct-but-divergent paths depending on how the condition is evaluated. Frost therefore sees the epoch outcome as `F-BC`.

MySQL bug #12848 exposes an incorrect intermediate cache size value during a cache resizing operation, leading MySQL to crash. Although this variable is protected by a lock for most accesses, one lock acquisition is missing, leading to the potential for an incorrect order of operations that results in a Type I bug. Since the crash occurs immediately, the failure is self-evident.

A Type I bug in pfscan causes the main thread to enter a spin-loop as it waits for worker threads to exit due to an incorrect count on the number of such threads. Frost detects the spin-loop as a self-evident failure and classifies the epoch as `F-FA`. The third replica avoids the spin-loop by choosing an order of racing instructions that

violates the bug's failure requirement.

While reproducing the prior bugs, Frost detected an additional, unreported data race bug in glibc. Multiple threads concurrently update `malloc` statistics counters without synchronization, leading to possibly incorrect values. When debugging is enabled, additional checks on these variables trigger assertions. If a data race causes invalid statistics, the assertion can trigger incorrectly. We wrote a test program that triggers this bug reliably. Since the assertion happens in the same epoch as the data race, the failure is self-evident. We have reported this data race to the glibc developer's mailing list and are awaiting confirmation.

In summary, for a diverse set of application bugs, Frost both detects and survives all bugs in all trials with minimal time needed for recovery. For latent bugs that corrupt application state, Frost detects the failure in the epoch that contains the data race bug rather than when the program exhibits a self-evident symptom of failure and thereby avoids externalizing buggy output.

4.4.2 Stand-alone race detection

We next compare the coverage of Frost's data race detector to that of a traditional happens-before dynamic data race detector. Section 4.4.1.2 showed that Frost detects (and survives) all harmful data races in our benchmarks. However, in those experiments, we considered only scenarios in which the race manifests in a harmful manner. This may have made it easier for Frost to detect these races by making it more likely for replicas to diverge.

In this section, we repeat the experiments of Section 4.4.1.2, but we make no special effort to have the bug manifest. That is, we simply execute a sequence of actions that could *potentially* lead to a buggy interleaving of racing instructions. For comparison, we built a data race detector based on the design of DJIT+ [69]. Although it is slow, this data race detector provides full coverage; in other words, it detects all data races that occur during program execution. Since modern data race detectors often compromise coverage for speed (e.g., by sampling), a full-coverage

App	Bug Number	Harmful Race Detected?		Benign Races	
		Traditional	Frost	Traditional	Frost
pbzip2	N/A	5	5	3	1
apache	21287	0	0	55	2
apache	25520	3	3	61	2
apache	45605	3	3	65	2
mysql	644	4	4	2899	2
mysql	791	3	3	808	1
mysql	2011	0	0	1414	1
mysql	3596	0	0	658	2
mysql	12848	0	0	1449	2
pfscan	N/A	5	5	0	0
glibc	12486	6	6	9	3

The third column shows the number of runs in which a full-coverage, traditional dynamic race detector identifies the harmful race and the fourth column shows the number of runs in which Frost identifies the harmful race. The last two columns report the number of benign races detected for that benchmark in our runs.

Table 4.3: Comparison of data race detection coverage

data race detector such as the one we used provides the strongest competition.

Comparing the coverage of race detection tools is challenging since there is ordinarily no guarantee that each tool will observe the same sequence of instructions and synchronization operations during different executions of the program. Fortunately, because Frost is built using the DoublePlay infrastructure, we can use DoublePlay to record the execution of the application and deterministically replay the same execution later. When we execute a dynamic race detector on the replayed execution, it is guaranteed to see the same happens-before order of synchronization operations as observed by both the thread-parallel and epoch-parallel executions. Further, the sequence of instructions executed by each thread is guaranteed to be the same up to the first data race. This ensures an apples-to-apples comparison.

Table 4.3 compares the coverage of Frost to that of the traditional dynamic race

detector. We evaluated each benchmark for the same amount of testing time; the table shows cumulative results for all runs.

For each run for which the traditional data race detector identified a harmful race, Frost also identified the same race. The third column in Table 4.3 lists the number of runs for which the traditional data race detector identified the harmful race. The fourth column shows the number of runs for which Frost identified the same race. For some harmful races, neither Frost nor the traditional data race detector detect the race during our preset testing duration; this is expected since dynamic data race detectors must see instructions execute without synchronization to report a race.

We also evaluated the benefit of the ordering heuristic described in Section 4.3.4.2. When we executed Frost with the heuristic disabled, it detected all harmful races detected in Table 4.3 except for the harmful race in pbzip2. We verified that Frost does not report this race without the heuristic due to a priority inversion.

The last two columns in Table 4.3 list the number of benign races identified by the traditional data race detector and Frost for each benchmark. We manually classified 79 benign races reported by the traditional race detector in the pbzip2, Apache, pfscan and glibc benchmarks, according to a previously-proposed taxonomy [59], with the following results: (a) user-constructed synchronization (42 races): for example, Apache uses custom synchronization that the traditional race detector is unaware of without annotation and so the traditional race detector incorrectly identifies correctly synchronized accesses as racing, (b) redundant writes (8 races): two threads write identical values to the same location, (c) double checks (11 races): a variable is intentionally checked without acquiring a lock and re-checked if a test fails, and (d) approximate computation (18 races): for example, glibc's malloc routines maintain statistics and some threads concurrently log the order in which they service requests without synchronization. We also classified MySQL #644 and found that user-constructed synchronization accounted for 2619 benign data races, redundant writes for 71, double checks for 153 and approximate computation for 156. Due to the effort required, we have not classified the other MySQL benchmarks.

In contrast, Frost reports many fewer benign races. For example, if a race leads

to transient divergence (e.g., an idempotent write-write race), Frost does not flag the race if the replica states converge before the end of the epoch. Frost also need not be aware of custom synchronization if that synchronization ensures that synchronized instructions have identical effects on all replicas. In our benchmarks, Frost identified only 8 benign races (2 double checks and 6 approximate computations). Thus, almost half of the races identified by Frost were harmful, while less than 0.25% of the races identified by the traditional race detector were harmful (with most benign races due to custom synchronization in MySQL).

