# Detecting and Surviving Data Races using Complementary Schedules

Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy
University of Michigan
{kaushikv,pmchen,jflinn,nsatish}@umich.edu

## ABSTRACT

Data races are a common source of errors in multithreaded programs. In this paper, we show how to protect a program from data race errors at runtime by executing multiple replicas of the program with complementary thread 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. Our system, called Frost[1], uses complementary schedules to cause at least one replica to avoid the order of racing instructions that leads to incorrect program execution for most harmful data races. Frost introduces outcome-based race detection, which detects data races by comparing the state of replicas executing complementary schedules. We show that this method is substantially faster than existing dynamic race detectors for unmanaged code. To help programs survive bugs in production, Frost also diagnoses the data race bug and selects an appropriate recovery strategy, such as choosing a replica that is likely to be correct or executing more replicas to gather additional information.

Frost controls the thread schedules of replicas by running all threads of a replica non-preemptively on a single core. To scale the program to multiple cores, Frost runs a third replica in parallel to generate checkpoints of the program's likely future states — these checkpoints let Frost divide program execution into multiple epochs, which it then runs in parallel.

We evaluate Frost using 11 real data race bugs in desktop and server applications. Frost both detects and survives all of these data races. Since Frost runs three replicas, its utilization cost is 3x. However, if there are spare cores to absorb this increased utilization, Frost adds only 3–12% overhead to application runtime.

[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.

## Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.4.8 [**Operating Systems**]: Performance

## General Terms

Design, Performance, Reliability

## Keywords

Data race detection, Data race survival, Uniparallelism

## 1. INTRODUCTION

The prevalence of multicore processors has encouraged the use of multithreaded software in a wide range of domains, including scientific computing, network servers, desktop applications, and mobile devices. Unfortunately, multithreaded software is vulnerable to bugs due to data races, in which two instructions in different threads (at least one of which is a write) access the same memory location without being ordered by synchronization operations. Many concurrency bugs are due to data races. These bugs are difficult to find in development and can cause crashes or other program errors at runtime. Failures due to races can be catastrophic, as shown by the 2003 Northeastern United States power blackout [33] and radiation overdoses from the Therac-25 [25].

Researchers have proposed various solutions that attempt to eliminate or identify data races in the development process, including disciplined languages [5, 7], static race analysis [13], and dynamic race analysis [15, 40, 42]. Despite these attempts, data races continue to plague production code and are a major source of crashes, incorrect execution, and computer intrusions.

To help address the problem of data race bugs, we propose running multiple replicas of a program and forcing two of these replicas to follow *complementary schedules*. Our goal in using complementary schedules is to force replicas to diverge if and only if there is a potentially harmful data race. We do this by exploiting a sweet spot in the space of possible thread schedules. First, we ensure that all replicas 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 [37]. Second, while obeying the previous constraint, we attempt to make the thread schedules executed by the two replicas as dissimilar as possible. Specifically, we try 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.

Our system, called Frost, uses 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 methods detect races by analyzing the events executed by a program, outcome-based race detection detects 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 generate identical correct outcomes using multiple non-preemptive schedules (See Section 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 [31] 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 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.

To generate complementary thread schedules, Frost must control tightly the execution of each replica. To do this, Frost timeslices the threads of a replica onto a single processor and switches between threads only at synchronization points (i.e., it uses non-preemptive scheduling). Running threads on a single processor without preemptions has another benefit: it prevents bugs that require preemptions (e.g., atomicity violations) from manifesting, thereby increasing availability. Because running all threads on a single processor prevents a replica from scaling to take advantage of multiple cores, Frost uses *uniparallelism* [44] to parallelize a uniprocessor execution by running multiple *epochs* (time intervals) of that execution 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.

This paper 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 if there are sufficient cores or idle CPU cycles to run all replicas.

## 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 those 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*. A data race bug may involve multiple failure requirements, all of which must be met for the program to fail.

