Deterministic, Mutable, and Distributed Record-Replay for Operating Systems and Database Systems

Nicolas Viennot

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2017

©2017

Nicolas Viennot

All Rights Reserved

ABSTRACT

Deterministic, Mutable, and Distributed Record-Replay for Operating Systems and Database Systems

Nicolas Viennot

Application record and replay is the ability to record application execution and replay it at a later time. Record-replay has many use cases including diagnosing and debugging applications by capturing and reproducing hard to find bugs, providing transparent application fault tolerance by maintaining a live replica of a running program, and offline instrumentation that would be too costly to run in a production environment. Different record-replay systems may offer different levels of replay faithfulness, the strongest level being *deterministic replay* which guarantees an identical reenactment of the original execution. Such a guarantee requires capturing all sources of nondeterminism during the recording phase. In the general case, such record-replay systems can dramatically hinder application performance, rendering them unpractical in certain application domains. Furthermore, various use cases are incompatible with strictly replaying the original execution. For example, in a primary-secondary database scenario, the secondary database would be unable to serve additional traffic while being replicated. No record-replay system fit all use cases.

This dissertation shows how to make deterministic record-replay fast and efficient, how broadening replay semantics can enable powerful new use cases, and how choosing the right level of abstraction for record-replay can support distributed and heterogeneous database replication with little effort.

We explore four record-replay systems with different semantics enabling different use cases. We first present SCRIBE, an OS-level deterministic record-replay mechanism that support multi-process applications on multi-core systems. One of the main challenge is to record the interaction of threads running on different CPU cores in an efficient manner. SCRIBE introduces two new lightweight OS mechanisms, rendezvous point and sync points, to efficiently record nondeterministic interactions such as related system calls, signals, and shared memory accesses. SCRIBE allows the capture and replication of hard to find bugs to facilitate debugging and serves as a solid foundation for our two following systems.

We then present RACEPRO, a process race detection system to improve software correctness. Process

races occur when multiple processes access shared operating system resources, such as files, without proper synchronization. Detecting process races is difficult due to the elusive nature of these bugs, and the heterogeneity of frameworks involved in such bugs. RACEPRO is the first tool to detect such process races. RACEPRO records application executions in deployed systems, allowing offline race detection by analyzing the previously recorded log. RACEPRO then replays the application execution and forces the manifestation of detected races to check their effect on the application. Upon failure, RACEPRO reports potentially harmful races to developers.

Third, we present DORA, a mutable record-replay system which allows a recorded execution of an application to be replayed with a modified version of the application. Mutable record-replay provides a number of benefits for reproducing, diagnosing, and fixing software bugs. Given a recording and a modified application, finding a mutable replay is challenging, and undecidable in the general case. Despite the difficulty of the problem, we show a very simple but effective algorithm to search for suitable replays.

Lastly, we present SYNAPSE, a heterogeneous database replication system designed for Web applications. Web applications are increasingly built using a service-oriented architecture that integrates services powered by a variety of databases. Often, the same data, needed by multiple services, must be replicated across different databases and kept in sync. Unfortunately, these databases use vendor specific data replication engines which are not compatible with each other. To solve this challenge, SYNAPSE operates at the application level to access a unified data representation through object relational mappers. Additionally, SYNAPSE leverages application semantics to replicate data with good consistency semantics using mechanisms similar to SCRIBE.

Table of Contents

List of Figures

List of Tables

Acknowledgments

During the past 9 years at Columbia University, I have met inspiring advisors and colleagues who defined who I am as a researcher. My deepest gratitude goes to my advisor, Jason Nieh, for believing in me and making me reach my full potential. He never gave up on me, even when he had all the reasons to. I am thankful to the members of my dissertation committee, especially Roxana Geambasu, Junfeng Yang, and Alfred Aho. Roxana taught me distributed systems design in a rigorous and practical manner. She offered countless hours of guidance both on a professional level and personal level. I could always count on her when I needed a listening ear. Junfeng shared his deep knowledge and expertise in systems which were invaluable in developing my work. Al provided numerous insights to improve this document. I am grateful to the friends and colleagues that helped me finish what I originally thought was impossible to complete. Sid Nair made me a better person by touching me with kindness, softness, and humility. Interacting with Sid has always been a wonderful, even during disagreements. Kareem Kouddous allowed me to implement my research prototype in a large production system he was responsible for. He brought back the self-esteem I had once lost. Jeremy Andrus was consistently available to give me a hand when I struggled getting things done. I have tremendous respect for his sense of discipline and commitment. Christoffer Dall kept me inspired. He became lead developer of the Linux kernel KVM/ARM subsystem thanks to his social, communication, and technical skills. My mentor, Oren Laadan, provided guidance when I started my research. His ability to express complex concepts with great ease is fascinating. I wish I could be as articulated and eloquent as he is. I want to thank all the Columbia University staff who provided the structure allowing me to complete my work. Jessica Rosa, Remiko Moss, Daisy Nguyen, and Cindy Walters, greatly improved my experience. Finally, I thank my friends Valentin Burlacu, Deirdre Campbell, Mathias Lecuyer, Quentin Decock, Aaron Allon, and Laure Theroude for providing many years of support.

This work was supported in part by a Google Research Award, NSF grants CNS-1018355, CNS-

0905246, CCF-1162021, CNS-1162447, CNS-1422909, CNS-1351089, CNS-1117805, CNS-1054906, CNS-1012633, CNS-0914845, CNS-0905246, AFRL FA8650-10-C-7024, AFRL FA8750-10-2-0253, AFOSR MURI FA9550-07-1-0527, and DARPA FA8650-11-C-7190.

À mes parents

Chapter 1

Introduction

Application record and replay is the ability to record application execution and replay it at a later time, possibly on a different host. Record-replay has many use cases including diagnosing and debugging applications by capturing and reproducing hard to find bugs, providing fault tolerance and scaling capabilities to applications by replaying application state on a replica. The desired semantics of record-replay systems varies depending on the use case. For example, when debugging a rarely occurring multi-threaded related bug, a record-replay system that guarantees an identical reenactment of the original execution is useful. Such systems are called *deterministic record-replay* systems as all sources of nondeterminism must be captured during the recording phase. By recording an application on production systems, a developer can capture a rare occurrence of an elusive bug, replay it in his development environment, exactly as it happened, repeatedly, to determine its root cause. While deterministic replay mechanisms are certainly useful, such mechanisms can dramatically hinder application performance, rendering them unpractical in certain application domains. Furthermore, deterministic replay alone can be insufficient for some use cases. A developer may want to inject faults into a replayed execution to observe how the application behaves upon failures, or enable full logging capabilities of the application by changing a configuration file. Here, a record-replay system tolerant to application changes is useful. Other record-replay use cases include database replication. For example, recording a primary database execution and replaying it live on an other replica provides fault tolerance. Upon primary failure, the replica is promoted to primary, and traffic redirected. Deterministic replay alone can provide fault tolerance, but cannot provide any scaling capabilities. If the replica was able to serve read-only queries while being replayed, it could offload some traffic from the primary. Deterministic replay does not allow any new executions to occur, and is thus not suitable for such use case. This

dissertation shows that controlling the emergence of new executions during replay enables powerful new use cases. We show four record-replay systems. We start by presenting a deterministic record-replay system that is fast and efficient, and gradually broaden replay semantics to open the range of use cases that can be supported through our other systems.