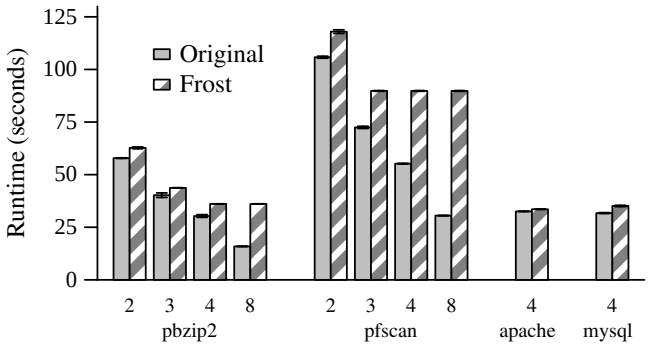
4.4.3 Performance

4.4.3.1 Methodology

Our previous experiment demonstrated Frost's ability to survive and detect data races in pbzip2, pfscan, Apache and MySQL. We next measured the throughput overhead introduced by Frost in these 4 applications by comparing the execution time with Frost on the same 8-core server running our modified Linux kernel and glibc to the execution time running without Frost (i.e., running the same kernel and glibc versions without the Frost modifications).

We evaluate pbzip2 compressing a 498 MB log file in parallel. We use pfscan to search for a string in a directory with 935 MB of log files. We extended the benchmark to perform 150 iterations of the search so that we could measure the overhead of Frost over a longer run while ensuring that data is in the file cache (otherwise, our benchmark would be disk-bound). We tested Apache using ab (Apache Bench) to simultaneously send 5000 requests for a 70 KB file from multiple clients on the same local network. We evaluate MySQL using sysbench version 0.4.12. This benchmark uses multiple client threads to generate 2600 total database queries on a 9.8 GB myISAM database; 2000 queries are read-only and 600 update the database.

For these applications, the number of worker threads controls the maximum number of cores that they can use effectively. For each benchmark, we varied the number of worker threads from two to eight. Some benchmarks have additional control threads



This figure shows how Frost affects execution time for four benchmarks on an 8-core machine. We show results for 2, 3, 4 and 8 threads for pbzip2 and pfscan. Apache and MySQL are I/O bound, so results are the same between 2 and 8 threads; we show the 4 thread results as a representative sample. Results are the mean of five trials—the error bars are 90% confidence intervals. Frost adds a small amount of overhead (3-12%) when there are sufficient cores to run the extra replicas. When the number of worker threads exceeds 3 (pfscan) or 4 (pbzip2), Frost cannot hide the cost of running additional replicas.

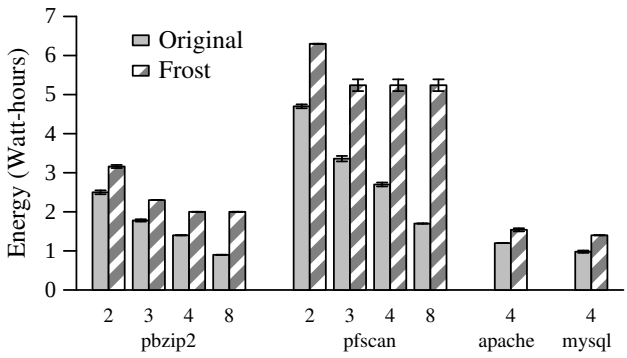
Figure 4.5: Execution time overhead

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. All results are the mean of five trials.

4.4.3.2 Throughput

The primary factor affecting Frost's performance for CPU-bound applications is the availability of unused cores. As Figure 4.5 shows, Frost adds a reasonable 8% overhead for pbzip2 and a 12% overhead for pfscan when these applications use only 2 cores. The reason that Frost's execution time overhead is low is that the server has spare resources to run its additional replicas.

To measure how Frost's overhead varies with the amount of spare cores, we gradually increased the number of threads used by the application up to the full capacity of the 8-core machine. Frost performance for pfscan stops improving at 3 worker threads, which is expected since running 3 replicas with 3 worker threads each requires 9 cores (1 more than available on this computer). Frost performance continues to scale up to 4 worker threads for pbzip2 due to application-specific behavior. A data race in pbzip2 sometimes leads to a spin-loop containing a call to `nanosleep`.



This figure shows Frost's energy overhead. We show results for 2, 3, 4 and 8 threads for pbzip2 and pfscan, and 4 threads for Apache and MySQL. Results are the mean of five trials—the error bars are 90% confidence intervals.

Figure 4.6: Energy overhead

One replica does not consume CPU time when this happens. As expected, if these two CPU-bound applications use all 8 cores, Frost adds slightly less than a 200% overhead. As with all systems that use active replication, Frost cannot hide the cost of running additional replicas when there are no spare resources available for their execution.

In contrast, we found the server applications Apache and MySQL do not scale with additional cores and are hence less affected by Frost's increased utilization. Specifically, we find that Apache is bottlenecked on network I/O and MySQL is bottlenecked on disk I/O. Since Apache and MySQL are not CPU-bound, neither the original nor Frost's execution time is affected as we vary the number of threads from 2 to 8. For this reason, we simply show the results for 4 threads. As shown in Figure 4.5, Frost only adds 3% overhead for Apache and 11% overhead for MySQL.

4.4.3.3 Energy use

Even when spare resources can hide the performance impact of executing multiple replicas, the additional execution has an energy cost. On perfectly energy-proportional hardware, the energy overhead would be approximately 200%. We were interested to know the energy cost on current hardware, which is not particularly energy-proportional.

We used a Watts Up? .Net power meter to measure the energy consumed by the 8-core machine when running the throughput benchmarks with and without Frost. As Figure 4.6 shows, Frost adds 26% energy overhead for pbzip2 and 34% overhead for pfscan when run with 2 threads. The energy cost increases to 122% and 208% respectively when the applications use 8 worker threads. Frost adds 28% energy overhead for Apache and 43% overhead for MySQL, independent of the number of worker threads.

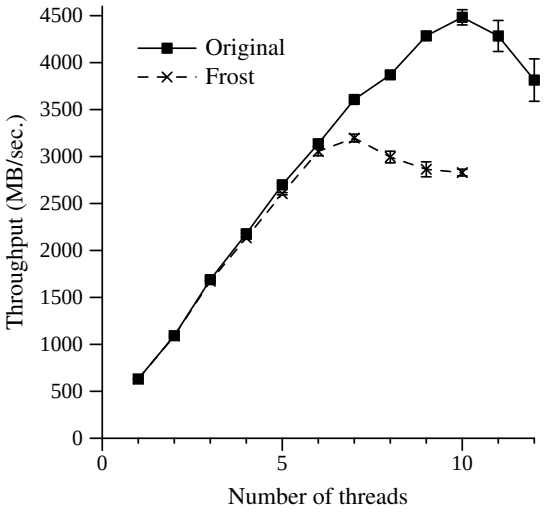
4.4.3.4 Scalability

As the 8-core machine runs out of CPU resources with only 2-3 worker threads, we next evaluate how Frost scales on a 32-core server with four 2.27 GHz 8-core Xeon X7560 processors and 1.8 GB of RAM. This server ran the same software as in the previous experiments. We look at pfscan in these experiments as it showed the highest 8-core overhead previously. We scaled up the benchmark by increasing the number of data scans by a factor of 100. We report the throughput, measured by the amount of data scanned by pfscan per second.