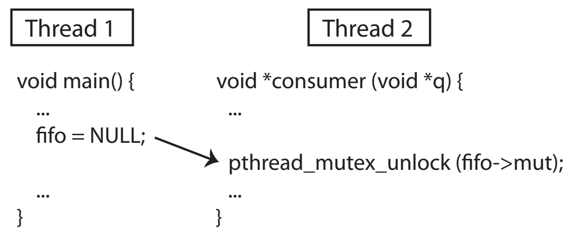
As an example, consider the simple bug in Figure 1(a). If thread 1 sets `fifo` to NULL before thread 2 dereferences the pointer, the program fails. If thread 2 accesses the pointer first, the program executes correctly. The arrow in the figure shows the failure requirement. Figure 1(b) shows a slightly 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 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 3.4 we discuss how generalizing the range of scenarios affects Frost's guarantees.

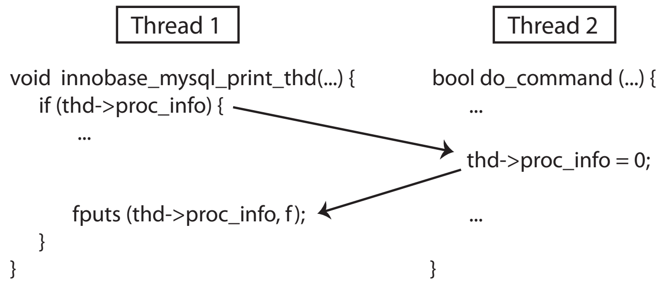
The goal of executing two replicas with complementary schedules is to ensure that one replica avoids the data race bug. We say that two replicas have perfectly complementary schedules if and only if, for every pair of instructions $a$ and $b$ executed by different threads that are not ordered by application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



(a) Type I bug



(b) MySQL #3596: Type II bug

**Figure 1: Data race examples**

complementary schedules. To understand why this is so, consider a canonical schedule with a preemption: 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 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*.

## 3. 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, and 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.

### 3.1 Constructing complementary schedules

Frost divides program execution 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 [37]. 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 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 equals or exceeds the value logged for the 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.

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

Frost uses uniparallelism [44] to achieve scalability. Uniparallelism is based on the observation that there exists 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

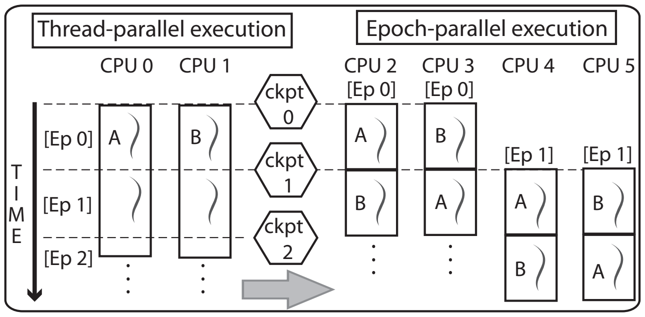


**Figure 2: Frost: Overview**

of uniparallelism in a system called DoublePlay provided efficient software-only deterministic replay [44]. 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 2, 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 [24] 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 executions. 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 executions do 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 [32]) to defer the output. This leads to a trade-off among correctness, overhead, and output latency when choosing how long an epoch should last. Longer epochs offer better correctness properties, as discussed in Section 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, epoch 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 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 threads to proceed.

## 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 this a *self-evident* failure 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 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. During execution, 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 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 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 such 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.

### 3.3.1 *Using the epoch outcome for survival*

Frost diagnoses results by applying 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 1 shows the explanation that Frost associates with each combination of results, and the right

| Epoch Results | Likely Bug | Survival Strategy |
|---|---|---|
| A-AA | None | Commit A |
| F-FF | Non-Race Bug | Rollback |
| A-AB/A-BA | Type I | Rollback |
| A-AF/A-FA | Type I | Commit A |
| F-FA/F-AF | Type I | Commit A |
| A-BB | Type II | Commit B |
| A-BC | Type II | Commit B or C |
| F-AA | Type II | Commit A |
| F-AB | Type II | Commit A or B |
| A-BF/A-FB | Multiple | Rollback |
| A-FF | Multiple | Rollback |

The left column shows the possible combinations 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. We use the same letter when replicas produce identical output and state.