First, we present SCRIBE, a multiprocessor deterministic record-replay system. To support a wide range of application, SCRIBE is implemented at the operating system (OS) level, and is transparent to applications. Implementing deterministic execution record-replay can be challenging for a variety of reasons. First, all sources of nondeterminism must be captured and recorded. Sources of nondeterminism can be categorized in 1) data related nondeterminism and 2) timing related nondeterminism. Data related nondeterminism is the easiest to capture. It includes recording all external inputs such as incoming network packets, or user keystrokes. Timing related nondeterminism is harder to capture, especially when the recorded application runs on multiple CPUs simultaneously. Application threads or processes access data and resources in an undefined order, resulting in many different possible schedules. It is sufficient to record all sources of nondeterminism, both data and timing sources, to replay an application execution deterministically. However, doing so with good performance is crucial since recording is done in deployed systems.

To incur minimal record overhead, SCRIBE introduces two new lightweight OS mechanisms, rendezvous and sync points, to efficiently record nondeterministic interactions such as related system calls, signals, and shared memory accesses. For example, when two concurrent processes access the same file, the original order in which each access took place must be preserved when replaying. Instead of recording the kernel scheduling decisions, SCRIBE piggy backs on existing kernel synchronization primitives such as inode mutexes or file descriptor locks, and records in which order these locks are taken by the kernel. We call these rendezvous points, and they make a partial ordering of execution based on system call dependencies sufficient for deterministic replay, avoiding the recording overhead of maintaining an exact execution ordering. During replay, application processes and threads are scheduled in a different order compared to the original ordering. This leads to executions that are different from a kernel perspective, but equivalent from an application perspective, as all interactions between the application's processes and threads are faithfully replicated.

Another difficult problem to achieve deterministic replay is to replay asynchronous interactions that can occur at arbitrary times during the execution. For example, POSIX signal delivery may occur at any point in the application instruction flow. Previous works have developed techniques of instrumenting applications

with the use of hardware counters to precisely locate where a signal was delivered in the instruction flow. SCRIBE takes a different approach by delaying these asynchronous events during the recording to a point where it is much easier to replay deterministically. For example, when an application is being recorded, instead of delivering a signal in the middle of the instruction flow, SCRIBE delivers the signal at the next encountered sync point, such as a system call or a deterministic page fault. In other words, Sync points allow SCRIBE to convert asynchronous interactions that can occur at arbitrary times into synchronous events that are much easier to record and replay. In our evaluation, we show that the introduced delay is imperceptible in an application as sync points occur very frequently.

With multi-threaded applications, replaying accesses to shared memory is difficult. The order in which each thread accesses a shared memory location must be replayed deterministically. To solve this problem, instead of doing any sort of binary instrumentation, SCRIBE leverages the MMU to monitor memory accesses through page faults. SCRIBE implements a concurrent read, exclusive write (CREW) protocol on shared pages. To do so, instead of having all threads share a common page table, threads access memory through a per-thread page table, which SCRIBE uses to record access order with rendezvous points and sync points. At a given point in time, a page may be accessed by only a single thread, the owner. When another thread tries to access that same page, a fault occurs, and an ownership relinquish request is issued to the current owner, which is processed at its next sync point. This lightweight mechanism allows low overhead of recording shared memory interactions.

Our results show for the first time that (1) sync points are an effective, lightweight mechanism for handling nondeterminism due to signals and shared memory, (2) sync points occur often enough in real server and desktop applications that the vast majority of asynchronous events are handled instantaneously, and even when events are deferred, they are delayed for 25 to 220 μs on average, (3) an operating system mechanism can record-replay real multi-threaded and multi-process applications, (4) transparent, low-overhead record-replay can be done for workloads across a wide range of server and desktop applications, including Apache, MySQL, Firefox, Acrobat, OpenOffice, parallel make, and MPlayer. On a 4-CPU multiprocessor, SCRIBE's recording overhead was under 2.5% for server applications, and less than 15% for desktop applications. These results show for the first time a new level of transparent record and replay performance on commodity multiprocessor systems that was not previously possible.

Second, we present RACEPRO, a process race detection system to improve software correctness. Process races occur when multiple processes access shared operating system resources, such as files, without proper

synchronization. To better understand process races, we present the first study of real process races. We study hundreds of real applications across six Linux distributions and show that process races are numerous and a real threat to reliability and security. Detecting harmful races is difficult for three key challenges. The first is scope: process races are extremely heterogeneous. They may involve many different programs. These programs may be written in different programming languages, run within different processes or threads, and access diverse resources. The second challenge is coverage: although process races are numerous, each particular process race tends to be highly elusive. They are timing-dependent, and tend to surface only in rare executions. Arguably worse than thread races, they may occur only under specific software, hardware, and user configurations at specific sites. It is hopeless to rely on a few software vendors and beta testing sites to create all possible configurations and executions for checking. The third challenge is algorithmic: what race detection algorithm can be used for detecting process races? Existing algorithms assume well-defined load and store instructions and thread synchronization primitives. However, the effects of system calls are often under-specified and process synchronization primitives are very different from those used in shared memory.

RACEPRO addresses these challenges with four ideas. First, it checks deployed systems in vivo. While a deployed system is running, RACEPRO records the execution without doing any checking. RACEPRO then systematically checks this recorded execution for races *offline*. By checking deployed systems, RACEPRO mitigates the coverage challenge because all user machines together can create a much larger and useful set of configurations and executions for checking. By decoupling recording and checking, RACEPRO reduces its performance overhead on the deployed systems. Second, RACEPRO uses the application transparent SCRIBE engine to record deployed applications, mitigating the scope challenge, as no application source code or modifications of the checked applications are required. Third, to detect process races in a recorded execution, RACEPRO models each system call by what we call *load and store micro-operations* to shared kernel objects. RACEPRO leverages SCRIBE's rendezvous points to facilitate the modeling of these two operations with low overhead. Because these two operations are well understood by existing race detection algorithms, RACEPRO can leverage these algorithms, mitigating the algorithmic challenge. Fourth, to reduce false positives and negatives, RACEPRO uses *replay and go-live* to validate detected races. A detected race based on the micro-operations may be either *benign* or *harmful*, depending on whether it leads to a *failure*, such as a segmentation fault or a program abort. RACEPRO considers a change in the order of the system calls involved in a race to be an *execution branch*. To check whether this branch leads to a failure, RACEPRO

replays the recorded execution until the *reordered* system calls then resumes live execution. It then runs a set of built-in or user-provided checkers on the live execution to detect failures, and emits a bug report only when a real failure is detected.