As Figure 4.7 shows, pfscan scales well without Frost until it reaches 10 cores. At this point, we conjecture that it is using all the available memory bandwidth for the 32-core computer. Frost scales well up to 6 cores on this computer, with overhead less than 4%. Frost's execution of pfscan achieves maximum throughput at 7 cores. We conjecture that it is hitting the memory wall sooner due to executing multiple replicas. Because replicas execute the same workload, cache effects presumably mitigate the fact that the combined replicas access 3 times as much data as the original execution.

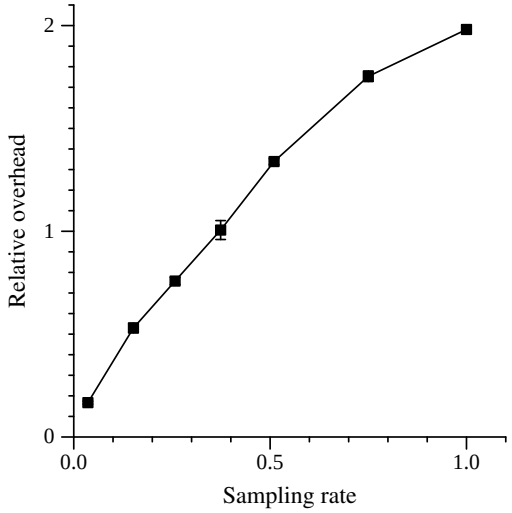
4.4.3.5 Sampling

As described in Section 4.3.3.3, Frost can be configured to sample only a portion of a program's execution. Sampling reduces overhead, but Frost will only detect and/or survive races data in the sampled intervals for which it executes epoch-parallel replicas. Thus, Frost will experience a decrease in its survival and detection rates for dynamic data races that is proportional to the sampling rate.



This figure shows the throughput (MB of data scanned per second) by pfscan running with and without Frost. We vary the number of threads in pfscan from 1 to 10. Results are the mean of five trials—the error bars are 90% confidence intervals.

Figure 4.7: Scalability on a 32-core server



This figure shows Frost's relative overhead running pfscan at various sampling rates. A sampling rate of 0.25 means that Frost detects and survives races in 1 out of 4 epochs. Results are the mean of five trials; error bars are 90% confidence intervals.

Figure 4.8: Effect of sampling on relative overhead

We re-ran the CPU-bound pfscan benchmark with 8 worker threads on the 8-core computer used in our previous experiments. We varied the percentage of epochs sampled and measured the relative overhead that Frost adds to application execution

time. Figure 4.8 shows that with a sampling rate of 3.5%, Frost adds only 17% relative overhead to the benchmark. As the sampling rate increases, Frost's relative overhead scales roughly linearly up to approximately 200% when no sampling is employed.

4.4.4 Discussion

In summary, Frost detects all harmful races detected by a full-coverage dynamic race detector and survives those races in our experiments. While these results are quite positive, we believe there are a small set of scenarios that Frost will fail to handle correctly, as described in Section 4.3.4. Frost's overhead ranges from 3–12% for the applications we measured when spare resources are available to execute additional replicas. When spare resources are not available, the cost of executing additional replicas cannot be masked. Frost scales well with the number of cores, though it may experience limitations in other resources such as memory bandwidth if that resource is the bottleneck for the application being executed.

Frost's measured overhead is slightly less than that reported for the DoublePlay system on which it is built [91] due to code optimizations added after the reporting of the DoublePlay results. Uniparallel execution, as used by both Frost and DoublePlay, can have substantially higher overheads for benchmarks that dirty memory pages very rapidly. For example, by far the worst case overhead we measured for DoublePlay was the `ocean` benchmark in the SPLASH-2 suite (121% with spare cores); we expect Frost would have similar overhead with spare cores and 3x that overhead without spare cores.

These measured overheads are substantially less than those reported for dynamic data race detectors that handle non-managed code. As with other systems that use multiple replicas, Frost offers a tradeoff between reliability and utilization. During the software life cycle, one may choose to employ Frost at different times as priorities change; e.g., one can use Frost when software is newly released or updated to survive and detect data race bugs, then disable Frost or sample a subset of epochs to reduce overhead when software is believed to be race-free. Additionally, since it is inherently

difficult to scale many workloads (e.g., those that are I/O bound), spare cores may often be available in production, in which case Frost can mask its extra utilization. One could, for instance, use a variation of sampling that only runs extra replicas when spare cores are available.

4.4.5 Frost versus Triple Modular Redundancy

When viewed through the lens of fault-tolerance, the cost of running Frost's three replicas is the same as triple modular redundancy (TMR): they both impose a 3X hardware utilization cost. The obvious distinction between the two is that Frost uses complementary schedules to tightly control the epoch-parallel replicas so at least one replica survives the data race bug, while TMR runs three loosely controlled replicas. The question then is whether Frost's complementary schedules are more effective than TMR's loosely controlled replicas at surviving harmful data race bugs.

We observe that when TMR is used for tolerating non-fail-stop faults (e.g., transient hardware faults), the system must bound the window over which at most one fault can occur by periodically comparing the state of the replicas for divergences. The frequency of these comparisons will bound how much work the system loses in case of a fault. Hence, both TMR and Frost experience the overhead of periodic checkpointing and comparison of replica state.

If the program state of the TMR replicas is to match when executing a multi-threaded program on a commodity multiprocessor, care must be taken to ensure that all replicas operate on the same program input and observe the same happens-before order of synchronization operations. If the TMR replicas do not observe the same program input, they could falsely diverge. Similarly, if the TMR replicas do not observe the same happens-before order of synchronization operations, a benign race (for instance, an incorrect worker could be assigned a task in a work-queue) could lead to replica divergence. A practical solution is to let one of the TMR replicas run ahead of the others and generate a log of program inputs and synchronization operations that the trailing replicas can replay. Additionally, all three TMR replicas should execute

until the same thread barrier (i.e., epoch boundary) before comparing program state.

As previous research has shown, harmful data race bugs are rare and highly unlikely to manifest [49]. Hence, the likelihood of a race manifesting on a production system that deploys Frost or TMR is extremely low. So, the question is: if a data race bug does manifest in the leading TMR replica, what is the likelihood that the other two replicas survive the bug? Since all three replicas see the same program inputs and obey the same happens-before ordering, their program state is identical until the last synchronization operation before the data race. By letting threads run freely between synchronization operations, TMR leaves it up to chance whether the other two replicas execute the racing instructions in a manner that avoids the harmful bug.