**Table 1: A taxonomy of epoch outcomes**

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).

### *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 system 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 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 [44]. 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+ [34] that replays a divergent epoch to precisely identify the set of racing instructions.

### 3.3.3 Sampling

Some recent race detection tools use sampling to reduce overhead at the cost of missing some data races [6, 14, 29]. 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.

## 3.4 Limitations

Section 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 4.1.2 show, these limitations did not compromise Frost's survival or detection properties in practice when we evaluated Frost with real application bugs.

### 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's diagnosis might explain the epoch outcome but be incorrect for that execution. 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 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 mis-classify 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.

### 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 3. 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 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.

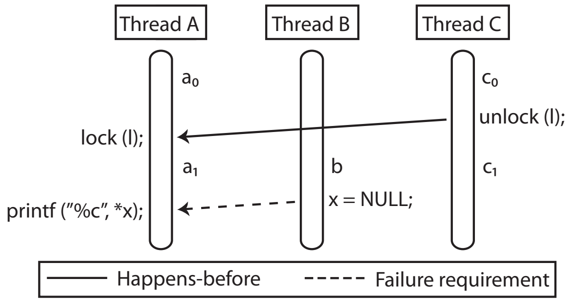


**Figure 3: Priority inversion scenario**

This may prevent Frost from surviving or detecting the race if the failure does not occur in the thread-parallel execution and 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.

#### 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 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 [26].

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 4.2 and not elsewhere in the evaluation.

#### 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. 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?

## 4.1 Detecting and surviving races

#### 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 [50] in Apache, MySQL, and pbzip2 compiled form several academic sources [27, 49, 51] 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 [51]. Finally, during our tests, Frost detected a previously unknown, potentially malign data race in glibc, which we added to our test suite. Table 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 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 execution. The next column shows the percentage of trials in which Frost detects the bug via divergence in replica output or state. The

| 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 2: Data race detection and survival**

final column shows how long Frost takes to recover from the bug — this includes the cost of rolling back and executing new replicas.

### *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. As with the Apache bug, the outputs in the two epoch-parallel replicas are different, but both 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 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 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. 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

| 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 3: Comparison of data race detection coverage**

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.

## 4.2 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 4.1.2 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 4.1.2, 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+ [34]. 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 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.

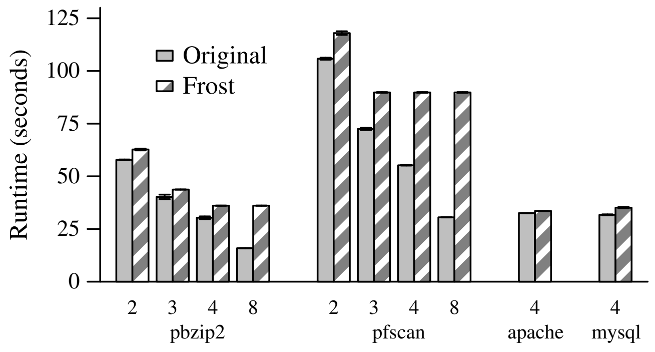
Table 3 compares the coverage of Frost to that of the traditional dynamic race detector. We evaluated each benchmark for the same amount of testing time; the table shows cumulative results for all runs.

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 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 3.4.2. When we executed Frost with the heuristic disabled, it detected all harmful races detected in Table 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.