This constitutes a departure from deterministic replay. First, RACEPRO replays a modified version of the original execution that includes reordered system calls. Second, after the reordered system calls are replayed, RACEPRO switches from a controlled execution to a live execution. This can be challenging as the live execution may use OS resources referenced during the controlled execution (e.g. file descriptors), making the replay mechanism more complex than SCRIBE's. Despite difficulties, our experimental results show that RACEPRO can detect real bugs due to process races in widespread Linux deployed systems, including several previously unknown bugs in shells, databases, and makefiles. We found the practicality of RACEPRO limited as we obtained best results only with user-provided race checkers as the built-in ones were too rudimentary. An ideal build-in checker would attempt replaying the rest of the original execution past the reordered system calls and measure how much the replayed execution diverged from the original. Replying a modified execution in the general case requires a much more evolved replayer, leading us to DORA.

Third, we present DORA, a record-replay system which allows a recorded execution of an application to be replayed with a modified version of the application. We call this feature *mutable* replay. This feature, not available in previous record-replay systems, enables powerful new functionality. In particular, DORA can help reproduce, diagnose, and fix software bugs by replaying a version of a recorded application that is recompiled with debugging information, reconfigured to produce verbose log output, modified to include additional print statements, or patched to fix a bug. We introduce the concept of mutable replay. Adding a `printf()` call to the replayed application is intuitively safe and the expected outcome clear, but changing thousands of lines of code in the application may incur significant differences from the original execution. Intuitively, a mutable replay system must find a execution that corresponds as much as possible to the original execution. We model the differences of two executions with a user-defined cost function, and provide a generic one that works well in most cases.

DORA consists of three components: (1) a recorder that records application execution to a log similar to the SCRIBE engine, (2) a replayer that can replay a modified version of the application using the log, and (3) an explorer that uses the replayer to find the execution of the modified program that best corresponds to the log file. The recorder logs not only nondeterministic interactions but also deterministic information

such as system call arguments, to allow the replayer to identify when a replay diverges from the original execution early. The explorer evaluates several possible execution paths to find a successful mutable replay. It performs a best-first search for an execution of the modified program that is as close to the original execution as possible according to some cost function. It begins by replaying a recorded execution on a modified program. When the replay diverges from the original execution, the explorer tries to determine why. For example, suppose the modified program made an unexpected `printf()` call. This could be a new call to produce debugging information, or it could simply occur earlier than expected because code was deleted. The explorer chooses the most promising possibility and communicates its decision to the replayer. This process repeats until a successful execution is found.

DORA is designed to handle an wide range of real-world programs, including multi-threaded applications. It can support a broad range of useful application changes, but cannot support arbitrary changes; major changes to the process layout or shared memory layout are not supported. Despite this limitation, DORA is useful in a wide range of real-world use cases for testing, debugging, and validating application changes. In fact, we even found a previously unknown bug in Apache using DORA. DORA's usefulness in practice makes sense given that bug fixes tend to be relatively small and rarely change core application semantics.

Lastly, we present SYNAPSE, an heterogeneous database (DB) replication system specifically designed for Web applications. These Web applications behave very differently compared to traditional single-host applications as they are distributed and have their state contained in databases. Typically, Web applications are comprised of many different services, each implementing a specific feature, using a specific database. For example, the recommendation feature of an e-commerce store can be implemented in a separate service powered by a graph DB, while the store frontend runs on a traditional SQL DB. These services share a common subset of the data; for example, the recommendation feature would share the product and user data with the store frontend. Application state modifications consist of database primitive changes, such as a node insertion in a graph DB, or a row update in a SQL DB. Designing a system that allows this common data subset to be synchronized across all the different DBs is challenging for four reasons. First, it should be compatible with a vast number of DBs, whose layouts and engines may be completely different. Second, it should be easy to use: orchestrating the data flows inside the internal eco-system of services should be seamless for developers. Third, it should provide good consistency guarantees at scale, and fourth the replication mechanism should be failure tolerant. Specifically, network partitions should not result in having half of the DBs missing some data.

Using DORA to record the source DB transparently and replying its execution on the destination DB would not work due to large differences between the two DB systems as DORA would fail at matching a stream of system calls. Instead of operating at the kernel level, SYNAPSE operates at the application level. On a high level, SYNAPSE records application state modifications at the source application, and replay these modifications at the destination application, making SYNAPSE a mutable replay engine for Web applications. Typically, Web applications are structured following the model-view-controller (MVC) pattern. Data is accessed following an object-oriented abstraction with *Models*, such as a `User` class with attributes such as `email` and `name`. Models are implemented on top of Object/Relational Mappers (ORMs). The ORM does the heavy lifting of interacting with the DB so developers do not have to write DB queries. Over the years, many ORMs have been developed, each one targeting a different DB. Thankfully, all these ORMs expose a similar API to developers to interact with the DB. For example, invoking `User.create()` would create a new user, regardless of the combination ORM/DB. SYNAPSE interposes on these ORMs to monitor accesses to data objects. This allow SYNAPSE to replicate data from one DB to another, effectively replicating application state, without developer intervention and with little DB-specific code. Further, SYNAPSE provides an easy-to-use API to describe data flows. Developers simply annotate their models with SYNAPSE's `publish` and `subscribe` keywords to connect data models together. Despite this simple API, developers can describe complex eco-systems of services. SYNAPSE provides causal consistency delivery semantics by transparently intercepting and ordering all read and write queries to the DB in a similar fashion to SCRIBE's rendezvous points. Finally SYNAPSE provides a fault-tolerant replication mechanism by implementing two-phase commits all the way from the source DB to the destination DB.

We have implemented SYNAPSE for Ruby-on-Rails. We present some experimental data showing that SYNAPSE scales well up to 60,000 updates/second for various workloads. We and others have built or modified 14 Web applications to share data with one another via SYNAPSE. Those built by others have been deployed in production by a startup, Crowdtap. The applications we built extend popular open-source applications to integrate them into data-driven ecosystems.

The development of SCRIBE, RACEPRO, DORA and SYNAPSE led to the following novel contributions:

1. We introduce SCRIBE, the first operating system mechanism to provide transparent, deterministic execution record and replay of multi-threaded and multi-process applications on commodity multiprocessors and operating systems.

2. We introduce rendezvous points to record partial ordering of execution with low overhead. Ren-

dezvous points piggy back on existing kernel synchronization primitives such as inode mutexes or file descriptor locks, and records in which order these locks are taken by the kernel.

3. We introduce sync points to convert difficult to record asynchronous events into easy to record synchronous events. Asynchronous events that occur at arbitrary times are delayed to the next encountered sync point such as a system call. We show that this solution is effective, and does not require hardware counters.