The challenge in quantifying the likelihood of whether TMR survives non-fail-stop harmful data race bugs is twofold: first, we would have to build a TMR system based on the design described above, and second, we would have to dedicate sufficient testing time so each of our diverse set of rare data race bugs manifests. As described in Section 4.4.2, we could not reproduce 4 of these data race bugs in our testing duration and several others caused our epoch-parallel replicas to diverge before manifesting in the thread-parallel execution (for instance, Frost detects a memory divergence in the replica states after encountering MySQL bug #644, so it initiates recovery before the bug causes a segmentation fault in the thread-parallel execution). Since an experimental comparison is hard, we instead provide a logical comparison.

Let $p(bug\_in\_natural\_run)$ be the probability that a harmful data race bug occurs in a given epoch of execution for a natural run of the program. In Frost, the thread-parallel execution is the only natural run as both epoch-parallel replicas replay the log captured by the thread-parallel execution. Similarly, the leading replica in the TMR system is the only natural run as it provides the log of program input and synchronization operations to the trailing replicas. Since data race bugs are rare, we know that $p(bug\_in\_natural\_run)$ is very small for both Frost and TMR.

Despite being highly unlikely, these rare data race bugs do manifest (as evidenced by BugZilla and similar bug repositories). So the question we ask is: what is the con-

ditional probability that the identical data race bug occurs in the two additional replicas given that the bug also occurred in the natural run? That is, what is $p(bug\_in\_both\_replicas \mid bug\_in\_natural\_run)$?

Some might say that the three replicas running under TMR are independent. Hence, given a rare data race bug that occurs with probability p, the likelihood of all three replicas experiencing the failure is p^3. But in practice, TMR systems do not run independent replicas. Specifically, if a program is running under TMR with independent replicas, either its application code has to be modified, or a detailed specification generated as in Pike [30], so its state can be periodically compared. A more practical approach is to construct the TMR system such that one of the replicas leads the execution and generates a log of events that the trailing replicas replay.

A TMR system as described above executes highly correlated replicas as all three replicas start executing from the same checkpoint, see the same program input and obey the same order of happens-before synchronization operations. Hence, if one replica experiences a failure, it is highly likely that the other replicas will experience the same failure. For example, if the leading replica executes a rare synchronization operation that results in a harmful thread interleaving, the trailing TMR replicas, which obey the log of program inputs and happens-before synchronization operations, are likely to execute the same harmful thread interleaving too. If this harmful interleaving results in one of the replicas failing, it is highly likely that the two trailing replicas also fail in the exact same way. In this case, despite the low probability that the leading TMR replica fails due to a harmful race bug, the conditional probability that the identical data race bug causes the two trailing replicas to fail is almost 100%.

In contrast to TMR, Frost uses complementary schedules so the two epoch-parallel replicas execute thread schedules that are as dissimilar as possible to ensure that at least one of the two replicas does not execute the harmful interleaving. So, even in the rare case that the leading thread-parallel execution executes a harmful interleaving and fails, the likelihood that both epoch-parallel replicas fail in the exact same way is zero, in the absence of any of the limitations described in Section 4.3.4. This is empirically supported in our evaluation where Frost survives harmful data race bugs

with 100% probability, for our tested benchmarks.

4.5 Conclusion

Frost introduces two main ideas to mitigate the problems of data races: complementary schedules and outcome-based race detection. Running multiple replicas with complementary schedules ensures that, for most types of data race bugs, at least one replica avoids the order of racing instructions that leads to incorrect program execution. This property enables a new, faster dynamic data race detector, which detects races by comparing outcomes of different replicas rather than analyzing the events executed. After Frost detects a data race, it analyzes the combination of results and selects the strategy that is most likely to survive the bug.

CHAPTER 5

Related Work

5.1 Uniparallelism

Uniparallelism builds upon many ideas from research on using speculation to run applications in parallel, such as thread-level speculation [84, 86, 65]. In particular, Master/Slave Speculative Parallelization [108], Predictor/Executor by Süßkraut [87], SuperPin [94], Speck [62], and Fast Track [40] also use a fast execution to start multiple slow executions in parallel. Uniparallelism applies this idea in a new way by using a fast, thread-parallel execution on multiple processors to start multiple uniprocessor executions that execute threads sequentially on one processor. Running a uniprocessor execution is critical to utilizing techniques such as deterministic replay, data race detection and data race survival, that are much more efficient on a uniprocessor than a multiprocessor. These techniques cannot be used with master/slave parallelism, which runs epochs in parallel on multiple processors.

Restricting each epoch-parallel execution to a single processor allows us to tightly control the schedule with which threads are timesliced on the processor. DoublePlay implements a deterministic scheduler to ensure that the thread schedule executed by an epoch-parallel execution during recording is reproduced during replay. The idea of controlling thread schedules has also been used to explore the space of possible thread interleavings in model checking and program testing [33, 56]. The goal of such prior work is to explore the space of the possible behaviors to find bugs. In contrast, the

primary goal of Frost is to ensure that at least one of the thread schedules assigned to an epoch-parallel replica executes racy accesses in the correct order. This difference changes the algorithm used to create schedules and leads to the design choice in Frost to use two complementary schedules instead of many schedules. Like Frost, CHESS [56] uses non-preemptive scheduling to tightly control the thread schedule. However, because CHESS is used only for testing, it has no need to parallelize the execution of non-preemptive runs which Frost achieves through uniparallelism.

The primary cost of uniparallelism is the increased hardware utilization resulting from the need to execute multiple instances of the program. One way to reduce this cost is to leverage the speculative control and data flow prediction scheme introduced in SlipStream [71] so the leading thread-parallel execution communicates values and branch outcomes to the trailing epoch-parallel execution.

5.2 DoublePlay

Because of its wide array of uses, deterministic replay has been the subject of intense research by the hardware and software systems communities.

Many of the early replay systems focused on uniprocessor replay; e.g., IGOR [28], Hypervisor [15], Mach 3.0 Replay [74], DejaVu [18], ReVirt [24], and Flashback [85]. The designers of these systems observed that when executing a multithreaded program on a uniprocessor, there are many fewer thread switch events than accesses to shared memory. DoublePlay leverages this observation by logging and replaying epochs that each run the multithreaded program on a single processor, while taking advantage of multiple processors by starting multiple epochs in parallel.