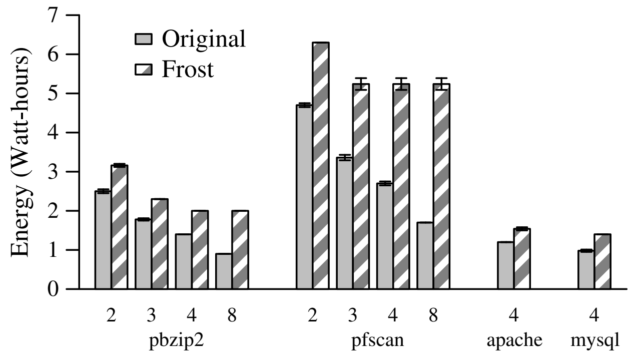
The last two columns in Table 3 list the number of benign races identified by the traditional data race detector and Frost for each benchmark. We manually classified 79 benign races reported by the traditional race detector in the pbzip2, Apache, pfscan and glibc benchmarks, according to a previously-proposed taxonomy [31], 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



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; 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: Execution time overhead**



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; error bars are 90% confidence intervals.

**Figure 5: Energy overhead**

races identified by the traditional race detector were harmful (with most benign races due to custom synchronization in MySQL).

## 4.3 Performance

### 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 for 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 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.

### 4.3.2 Throughput

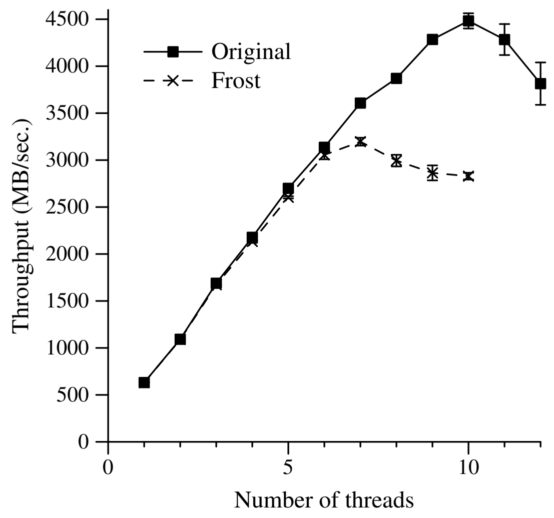
The primary factor affecting Frost's performance for CPU-bound applications is the availability of unused cores. As Figure 4 shows, Frost adds a reasonable 8% overhead for pbzip2 and a 12% overhead for pfscan when these applications use only 2 cores. The reason that Frost's execution time overhead is 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`. 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, Frost only adds 3% overhead for Apache and 11% overhead for MySQL.

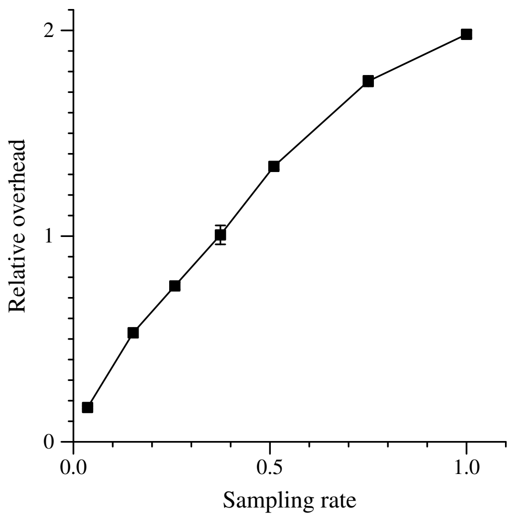
### 4.3.3 Energy use

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.



This figure shows pfscan throughput (MB of data scanned per second) with and without Frost. We vary the number of pfscan worker threads. Results are the mean of five trials; error bars are 90% confidence intervals.

**Figure 6: 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 7: Effect of sampling on relative overhead**

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 5 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.

### *4.3.4 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. 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 6 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 hits 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.

### *4.3.5 Sampling*

As described in Section 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 data races 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.

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 7 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 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 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 [44] 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.

## 5. RELATED WORK

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

### 5.1 Data race survival

The idea of using replication to survive errors dates back to the early days of computing [45, 28]. In active (state-machine) replication, replicas run in parallel and can be used to detect errors and vote on which result is correct [41]. 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 [8]. 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 [47], proactively restarting to eliminate latent errors [20], shrinking the part of the system that needs to be restarted [9], and reducing the cost of running multiple replicas [19].