4. We implement a prototype of SCRIBE. Our evaluation shows for the first time that an operating system mechanism can correctly and transparently record and replay multi-process and multi-threaded applications on multiprocessors with low overhead.

5. We provide strong empirical evidence that real server and desktop applications perform frequent operating system activities which can serve as sync points. The introduced delay from using sync points is imperceptible in applications as sync points occur very frequently.

6. We present the first study of real process races. We study hundreds of real applications across six Linux distributions and show that process races are numerous and a real threat to reliability and security.

7. We present RACEPRO, the first system for automatically detecting process races beyond TOCTOU and signal races. It checks deployed systems in vivo by recording live executions which are then used offline for race detection.

8. We show that previously known thread race detection algorithms can be reused to detect process races. Using our record-replay system, operating system resources can be mapped to memory accesses by modeling system calls as micro-operations, on which such algorithms can perform race detection. We show how to detect three different types of process races, load-store races, wait-wakeups races, and wakeup-waits races.

9. We implement a RACEPRO prototype and demonstrate the effectiveness of our approach by performing process race detection in real applications. We show that our system can detect 10 real bugs due to process races in widespread Linux distributions, including several previously unknown bugs in shells, databases, and makefiles.

10. We introduce the concept of mutable replay. That is, we define what can be considered a desirable outcome when replaying an application recording on a modified version of the application.

11. We introduce DORA, the first transparent mutable record-replay system.

12. We introduces an explorer that directs the replay mechanism to identify a mutable replay of the modified application that minimizes differences with the original unmodified application execution. We show that a simple explorer using a best first search algorithm can be effective to perform mutable replay.

13. We provide a few useful properties of DORA. An example of such useful property is the following: if all explored mutations are safe, that is, any addition that does not change any state which is read by the original execution, DORA deterministically replays all events in the original execution with the modified program.

14. We implement a DORA prototype and show that mutable replay is feasible across a wide range of real-world applications and application changes which can reach thousands of lines of code, even without support for major changes to core application semantics.

15. We show that mutable replay is useful for enabling common debugging techniques not possible with previous record-replay systems. We also show that mutable replay enables validation of security patches against both exploits and production workloads. This is all accomplished without requiring source code modifications and with low recording overhead, enabling usage on production systems.

16. We present SYNAPSE, an easy-to-use, strong-semantic, heterogeneous database replication system specifically designed for large-scale Web applications in a service-oriented architecture. These applications run on top of their own databases, whose layouts, and engines can be completely different, and incorporate read-only views of each others' shared data.

17. We introduce a replication mechanism to synchronizes these heterogeneous views in a scalable and consistent manner. SYNAPSE leverages the high-level data models in popular MVC-based Web applications to replicate data across heterogeneous databases. It also leverages application controllers to support application-specific consistency semantics without sacrificing scalability.

18. We implement SYNAPSE for Ruby-on-Rails, show that it provides good performance and scalability, release it on GitHub, and deploy it in production to run the web services for a company.

This dissertation is organized as follows. Chapter §2 presents SCRIBE, our deterministic record-replay system which serves as a foundation for our two next systems. Chapter §3 presents RACEPRO that extends SCRIBE to perform process race detection. Chapter §4 presents DORA that further extends SCRIBE to perform mutable replay. Chapter §5 presents SYNAPSE, our database replication system. Finally, we present some conclusions and directions for future work in Chapter §6.

Chapter 2

SCRIBE: Transparent Deterministic Record-Replay

2.1 Introduction

Deterministic application record and replay is the ability to record application execution and deterministically replay it at a later time. Record-replay has many potential uses, including diagnosing and debugging applications by capturing and reproducing hard to find bugs, dynamic application analysis by performing costly instrumentation on replicas that replay application behavior recorded on production systems, intrusion analysis by capturing intrusions involving nondeterministic effects, and fault-tolerance by providing replicas that replay execution and at the occurrence of a fault, go live in place of the previously running application instance.

Many approaches have tried to provide record-replay functionality, but have suffered from fundamental limitations that make them unusable in many cases. First, most approaches only support replaying the recorded application execution, and do not allow the replayed instance to go live and continue normal execution. This only works for simple debugging uses. It does not work for most scenarios, including any form of debugging that requires the replayed instance to go live, such as debugging past the end of a recorded execution, fault-tolerance which requires the replayed instance to be able to go live when the primary fails.

Second, previous approaches either require application changes or rely on specialized hardware that is not always available. Approaches requiring application changes impose a recurring development cost on each application to provide record-replay, and do not work for unmodified applications. Approaches

requiring specialized hardware rely on either hardware architectures that exist in simulation only, or assume the availability of certain performance counters to track the precise timing of asynchronous events. Such performance counters may not be available on all hardware platform (e.g. ARM).

Third, previous application transparent approaches either do not support multiprocessor systems at all, or require using a virtual machine monitor (VMM) and suffer significant performance overhead on multiprocessor systems. This overhead is imposed on the recording of execution and can result in more than an order of magnitude reduction in application performance [42]. Such overhead is unacceptable even for debugging or analysis. Because the recording must often be done on a production system to capture and identify real bugs for debugging or real application behavior for analysis, minimizing recording overhead is crucial to avoid any adverse impact on production application execution.

To address these problems, we introduce SCRIBE, the first system to provide transparent, low-overhead application record-replay and the ability to go live from replayed execution. SCRIBE uniquely combines transparency and low-overhead for application execution recording based on two principles. First, SCRIBE primarily operates at the well-defined interface between applications and the operating system to record and replay the execution of multiple processes and threads in a consistent and coordinated manner. Using a standard interface that applications already use avoids the need to modify applications to enable record-replay, providing transparency. Using a higher-level interface avoids the need to track and record low-level hardware and operating system nondeterministic effects that have no impact on enabling deterministic application replay, reducing overhead. Unlike VMM approaches, it also enables finer granularity per application record-replay as opposed to limiting record-replay to an entire operating system instance. Second, SCRIBE observes that real applications do frequent system activities such as I/O. These activities can be recorded efficiently with relative ease because their timing is synchronous with the execution of the process performing the activities. Using these activities, SCRIBE converts nondeterministic asynchronous interactions that are difficult to record and replay efficiently without additional hardware support into synchronous interactions that can be recorded in software with low overhead. In other words, the timing of an application execution may be perturbed in a manner that makes it easier to record efficiently without sacrificing correctness or performance.

Using these principles, SCRIBE introduces two novel mechanisms to address the key challenge of handling nondeterministic execution. First, SCRIBE introduces *rendezvous points* to record all nondeterministic interactions between applications and the operating system that involve system calls. To be able to go live

at any point during replay, the effect of system calls inside the kernel must be replayed; replaying just the outcome of system calls in user space is not sufficient. SCRIBE does not aim to replay the exact scheduling order as in the original execution, but instead uses rendezvous points to make a partial ordering of execution based on system call dependencies sufficient for deterministic replay. Since an exact execution ordering is not needed, SCRIBE does not incur the associated recording overhead and does not need hardware counters used to maintain such an ordering. SCRIBE also logs input data delivered through system calls to account for nondeterminism due to external input.