Multiprocessor replay has been a particular area of focus in recent years because of the growing prevalence of multi-core processors. The main difficulty in replaying multithreaded programs on multiprocessors is logging and replaying shared memory accesses efficiently. One approach is to log the order of shared accesses [44], but this incurs high overhead. Systems such as SMP-ReVirt [25] and SCRIBE [42] use page protections to log only the order of conflicting accesses to memory pages. The cost

of handling memory protection faults can be quite high in these systems due to true or false sharing, though this cost can be lowered by intelligent scheduling. Another approach is to log and replay the values returned by load instructions [11, 58], but this also incurs high overhead.

One way to reduce the overhead of logging multiprocessors is to add hardware support. The most common strategy is to modify the cache coherence mechanism to log the information needed to infer the order of shared memory accesses [4, 60, 35, 55, 57, 92]. While these approaches are promising, we would like to support deterministic replay on commodity multiprocessors.

Another response to the high overhead of logging multiprocessors is to reduce the scope of programs that can be replayed. RecPlay [73] logs only explicit synchronization operations and so is unable to replay programs with unsynchronized accesses to shared memory (data races). DoublePlay also logs synchronization operations, but it uses these only as hints to guide the execution of the epoch-parallel run; DoublePlay guarantees deterministic replay for programs with and without data races by logging all non-determinism in the epoch-parallel run.

Researchers have also tried to reduce logging overhead by relaxing the definition of deterministic replay. Instead of requiring that all instructions return the same data returned in the original run, these approaches provide slightly weaker guarantees, which still support the proposed uses of replay. PRES [67] and ODR [1] guarantee that all failures in the original run are also visible during replay, where failures are usually defined as the output of the program and program errors (e.g., assertions). Respec [46] guarantees that both the output and the ending state of the replayed execution match the logged execution. DoublePlay requires the same criteria as Respec for deterministic replay when evaluating whether the epoch-parallel execution matches the thread-parallel execution, since this guarantees that each checkpoint that starts a future epoch matches the ending state of the prior epoch.

Respec also distinguishes between online and offline replay [46]. Online replay refers to the ability to replay while the recording is in progress; offline replay refers to the ability to replay after the recording is complete. Respec provides only online

replay. DoublePlay uses online replay, but only for the purpose of guiding the epoch-parallel execution. DoublePlay supports offline replay by logging and replaying the epoch-parallel execution.

Another way to reduce logging overhead is to shift work from the recording phase to the offline replay phase. To achieve this, recent research has investigated an alternate approach to deterministic replay based on *search* [1, 67, 45, 98, 107]. Rather than logging enough data to quickly replay an execution, these systems record a subset of information (e.g., synchronization operations or core dumps), then use that information to guide the search for an equivalent execution. The search space includes all possible orders of shared-memory accesses that are allowed by the logged information, so it grows exponentially with the number of racing accesses. With good heuristics, search-based replay can often find equivalent executions quickly, especially for programs with few racing accesses. However, because of the exponential search space, they may not be able to find an equivalent execution within a reasonable time frame. In addition, even if the search succeeds within a few tries, the execution of the program during search can be slowed by several orders of magnitude due to the need to log the detailed order of shared-memory accesses [67]. DoublePlay logs similar information as some of these systems (e.g., the SYS configuration in PRES [67]), so its recording should add a similar amount of overhead to the original application. One can view DoublePlay as using extra cores to search for an equivalent replay *during the original run*. However, while other systems risk not being able to later find an equivalent run quickly (or at all), DoublePlay verifies during recording that the epoch-parallel execution matches the thread-parallel execution. While DoublePlay must execute intervals that contain races sequentially, this is unlikely to slow the program significantly because these intervals are likely to be a small fraction of the overall execution time.

Deterministic replay helps reproduce non-deterministic multiprocessor executions. An alternative approach is to ensure that all inter-thread communication is deterministic for a given input [23, 12, 9, 64]. This approach eliminates the need to log the order of shared-memory accesses. However, current solutions for deterministic execu-

tion only support programs that are free of data races [64], require language [12] or hardware [23] support, increase runtime severalfold [7, 8], or only support programs with fork-join parallelism [9] or shared-nothing address spaces [2].

In summary, DoublePlay distinguishes itself from prior software-only multiprocessor deterministic replay systems by adding little overhead to application execution time during recording while also guaranteeing that the recorded execution will be able to be replayed in the future in a reasonable amount of time. The cost of this guarantee is that DoublePlay must use additional cores to run two executions of the program during recording.

5.3 Frost

As Frost can serve as a tool for either surviving or detecting data races, we discuss related work in both areas.

5.3.1 Data race survival

The idea of using replication to survive errors dates back to the early days of computing [93, 51]. In active (state-machine) replication, replicas run in parallel and can be used to detect errors and vote on which result is correct [79]. In passive (primary-backup) replication, a single replica is used until an error is detected, then another replica is started from a checkpoint of a known-good state [16]. Passive replication incurs lower run-time overhead than active replication but cannot detect errors by comparing replicas. Frost uses active replication to detect and survive programming bugs.

In 1985, Jim Gray observed that just as transient hardware errors could be handled by retrying the operation (a type of passive replication), some software errors (dubbed Heisenbugs) could be handled in the same manner. Researchers have extended this idea in many ways, such as retrying from successively older states [96], proactively restarting to eliminate latent errors [37], shrinking the part of the system that needs to be restarted [17], and reducing the cost of running multiple replicas [36].

A general technique to increase the chance of survival in replication-based systems is to use *diverse* replicas to reduce the probability of all replicas failing at the same time. Many types of diversity can be added, including changing the layout of memory [10, 31, 72], changing the instruction set [6, 39], or even running multiple independently-written versions of the program [3]. Our focus on ensuring at least one correct replica is similar to work in security that creates replicas with disjoint exploitation sets [21, 76].

The replication-based systems most closely related to Frost are those that add diversity by changing the scheduling of various events, such as changing the order in which messages or signals are delivered [72, 96] or changing the priority order of processes [72]. Frost contributes to the domain of replica diversity by introducing the idea of complementary schedules, describing how complementary schedules enable data race detection, and showing how to produce complementary schedules efficiently via non-preemptive scheduling and uniparallelism.

Past research has examined several approaches that do not require active replication for surviving concurrency bugs that cause deadlocks [38, 95]. Frost is complementary to these techniques as it targets a different class of concurrency bugs due to data races. Instead of detecting concurrency bugs and then recovering from them, recent research proposes to actively avoid untested thread interleavings and thereby reduce the chance of triggering concurrency bugs. This approach, however, incurs high overhead [22] or requires processor support [104]. Other researchers have observed that some concurrency bugs can be eliminated by minimizing preemptions and providing sequential semantics [9]. Other systems [97] avoid known bugs by avoiding thread schedules that lead to the buggy behavior; unlike Frost, these systems do not survive the first occurrence of unknown bugs.