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 [4, 17, 36], changing the instruction set [2, 22], or even running multiple independently-written versions of the program [1]. Our focus on ensuring at least one correct replica is similar to work in security that creates replicas with disjoint exploitation sets [11, 39].

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 [36, 47] or changing the priority order of processes [36]. 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.

The idea of controlling thread schedules has also been used to explore the space of possible thread interleavings in model checking and program testing [18, 30]. 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 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 [30] 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 as Frost does.

Past research has examined several approaches that do not require active replication for surviving concurrency bugs that cause deadlocks [21, 46]. 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 [12] or requires processor support [51]. Other researchers have observed that some concurrency bugs can be eliminated by minimizing preemptions and providing sequential semantics [3]. Other systems [48] 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.

### 5.2 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., [13]) 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 [23, 42] and lockset analysis [40]. 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 [15] 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 [38] and Google [43], which incur more than 30x performance overhead.

Dynamic race detectors can use language-specific or runtime-specific features to reduce overhead. RaceTrack [52] 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 [10]. 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 spare cores are not 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 [6] 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 [29] 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 epochs. 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 [35]. Data Collider [14] 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 increases 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 [31]. 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 4.2 to implement a complete dynamic race detector. Other tools such as Intel's Thread-Checker [38] provide stack traces for both threads participating in a data race, and some tools, such as RaceTrack [52] can guarantee a stack trace for only one thread.

Pike [16] 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 cononicalized 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 [53] compares the output of a replica with access to sensitive data with that of a replica without such access to detect information leaks.

## 6. CONCLUSION

Frost introduces two main ideas to mitigate the problem 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.

## Acknowledgments

We thank the anonymous reviewers and our shepherd, Tim Harris, for comments that improved this paper. Jie Hou and Jessica Ouyang helped gather data used in the paper. The work is supported by the National Science Foundation under awards CNS-0905149 and CCF-0916770. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, the U.S. government, or industrial sponsors.

## 7. REFERENCES

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

[2] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. 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.

[3] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 81–96, Orlando, FL, October 2009.

[4] E. D. Berger and B. G. Zorn. 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.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 97–116, Orlando, FL, October 2009.

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

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 211–230, Seattle, WA, November 2002.

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

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

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

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

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

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

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

[15] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pages 121–133, Dublin, Ireland, June 2009.

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

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

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

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

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

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

[22] G. S. Kc, A. D. Keromytis, and V. Prevelakis. 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.

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

[24] D. Lee, B. Wester, K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. 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*, pages 77–89, Pittsburgh, PA, March 2010.

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

[26] D. E. Lowell, S. Chandra, and P. M. Chen. 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.

[27] S. Lu, S. Park, E. Seo, and Y. Zhou. 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*, pages 329–339, 2008.

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

[29] D. Marino, M. Musuvathi, and S. Narayanasamy. 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.

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

[31] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. 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.

[32] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the*

*20th ACM Symposium on Operating Systems Principles*, pages 191–205, Brighton, United Kingdom, October 2005.

[33] K. Poulsen. Software bug contributed to blackout. *SecurityFocus*, 2004.

[34] E. Pozniansky and A. Schuster. 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*, pages 179–190, San Diego, CA, June 2003.

[35] M. Prvulovic and J. Torrellas. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer architecture*, pages 110–121, San Diego, California, 2003.

[36] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *ACM Symposium on Operating Systems Principles*, pages 235–248, Brighton, United Kingdom, October 2005.

[37] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[38] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. 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*, pages 34–41, San Jose, CA, October 2002.

[39] B. Salamat, T. Jackson, A. Gal, and M. Franz. 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.

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

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

[42] E. Schonberg. 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.

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

[44] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. 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.

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

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

[47] Y.-M. Wang, Y. Huang, and W. K. Fuchs. 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.

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

[49] M. Xu, R. Bodik, and M. D. Hill. 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.

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

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

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

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