Second, SCRIBE introduces *sync points* that correspond to synchronous system events such as system calls and certain page faults to deterministically record the timing of nondeterministic events like signals and shared memory interleavings. For a target process or thread, an asynchronous event such as a signal or shared memory access by other processes or threads may occur at any time during the target process's execution. This is hard to replay since the event must be replayed at the exact same instruction in the target process as during recording. SCRIBE defers asynchronous events until sync points occur to make their timing deterministic so that they are easier to efficiently record and replay. Sync points do not require hardware counters or application modifications that are necessary with previous approaches, and do not adversely impact application performance because they occur frequently enough in real server and desktop applications due to operating system activities.

SCRIBE fully supports record and replay of real multi-process and multi-threaded applications, and enables an application to switch from being replayed to running live at any point in time. SCRIBE accomplishes all of this in an application transparent manner, does not require changing, relinking, or recompiling applications, libraries, or operating system kernels, does not require any specialized hardware support, does not require a VMM or incur its associated costs, and works on commodity multi-core and multiprocessor hardware and operating systems.

We have implemented a SCRIBE Linux prototype and evaluated its performance on multi-core and multiprocessor systems on a wide range of real applications. Our results show for the first time that (1) sync points are an effective, lightweight mechanism for handling nondeterminism due to signals and shared memory, (2) sync points occur often enough in real server and desktop applications that the vast majority of asynchronous events are handled instantaneously, and even when events are deferred, they are delayed for 25 to 220 μs on average, (3) an operating system mechanism can record-replay real multi-threaded and multi-process applications, (4) transparent, low-overhead record-replay can be done for workloads across a wide

range of server and desktop applications, including Apache, MySQL, Firefox, Acrobat, OpenOffice, parallel make, and MPlayer. On a 4-CPU multiprocessor, SCRIBE's recording overhead was under 2.5% for server applications, and less than 15% for desktop applications. These results show for the first time a new level of transparent record and replay performance on commodity multiprocessor systems that was not previously possible.

This chapter is organized as follows. §2.2 presents an overview of SCRIBE's architecture. §2.3 describes how system calls are recorded and replayed §2.4 describes how shared memory interleavings are recorded replayed. §2.5 covers the *rendezvous* event used for ordering access to shared resources. §2.6 describes signal delivery and the recording of asynchronous events. §2.7 presents experimental results. §2.8 discusses related work. Finally, §2.9 presents a summary and concluding remarks of this chapter.

2.2 Architecture Overview

SCRIBE can record and replay the execution of a group of processes and threads from any point in time. We refer to a group of processes and threads being recorded or replayed as a *session*. SCRIBE checkpoints the session at a desired starting time and records the execution going forward. It can then restart and replay the session from the checkpoint. Checkpoints can be taken at any time, replay can be done at any later time as well as on another machine, and replayed execution can go live at any time and continue normal execution. SCRIBE's checkpoint-restart mechanism provides a consistent checkpoint of process and filesystem state based on Zap [59; 60; 81]. We only consider execution replay on a machine with the same CPU type and features, such as x86 MMX/SSE instructions, as where the execution was recorded. For example, a process that uses MMX instructions when it is recorded cannot be replayed on a machine without MMX instructions. We will use Linux semantics to describe how record-replay is accomplished in further detail.

SCRIBE can start recording execution from a checkpoint of a session, or it may begin with an empty session by launching a new process. To begin recording, a dedicated monitor process attaches itself to the target process(es), setting a special *recording* flag for each process to indicate that it is being recorded. This flag is inherited via the `fork` and `clone` system calls, so that new threads and children of a recorded process will automatically become part of the recorded session. Recording takes place in the context of the recorded process. SCRIBE uses stubs to interpose on key operating system kernel entry points to perform some processing before and after the entry points as needed. For instance, when a recorded process executes a system

Event	Description	Payload
hw_inst	hardware instruction trap	op-code and data, e.g. `RDTSC` and counter value
syscall_ret	system call return	system call return value
copy_data	data transfer to/from user space	size and contents of data transfer
page_public	make page public (not owned)	page address
page_share_read	make page shared read-only	page address, page sequence number
page_own_write	make page owned read/write	page address, page sequence number
rendezvous	resource synchronization	resource sequence number
signal_receive	process received signal	signal number and info, whether or not in system call
async_reset	force a sync point	process user space signature at forced sync point

Table 2.1: **SCRIBE Record-Replay Events.**

call, SCRIBE produces events that describe the system call and its outcome by recording information about the system call before and after the system call executes. SCRIBE records by intercepting all interactions of processes with their environment, capturing all nondeterminism in *events* that are stored in *log queues* inside the kernel. SCRIBE allocates a private kernel log queue for each process. Processes generate events during recording, and append them to the log queue. As processes fill their log queues with events, the monitor pulls them from the queues and saves them to permanent storage. Recording of a process ends when the process exits, or when SCRIBE explicitly tells the monitor to stop the recording.

SCRIBE can start replaying a session from the beginning of its execution or from a restarted session. To replay, the monitor launches a new process, or restarts the desired session from the respective checkpoint and marks all processes with a special *replaying* flag. A log queue is allocated for each process. Thereafter, the monitor reads the recorded events from storage and places the data in the respective log queues.

The recorded events are consumed from the log queues to steer the processes to follow the same execution paths they had during recording. Replay takes place in the context of the process being replayed. SCRIBE uses the same stubs for replay as it did for recording to take control over process execution by interposing on key operating system kernel entry points. For example, when a system call is invoked at replay, SCRIBE consumes an event from the log queue to determine how to correctly replay the effect of the system call. Replay of a process ends when the process terminates, or when all the recorded events have been consumed.

SCRIBE can also let the session *go live*, transitioning it from controlled replay to live execution, by

detaching the monitor from the processes and flushing all remaining events. To do this, SCRIBE must do two things to ensure that the replayed session is always in a state that allows it to transition to live execution. First, SCRIBE needs to not only replay the application state in user space, but also the corresponding state that is internally maintained by the operating system on the application's behalf. Second, SCRIBE must ensure that the replayed processes perceive the underlying system to be the same as at the time of recording. System identifiers such as process IDs and network port numbers must be perceived by processes to remain the same for them to run correctly after they transition to live execution. To guarantee this even if the underlying system has changed, SCRIBE uses operating system virtualization [81] to encapsulate processes in a virtual execution environment that provides the same private, virtualized view of the system when the session is replayed or goes live as when it was recorded. Processes only see virtual identifiers that always stay the same. Virtual identifiers are transparently remapped by the environment to real operating system resource identifiers, and the mappings are updated so that the session can go live at any time.