5.3.2 Data race detection

In addition to its survival functionality, Frost can also be used as a dynamic race detection tool, targeted either at production or test environments. Data race detectors

can be compared along many dimensions, including overhead, coverage (how many data races are detected), accuracy (how many false positives are reported), and fidelity (how much data about each race is provided).

Static race detectors (e.g., [26]) try to prove that a program is free of data races; they incur no runtime overhead but report many false positives (lowering accuracy) due to the limits of static analysis, especially for less-structured languages such as C. On the other hand, dynamic race detectors seek only to detect when a specific run experiences a data race; they must observe potentially racing instructions execute in order to report a race. Prior dynamic data race detectors are mostly based on two basic techniques: happens-before analysis [43, 80] and lockset analysis [78]. Both techniques analyze the synchronization and memory operations issued by a program to determine whether a data race may have occurred. Because memory operations occur frequently, dynamic race detectors have historically slowed programs by an order of magnitude. In a recent study, Flanagan and Freund [29] compared several state-of-the-art dynamic data race detectors and showed that their best Java implementation is about 8.5x slower than native execution. Implementations that check for data races in less-structured, optimized code running outside of a virtual machine (such as C and C++ programs) may have even higher overhead, as exemplified by recently-released industrial strength race detectors from Intel [75] and Google [82], which incur more than 30x performance overhead.

Dynamic race detectors can use language-specific or runtime-specific features to reduce overhead. RaceTrack [105] runs CPU-intensive benchmark in Microsoft's CLR 2.6-3.2x slower, but limits coverage by not checking for races involving native code, which represents a non-negligible number of methods. RaceTrack also leverages the object-oriented nature of the checked code to employ a clever refinement strategy in which it first checks for races at object granularity, then subsequently checks accesses to the object for races at field granularity. Object-granularity checks may have substantial false positives, so are reported at lower priority. However, unless a particular pair of instructions races twice for the same object, RaceTrack cannot report the race with high confidence. Overhead can also be reduced by eliminating checks that are

shown to be unnecessary via a separate static analysis phase [19]. However, these optimizations are difficult to implement precisely for unsafe languages.

Frost executes applications 3–12% slower if spare cores are available to parallelize replica execution, and approximately 3x slower if no spare cores are available. This compares very favorably with all prior dynamic race detection tools for general code running outside of a virtual machine, and also with most tools for managed code. While Frost may miss races that are detected by the higher-overhead happens-before race detectors, in practice Frost has detected all harmful races that would be reported by such detectors.

Several recent race detectors use sampling to trade coverage for reduced overhead by monitoring only a portion of a program's execution. PACER [14] uses random sampling, so has coverage approximately equal to the sampling rate used. At a 3% sampling rate, PACER runs CPU-intensive applications 1.6-2.1x slower. However, PACER reports only 2–20% of all dynamic races at that sampling rate. LiteRace [53] uses a heuristic (adaptive bursty thread-local sampling that biases execution toward cold code) to increase the expected number of races found, but the same heuristic may systematically bias against finding certain races (such as those executed along infrequent code paths in frequently-executed functions). LiteRace runs CPU-intensive applications 2.4x slower to find 70% of all races and 50% of rare races.

Sampling is orthogonal to most data race detection techniques. Frost implements sampling by checking only a portion of epochs. At a slightly greater than 3% sampling rate, Frost's overhead is only 17% for a CPU-bound benchmark. It would also be possible to use heuristics similar to those used by LiteRace, but the application is complicated by the granularity of Frost's epoch. Whereas LiteRace toggles instrumentation at function granularity, Frost can only toggle instrumentation at epoch granularity. However, Frost could benefit from its thread-parallel execution, for example by measuring the percentage of cold code executed before deciding which epochs to check via epoch-parallel execution.

It is possible to reduce dynamic data race detection overhead further through the use of custom hardware [70]. Data Collider [27] repurposes existing hardware

(watchpoints) to implement a novel dynamic race detection technique. Data Collider samples memory accesses by pausing the accessing thread and using watchpoints to identify unsynchronized accesses to the memory location made by other threads. The paucity of hardware watchpoints on existing processors (4 in their experiments) limits the number of memory locations that can be sampled simultaneously. Data Collider can thus achieve very low overhead (often less than 10%) but may not have suitable coverage to detect rare races since the sampling rate (only 4 memory locations at a time) is very low. It is also not clear how Data Collider will scale as the number of cores increase because the number of watchpoints per core does not increase, and sampling an address requires an IPI to all cores to set a watchpoint.

Most data races are not bugs. Prior work has shown that comparing execution outcomes for schedules with different orderings of conflicting memory accesses can be used to classify data races as benign or potentially harmful [59]. This can be viewed as a method of improving accuracy. Frost's design applies this filtering technique. In contrast to the prior work that assumed that the data race was known in order to generate thread schedules, Frost uses complementary schedules to detect races that are *unknown* at the time that the schedules are generated.

Frost has extremely high fidelity because it can deterministically replay the execution of a program up to the first data race in an epoch (and often beyond that). This allows Frost to re-generate any diagnostic information, such as stack traces, required by a developer. We use this capability, for example, in Section 4.4.2 to implement a complete dynamic race detector. Other tools such as Intel's ThreadChecker [75] provide stack traces for both threads participating in a data race, and some tools, such as RaceTrack [105] can guarantee a stack trace from only one thread.

Pike [30] also uses multiple replicas to test for concurrency bugs by comparing executions with interleaved requests with executions with serialized requests (which are assumed to be correct). Pike requires that the application provide a canonicalized representation of its state that is independent of thread interleavings, which could be time-consuming to develop. Pike has high overhead (requiring a month to test one application) but can find more types of concurrency bugs than just data race bugs.

TightLip [106] compares the output of a replica with access to sensitive data with that of a replica without such access to detect information leaks.

CHAPTER 6

Conclusion

We next describe how we plan to extend our work on uniparallel execution and summarize the contributions of this thesis.

6.1 Future Work

We plan to extend this work in two directions: reduce the utilization penalty of uniparallelism and leverage uniparallelism to improve software reliability.

If there are spare cores available, DoublePlay slows program execution only modestly (15% with 2 threads, 28% with 4 threads). Similarly, given spare cores, Frost slows program execution by 3—12% for the tested benchmarks. The main cost of uniparallelism is the increased hardware utilization caused by executing the program multiple times; if there are no spare cores, this increased utilization doubles program runtime for DoublePlay and triples program runtime for Frost.