The events that SCRIBE records and replays each contains two fields: the event type, and a payload whose size and contents depend on the event in question. Events are not timestamped because SCRIBE does not replay based on explicit event timing information, and does not aim to repeat the exact scheduling order as in the original execution; rather, it ensures that events are ordered correctly by tracking dependencies among events. Two events are *related* if they access the same resource and at least one of them modifies it, for instance a `write` and `read` on a pipe. SCRIBE tracks dependencies to preserve the partial order of related events during replay.

Table 2.1 lists all event types recorded and replayed by SCRIBE. These events correctly account for all sources of nondeterministic execution that are needed to support deterministic replay: nondeterministic machine instructions, system calls, signals, and shared memory interleavings. External input is also a source of nondeterminism, but this occurs through system calls.

The *hw_inst* event is used for nondeterministic machine instructions which interact directly with the hardware and bypass the operating system. There are three such instructions on x86 CPUs. They all involve reading CPU counters and can be recorded by simply trapping when they occur. While trapping is expensive, these instructions typically occur infrequently; standard binary instrumentation techniques can be used to optimize performance. When a nondeterministic machine instruction occurs, SCRIBE records a *hw_inst* event whose payload is the instruction type and its result. For example, when the `RDTSC` instruction occurs, SCRIBE records a *hw_inst* event whose payload is the `RDTSC` instruction opcode and the 64-bit value of the

timestamp counter. During replay, SCRIBE returns the recorded result instead of executing the instruction.

2.3 System Calls

System calls are the predominant form for processes to interact with the environment and with other processes. System call interposition is used to record and replay the execution of system calls. Unlike other approaches [50; 96; 102], SCRIBE does not simply feed processes with logged data to simulate the effect of system calls. This is not sufficient to enable replayed execution to go live. Instead, SCRIBE re-executes system calls during replay to ensure that the corresponding in-kernel state of a replayed session is updated properly so that it can transition to live execution at any time. We first describe the basics of how SCRIBE handles system calls, then describe in §2.5 how SCRIBE handles nondeterminism due to system calls that access shared resources.

2.3.1 Record

During recording, SCRIBE always allows each system call to execute and records its return value using the *syscall_ret* event so that the same values can be returned on replay. The system call number is not recorded since it will be available on replay when the process executes the same system call. For system calls that create and terminate processes and threads, namely `fork`, `clone`, and `exit`, SCRIBE also sets up the log queues, arranges to control the execution of new processes and threads when they are created, and performs proper cleanup as processes and threads exit. For system calls that transfer nondeterministic or external data from kernel to user space, SCRIBE records the data to the log queue of the calling process using the *copy_data* event so that the same data can be output by the system call on replay. For example, Figure 2.1 shows the recording of `gettimeofday`, which outputs to a data structure a time value which must be recorded. Similarly, all external input data, including network inputs and data from special devices such as `/dev/urandom`, are delivered via system calls, mainly the `read` system call, and must be recorded.

Data from user space used as input for system calls never needs to be recorded; it is always deterministic on replay since it resides in the address space of a replayed process. The only exception is if a buffer corresponds to mapped I/O memory, whose contents are logged as well using the *copy_data* event. Similarly, SCRIBE does not record input from file descriptors that refer to a local filesystem or to objects such as pipes, because their state and contents during replay are controlled by the replay and therefore deterministic. In

Record (action) →	**Event log** →	**Replay (action)**
ret = gettimeofday(K, NULL)		(do nothing)
(system call returned)	*syscall_ret(ret)*	
copy out: K→u (size)	*copy_data(size, K)*	copy out: K→u (size)
return(ret)		return(ret)

Figure 2.1: **Record-Replay of `gettimeofday`.** To record, SCRIBE invokes the system call with an in-kernel buffer (K), logs the return value and input data, copies the data to the user buffer (u) and returns. To replay it copies the logged data to the user space buffer and returns the logged return value.

contrast, approaches that use system call simulation must explicitly log such data, significantly inflating the resulting log size.

2.3.2 Replay

During replay, SCRIBE replays the system calls of each process independently, unless system calls access shared resources as discussed in §2.5. When a replayed process invokes a system call, SCRIBE intercepts it and uses the return value from the corresponding *syscall_ret* event in the log queue of the calling process as the return value of the system call. It does not return the value from executing the actual system call, which may differ and result in the replay diverging from the recorded execution.

For system calls that transfer nondeterministic data from kernel to user space, the data logged in the corresponding *copy_data* event is also returned on replay. For example, Figure 2.1 shows the replay of the `gettimeofday` system call from an event log.

Beyond dealing with the return value and nondeterministic data, we can classify system calls into two categories: idempotent and non-idempotent. Idempotent system calls do not modify the internal kernel state, and therefore the underlying call does not even need to be executed. These system calls typically query resource identifiers, such as `getpid`, `getppid`, `getuid` and `getgid`, or transfer data about resources, such as `uname`, `getrusage`, `time`, `getitimer`, `gettimeofday`, and `sysinfo`. Non-idempotent system calls modify system state and therefore replay typically requires executing the underlying system call.

The processing of non-idempotent system calls varies for different system calls. Most of these system

calls are replayed directly by executing them. Examples include `setsid`, `brk`, reading from and writing to a pipe, etc. By executing these system calls, SCRIBE guarantees that the state of all the resources that belong to the session is correct at all times, and the session may safely stop replaying, go live, and proceed to execute normally. If a system call execution that was successful in the original application execution fails during replay, SCRIBE aborts the replay.

For system calls that create and terminate processes and threads, namely `fork`, `clone` and `exit`, SCRIBE sets up the log queues, arranges to control the execution of new processes and threads when they are created, and performs proper cleanup as processes and threads exit. When creating processes and threads during replay, SCRIBE relies on the underlying virtual namespace to provide a method to select predetermined virtual process identifiers so that processes can reclaim the same set of virtual resource identifiers they had used during recording. The same is true for other system calls that allocate resources with identifiers assigned by the kernel, such as IPC identifiers.

For system calls that carry out external I/O, the internal state of file descriptors, such as file position, is updated even though data may not be explicitly sent or received through the file descriptors. For example, for external input, SCRIBE replays the data to the application from the log rather than fully execute the system call.

For system calls that accept wildstar (catch-all) arguments, such as `mmap` and `wait`, SCRIBE already knows the outcome of the system call, e.g., which address or process was selected. For deterministic replay, it simply substitutes that outcome for the wildstar argument.

2.3.3 Go Live

In most cases, executing recorded system calls during replay is sufficient to automatically replay the kernel state correctly, due to the deterministic behavior of the application and the operating system. For example, when an application creates and then writes to a pipe, the kernel internally allocates a pipe object, populates the process's file table with suitable file descriptors, and then places data in the pipe's internal buffer. During replay, the application will issue the same system calls, in the same partial order, and the kernel will deterministically behave in the same way and reconstruct the same internal state.

However, internal kernel state that is related to external entities is unique in that it interacts with, and is affected by, state that is not controlled by the session. How such state is handled is predicated on what is assumed about the external environment when a session goes live. We identify two scenarios: *stand-*

alone execution assumes that the original external links are non-existent, e.g. in debugging use case, and *switch-over* execution assumes that they remain as is, e.g. for replica execution.

In stand-alone replay, internal kernel state linked to outside the session would become meaningless once the session goes live. Thus, SCRIBE needs to cast meaningful state that gracefully reflects the new status of the resources it represents. It does so by partially executing select system calls that create or manipulate this state.

To illustrate this concept, consider network connections created via `connect` and `accept` system calls. Since `connect` attempts to create a connection to the external world, SCRIBE skips its invocation during stand-alone replay. A successful `accept` will receive an incoming connection into a new socket. To replay this, SCRIBE creates a new, disconnected, socket instead. The end result in both cases, is that the socket remains closed; should the application thereafter go live, it will perceive a network disconnect upon the next attempt to read or write the socket.

Switch-over replay, on the other hand, introduces two additional complexities. First, it requires that the transition to live execution occur transparently, in a way that external entities, such as remote connections, would not notice. Second, replaying of kernel state is no longer deterministic, since it is affected by the interaction with external entities, e.g. the random choice of sequence numbers for TCP connections, and interleaved order of incoming messages.

SCRIBE's approach is to maintain a compatible, but not necessarily identical, internal kernel state during replay. This avoids numerous intricacies involved in identifying, recording and replaying nondeterministic events of, for instance, the network stack. A key observation is that the replay does not interact with the real world until it goes live. It is permissible to have differences in the internal state, provided that when the transition to live execution takes place, the state is consistent with what was previously published to, and hence expected by, the external world.

Consider, for instance, internal kernel state that corresponds to network communication. For protocols that lack reliability guarantees, such as UDP, SCRIBE need only maintain the corresponding network end-point, and may safely ignore buffered or in-transit data. For connection-oriented protocols, like TCP, it records important events that permanently affect the internal state. This includes selection of port number, updates of sequence numbers and timestamps, setup of timer expirations, and acknowledged received data. Unacknowledged received data is not tracked since it will be retransmitted. Data in the send buffers is not logged either because it will be deterministically reproduced by replaying the application.

2.4 Shared Memory

Replaying shared memory interleaving is critical for deterministic replay, especially on multiprocessor machines. Memory sharing happens either explicitly when multiple processes share a common shared mapping, or implicitly when the entire address space is shared, e.g. with threads. The main tool to monitor and control memory access in software is the page protection mechanism. Replaying the order of memory accesses efficiently in software is fundamentally difficult since one process may access shared memory asynchronously with, and at any arbitrary location within, another process's execution.

SCRIBE addresses this problem by introducing page ownership management. Because of spatial and temporal locality, a process typically accesses multiple locations on a page during a given time interval. If we can guarantee that no other processes modify that page during the same time interval, then the page can be treated like private memory for that process during that interval. There would be no need to track memory accesses since there are no nondeterministic shared memory interleavings. This scheme requires a protocol to manage page ownership transitions, and a method to ensure that such transitions occur at precisely the same location in the execution during both record and replay. The latter problem is the key challenge, and is discussed in §2.6.

SCRIBE employs a concurrent read, exclusive write (CREW) protocol [34; 64] for shared memory access, but with additional optimizations. A state field of a page indicates whether it is un-owned (*public*), owned exclusively for read and write (*owned_write*) or shared for read by one or more processes (*shared_read*). A process that owns a page exclusively has its PTE set as read and write. A process that shares a page has the PTE set to read-only. Otherwise the respective PTE will remain invalid to prevent access. A page that is shared for reading continuously tracks its list of readers, and an exclusively owned page tracks its writer (owner).

Transitions between the page states are as follows. A *public* page becomes *shared_read* or *owned_write* on the first read or write access, respectively. An *owned_write* page becomes *shared_read* when another process attempts to read from it; the owner process will give up exclusive access and downgrade its PTE to be read-only. An *owned_write* page can also change owner, in which case the old owner will give it up and invalidate its own PTE, while the new owner will adjust its PTE accordingly. A *shared_read* page becomes *owned_write* when the page is accessed for writing. Finally, a page becomes *public* when all processes that have a right to access terminate.

Transitions between page states occur as a result of page faults, which indicate that the faulting process

Record (action) →	**Event log** →	**Replay (action)**	**User-space**
copy in: u→K (size)		copy in: u→K (size)	*(A) write(fd, u, size)*
rendezvous(A, fd.inode)	*(A) rendezvous(SEQ)*	rendezvous(A, fd.inode)	
ret = write(fd, K, size)		ret = write(fd, K, size)	
(system call returned)	*(A) syscall_ret(ret)*	(system call returned)	
return(ret)		return(ret)	
...	...	...	...
rendezvous(B, fd.inode)	*(B) rendezvous(SEQ+1)*	rendezvous(B, fd.inode)	*(B) read(fd, u, size)*
ret = read(fd, K, size)		ret = read(fd, K, size)	
return ret	*(B) syscall_ret(ret)*	return ret	
copy out: K→u (size)		copy out: K→u (size)	
return(ret)		return(ret)	

Figure 2.2: **Rendezvous Points.** Process A must pass a rendezvous point before it can invoke the system call in both record and replay. Process B must do so too, but with a larger sequence number, thus preserving their relative order. The data itself is deterministic and not logged.

is requesting access to a given page. SCRIBE employs two optimizations to reduce the occurrence of these faults. First, SCRIBE optimizes for the common memory access pattern of reading then writing the same, or nearby, memory addresses. For this pattern, the standard CREW protocol incurs two page faults: a read fault makes the page *shared_read* and a write fault makes it *owned_write*. To avoid this cost, SCRIBE marks pages when they experience a double fault by the same process. A marked page will transition directly to *owned_write* on subsequent page faults, whether they are reads or writes. Finally, SCRIBE clears the flag if the number of page faults exceeds a defined threshold, to adjust its behavior for possibly changing memory access patterns.

Second, SCRIBE optimizes to reduce frequent transfers of page ownership. This can occur among multiple threads due to true or false data sharing. Such page ping-ponging can cause thrashing, especially when multiple pages are involved. For instance, a thread that uses two pages repeatedly in a tight loop may lose ownership of one page while faulting on the other. To mitigate this, SCRIBE defines a minimal ownership retention interval that begins with an ownership change. Ownership transitions are disallowed until the interval expires. The length of the interval is comparable to a standard scheduler time quantum so that a running process is likely to complete its scheduled time quantum of work.

SCRIBE's page ownership management mechanism requires updating PTEs. To support threads without high overhead due to TLB invalidations, we use private page tables to track thread shared memory accesses. All threads associated with a process share a common page table for reference, but each thread uses its own private page table. The reference page table maintains the current state of all pages. When a thread causes a page fault, SCRIBE consults the corresponding entry in the reference page table and copies the PTE to the thread's private table. This is inexpensive because it only flushes a single TLB entry on the local CPU, instead of a costly inter-processor interrupt followed by a global TLB flush. The reference page table is explicitly updated when the process's address space layout is modified, e.g. through `mmap`, `munmap` and `mprotect`. For a single thread, the private page table directly mirrors the reference page table.