One approach to reduce the extra utilization cost is to distribute the thread-parallel execution and epoch-parallel executions across computers. Being able to use multiple computers makes it more likely that there are spare cores that can hide the extra utilization of uniparallelism. We could leverage multicast [5] to simultaneously transmit each checkpoint from the computer running the thread-parallel execution to the group of computers running the epoch-parallel executions. To reduce network traffic, we need only send the pages modified since the prior epoch. For Double-

Play, the total number of pages sent in these checkpoints is equal to the number of pages compared in Table 3.1. For the applications shown in Table 3.1, the network bandwidth needed to support distributed DoublePlay ranges from 0.5 Mbits/second (pfscan with 2 threads) to 736 Mbits/second (ocean with 4 threads). Frost builds upon DoublePlay so requires a comparable number of page transmissions for these applications. The required intra-computer communication bandwidth is easily available if the computers happen to be co-located in modern data centers [34]. Another approach to reduce the utilization cost is to leverage information gained while running the thread-parallel execution to speed up the epoch-parallel execution (e.g., by improving cache prefetching or branch prediction accuracy [71]).

With uniparallelism, the epoch-parallel execution timeslices threads onto a single processor. As DoublePlay demonstrates, running on a uniprocessor makes the epoch-parallel execution much easier to replay. We plan to explore how to take advantage of uniparallelism to improve software reliability. For example, the Frost system uses the control that uniparallelism affords over thread interleaving to create replicas with complementary thread schedules, which can be used to detect and avoid data races. By limiting preemptions, one can also use uniparallelism to execute program regions atomically, which could in turn be used to improve the semantics of concurrent programs [50], support optimistic concurrency, or enable deterministic execution.

Lastly, it would be nice if we can employ uniparallel execution to speed up offline replay, which currently runs at the speed of single-threaded execution. Basically, we would execute multiple threads of a replayed process on multiple cores. At epoch boundaries, we create a copy of the address space via a multi-threaded fork and execute the forked replay process on a single core using the logged deterministic schedule. When the single-threaded replay reaches the end of its epoch, we check the architectural state. If a divergence is detected, we roll back the multi-core replay to the previous epoch and retry. This technique would be useful, for example, to skip ahead to the next break point during debugging.

6.2 Thesis contribution

This thesis makes both a conceptual and system level contribution.

At a conceptual level, we introduce the notion of uniparallel execution which allows systems to benefit from properties that are easy to achieve on a uniprocessor while still scaling performance with increasing processors. With the advent of multicore and manycore computing, we believe that novel approaches like uniparallel execution that can utilize spare resources to improve the reliability of multithreaded applications will prove useful in both the home and enterprise environments.

At the system level, this thesis presents a detailed design for uniparallel execution and explains which techniques benefit from uniparallel execution. To prove our design, we implemented support for uniparallel execution in the Linux kernel.

We demonstrate the utility of uniparallel execution by addressing two challenging problems. First, we used uniparallelism to implement the DoublePlay system for recording multithreaded execution on commodity multiprocessors. Second, we use uniparallelism to implement the Frost system for data race detection and survival. Our evaluation of DoublePlay and Frost on a variety of desktop, network and scientific benchmarks demonstrates that uniparallel execution allows these systems to improve upon existing approaches in both their efficiency and effectiveness.

Finally, we plan to provide the tools and infrastructure we have built to other members of our research group, and the wider research community, that are interested in applications of uniparallel execution.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] 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.

[2] 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).

[3] Avizienis, A. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering SE-11*, 12 (December 1985), 1491–1501.

[4] Bacon, D. F., and Goldstein, S. C. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging* (1991), ACM Press, pp. 194–206.

[5] Banerjee, S., Bhattacharjee, B., and Kommareddy, C. Scalable application layer multicast. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '02)* (August 2002), pp. 205–217.

[6] Barrantes, E., Ackley, D., Forrest, S., Palmer, T., Stefanovic, D., and Zovi, D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)* (Washington, DC, October 2003).

[7] 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.

[8] Bergan, T., Hunt, N., Ceze, L., and Gribble, S. D. Deterministic process groups in dOS. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[9] 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.

[10] Berger, E. D., and Zorn, B. G. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation* (Ottawa, Canada, June 2006).

[11] Bhansali, S., Chen, W., de Jong, S., Edwards, A., and Drinic, M. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments* (June 2006), pp. 154–163.

[12] 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.

[13] Boehm, H. J., and Adve, S. Foundations of the c++ concurrency memory model. In *Proceedings of PLDI* (2008), pp. 68–78.

[14] Bond, M. D., Coons, K. E., and McKinley, K. S. PACER: Proportional detection of data races. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation* (Toronto, Canada, June 2010).

[15] Bressoud, T. C., and Schneider, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 80–107.

[16] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. *The primary-backup approach.* Addison-Wesley, 1993. in Distributed Systems, edited by Sape Mullender.

[17] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. Microreboot – A technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 31–44.

[18] 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).

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

[20] Chow, J., Garfinkel, T., and Chen, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Technical Conference* (June 2008), pp. 1–14.

[21] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. N-variant systems: A secretless framework for security through diversity. In *USENIX Security* (August 2006).

[22] Cui, H., Wu, J., Tsai, C.-C., and Yang, J. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[23] Devietti, J., Lucia, B., Ceze, L., and Oskin, M. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 2009 International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS) (March 2009), pp. 85–96.

[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] Engler, D., and Ashcraft, K. RacerX: Efficient static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, 2003), pp. 237–252.

[27] Erickson, J., Musuvathi, M., Burckhardt, S., and Olynyk, K. Effective data-race detection for the kernel. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[28] 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.

[29] 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.

[30] Fonseca, P., Li, C., and Rodrigues, R. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the European Conference on Computer Systems* (Salzburg, Austria, April 2011).

[31] Forrest, S., Somayaji, A., and Ackley, D. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems* (Cape Cod, MA, May 1997), pp. 67–72.

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

[33] Godefroid, P. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, January 1997), pp. 174–186.

[34] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Vl2: a scalable and flexible data center network. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '09)* (August 2009), pp. 51–62.

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

[36] Huang, R., Den, D. Y., and Suh, G. E. Orthrus: Efficient software integrity protection on multi-cores. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, March 2010), pp. 371–383.

[37] Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium of Fault-Tolerant Computing* (Pasadena, CA, June 1995), pp. 381–390.

[38] Jula, H., Tralamazza, D., Zamfir, C., and Candea, G. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (San Diego, CA, December 2008), pp. 294–308.

[39] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)* (Washington, DC, October 2003).

[40] Kelsey, K., Bai, T., Ding, C., and Zhang, C. Fast Track: A software system for speculative program optimization. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization (CGO)* (March 2009), pp. 157–168.

[41] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference* (April 2005), pp. 1–15.

[42] Laadan, O., Viennot, N., and Nieh, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)* (June 2010), pp. 155–166.

[43] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

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

[45] 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).

[46] Lee, D., Wester, B., Veeraraghavan, K., Chen, P. M., Flinn, J., and Narayanasamy, S. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, March 2010), pp. 77–89.

[47] Leveson, N. G., and Turner, C. S. Investigation of the Therac-25 accidents. *IEEE Computer 26*, 7 (1993), 18–41.

[48] 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).

[49] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), pp. 329–339.

[50] Lucia, B., Ceze, L., Strauss, K., Qadeer, S., and Boehm, H.-J. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *Proceedings of the 2010 International Symposium on Computer Architecture* (June 2010), pp. 210–221.

[51] Lyons, R. E., and Vanderkulk, W. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development 6*, 2 (1962), 200–209.

[52] Manson, J., Pugh, W., and Adve, S. The Java memory model. In *Proceedings of POPL* (2005), pp. 378–391.

[53] Marino, D., Musuvathi, M., and Narayanasamy, S. LiteRace: efficient sampling for lightweight data-race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 2009).

[54] Mellor-Crummey, J. M., and LeBlanc, T. J. A Software Instruction Counter. In *Proceedings of the 1989 International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 78–86.

[55] Montesinos, P., Ceze, L., and Torrellas, J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 2008 International Symposium on Computer Architecture* (June 2008), pp. 289–300.

[56] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (San Diego, CA, December 2008), pp. 267–280.

[57] Narayanasamy, S., Pereira, C., and Calder, B. Recording shared memory dependencies using Strata. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 229–240.

[58] Narayanasamy, S., Pokam, G., and Calder, B. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)* (June 2005), pp. 284–295.

[59] 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).

[60] Netzer, R. H. B. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (1993), pp. 1–11.

[61] 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.

[62] 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.

[63] 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.

[64] Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 2009 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2009), pp. 97–108.

[65] Oplinger, J., and Lam, M. S. Enhancing software reliability using speculative threads. In *Proceedings of the 2002 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (October 2002), pp. 184–196.

[66] 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.

[67] 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.

[68] Poulsen, K. Software bug contributed to blackout. *SecurityFocus* (2004).

[69] Pozniansky, E., and Schuster, A. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, June 2003), pp. 179–190.

[70] Prvulovic, M., and Torrellas, J. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual*

International Symposium on Computer architecture (San Diego, California, 2003), pp. 110–121.

[71] Purser, Z., Sundaramoorthy, K., and Rotenberg, E. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (November 2000), pp. 257–268.

[72] 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.

[73] 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.

[74] Russinovich, M., and Cogswell, B. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (1996), pp. 258–266.

[75] Sack, P., Bliss, B. E., Ma, Z., Petersen, P., and Torrellas, J. Accurate and efficient filtering for the Intel thread checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (San Jose, CA, October 2002), pp. 34–41.

[76] Salamat, B., Jackson, T., Gal, A., and Franz, M. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the European Conference on Computer Systems* (April 2009).

[77] 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.

[78] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15*, 4 (November 1997), 391–411.

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

[80] Schonberg, E. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, OR, June 1989).

[81] Sen, K., and Agha, G. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing* (Haifa, Israel, 2007), pp. 166–182.

[82] Serebryany, K., and Iskhodzhanov, T. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (December 2009).

[83] Shavit, N., and Touitou, D. Software transactional memory. In *Symposium on Principles of Distributed Computing* (1995), pp. 204–213.

[84] Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. Multiscalar processors. In *Proceedings of the 1995 International Symposium on Computer Architecture* (June 1995), pp. 414–425.

[85] 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 Technical Conference* (Boston, MA, June 2004), pp. 29–44.

[86] Steffan, J. G., and Mowry, T. C. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 1998 Symposium on High Performance Computer Architecture* (February 1998), pp. 2–13.

[87] Süsskraut, M., Knauth, T., Weigert, S., Schiffel, U., Meinhold, M., Fetzer, C., Bai, T., Ding, C., and Zhang, C. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization (CGO)* (April 2010), pp. 131–140.

[88] Intel's Teraflops Research Chip. Tech. rep., Intel, 2007.

[89] 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.

[90] Veeraraghavan, K., Chen, P. M., Flinn, J., and Narayanasamy, S. Surviving and detecting data races using complementary schedules. In *Proceedings of the 2011 Symposium on Operating Systems Principles (SOSP)* (October 2011).

[91] Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P. M., Flinn, J., and Narayanasamy, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

[92] Vlachos, E., Goodstein, M. L., Kozuch, M. A., Chen, S., Falsafi, B., Gibbons, P. B., and Mowry, T. C. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, March 2010), pp. 271–284.

[93] von Neumann, J. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies* (1956), 43–98.

[94] Wallace, S., and Hazelwood, K. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)* (March 2007), pp. 209–220.

[95] Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., and Mahlke, S. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (San Diego, CA, December 2008), pp. 281–294.

[96] Wang, Y.-M., Huang, Y., and Fuchs, W. K. Progressive retry for software error recovery in distributed systems. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing* (Toulouse, France, June 1993).

[97] We, J., Cui, H., and Yang, J. Bypassing races in live applications with execution filters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[98] Weeratunge, D., Zhang, X., and Jagannathan, S. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the 2010 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2010), pp. 155–166.

[99] 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.

[100] Xu, M., Bodik, R., and Hill, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 2003 International Symposium on Computer Architecture* (June 2003), pp. 122–135.

[101] Xu, M., Bodik, R., and Hill, M. D. A serializability violation detector for shared-memory server programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (Chicago, IL, June 2005).

[102] Xu, M., Bodik, R., and Hill, M. D. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 49–60.

[103] Yu, J. Collection of concurrency bugs. http://www.eecs.umich.edu/ jieyu/bugs.html.

[104] Yu, J., and Narayanasamy, S. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (June 2009), pp. 325–336.

[105] Yu, Y., Rodeheffer, T., and Chen, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 221–234.

[106] Yumerefendi, A. R., Mickle, B., and Cox, L. P. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation* (Cambridge, MA, April 2007), pp. 159–172.

[107] Zamfir, C., and Candea, G. Execution synthesis: A technique for automated software debugging. In *Proceedings of the European Conference on Computer Systems* (April 2010), pp. 321–334.

[108] Zilles, C., and Sohi, G. Master/slave speculative parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)* (2002), pp. 85–96.