2.5 Rendezvous Points

System calls that access shared resources may cause nondeterminism arising from the order of the execution of *related* system calls that access the same resource and at least one of them modifies it. For instance, a `write` and a `read` on the same pipe are related. The order in which related system calls occur needs to be recorded so they can be deterministically replayed. It is crucial to do this in a way that does not degrade performance and scalability on multiprocessor systems.

By operating at the system call level, SCRIBE introduces a novel mechanism to address this problem by capturing concurrency at the same granularity as the operating system. The only requirement is to record and replay the order of any two related system calls. Related system calls occur from accessing shared resources. We observe that the operating system kernel must already provide locations where access to shared kernel objects is serialized for correctness. SCRIBE can thus mimic these serialized access points to record and replay the order of any two related system calls.

SCRIBE introduces *rendezvous points*, locations at which system call ordering is tracked during recording and enforced during replay. Because related system calls occur from accessing shared resources, SCRIBE synchronizes access to each such resource by converting all locations in which shared resources are accessed into rendezvous points. Since these locations are already used by the kernel to serialize access to an instance of a shared resource, rendezvous points do not reduce the scalability or concurrency of the kernel. Our approach avoids the overhead of maintaining an exact execution ordering of system calls and makes a partial ordering of execution based on system call dependencies sufficient for replay.

Rendezvous points are recorded by associating each resource instance with a wait queue and a unique sequence number counter. At any time, exactly one process may be executing inside a given rendezvous point, while others must block until the resource is released. During recording, a process that attempts to access a shared resource will first pass through the corresponding rendezvous point. By doing so, it will increment the sequence number and generate a matching *rendezvous* event. The sequence number in the *rendezvous* event indicates the exact access order for the resource, which can be used to enforce the order during replay. Figure 2.2 shows the recording of the `write` and `read` system calls using rendezvous points and the resulting log. Ideally, for each rendezvous point, we could reuse the respective kernel locking primitive already in place for the associated resource, but this involves kernel changes. To avoid changing the underlying kernel, SCRIBE resorts to its own, separate mutex to interpose transparently at well-defined kernel entry points. §2.7 shows that our approach incurs low overhead on real applications.

During replay, SCRIBE replays the system calls of each process independently from other processes until reaching a rendezvous point. SCRIBE repeats the order in which processes executed through rendezvous points when originally recorded by only permitting the process with matching (smallest) sequence number to enter at any single time. Processes with higher sequence numbers will block and wait for their turn. SCRIBE exploits these rendezvous points to preserve the partial ordering of related system calls during replay. Figure 2.2 illustrates the use of rendezvous points when replaying the `write` and `read` system calls.

Table 2.2 lists all categories of related systems calls, and the respective resources used for rendezvous points. SCRIBE defines a special pseudo rendezvous point that is used for system calls that access properties global to the execution environment, such as `syslog`, `sethostname`, and `settimeofday`. It is also used for system calls that modify system-wide state such as mount points, pseudo terminals, etc. This preserves their order to ensure that the settings are accurate should the system go live at any point.

For system calls that operate on open file objects, including files, devices, network sockets, and pipes, SCRIBE uses inodes as rendezvous points. Inodes are referenced by a variety of file-related system calls such as `read`, `write`, `close`, and `fcntl`. Since most file-related operations are re-executed during replay, this ensures that they occur in the proper order for a given inode.

Similarly, for system calls that operate on System V IPC objects, including message queues, semaphores, and shared memory, SCRIBE uses the respective System V IPC resources as rendezvous points.

Shared memory pages that are file mapped can be accessed either via direct memory references, or

System call category	Rendezvous resource
actions on globals	global (pseudo)
actions on open file objects	`inode` of the file
actions on IPC objects	IPC objects
read/write shared mapped files	memory page
actions on pathnames	filesystem mount point
create file descriptors	file descriptors table
modify memory layout	memory descriptor
actions on process properties	process descriptor

Table 2.2: **List of Rendezvous Points Categories.**

through the virtual filesystem (VFS) using `read` and `write`. By definition, access via system calls will bypass the page protection mechanism that enforces the CREW protocol. For example, through the VFS, a process may change a page that it does not own. To prevent deadlocks and ensure consistency with CREW, SCRIBE associates rendezvous points with these pages. This guarantees that the two methods to access shared mapped pages are properly coordinated, and that their order is preserved between record and replay.

For system calls that operate on filesystem pathnames, including `open`, `unlink`, `creat`, `fifo`, `access`, `stat`, `chmod`, `chown`, `execve`, and `chroot`, SCRIBE must be able to track their ordering to replay them correctly because they may modify the state of the filesystem by creating, deleting and modifying attributes of files. SCRIBE uses filesystem mount points as rendezvous points, but uses them at the VFS layer, not the system call layer. Using them at the system call layer would serialize all filesystem accesses during recording and cause high overhead. Instead, we observe that the order in which system calls that act on pathnames view and modify the filesystem state depends on the order of pathname lookup progress at the VFS layer. The VFS performs pathname traversals one component at a time, always holding the lock of a parent directory while accessing or modifying its contents. To reproduce the order of system calls that act on pathnames, it suffices to record the order of pathname traversal. SCRIBE achieves this by interposing on the VFS pathname traversal to increment the sequence number for the rendezvous point associated with the mount point. Since SCRIBE is only concerned with actions that affect the existence or access permissions of files, it only needs to increment the sequence number for system calls that perform such operations. This

imposes negligible overhead during recording.

In the presence of threads, SCRIBE must also use rendezvous points to track system calls that create file descriptors, modify memory layout, and modify process properties or credentials, since those per process resources are shared among threads. For system calls that create file descriptors, such as `open`, `pipe`, and `fifo`, SCRIBE uses the calling process's file descriptor table as a rendezvous point. SCRIBE cannot rely on an underlying inode for synchronization, because it does not yet exist. For system calls that modify the memory layout, such as `brk`, `mmap`, `munmap`, and `mprotect`, SCRIBE uses the calling process's memory descriptor as a rendezvous point. For system calls that modify process properties and credentials, including `setuid`, `setgid`, `setpgid`, `setsid`, `setrlimit`, SCRIBE uses the process descriptor as the rendezvous point. This is convenient because the properties belong to a process, and the affected operations are performed in that process's context.

SCRIBE also uses the process descriptor rendezvous point to ensure correct ordering among system calls that modify the filesystem view of a process, such as `chroot` and `chdir`. System calls that are re-executed on replay and implicitly rely on process properties and credentials must also use the rendezvous point associated with the process descriptor. For example, `open` and `access` use a process's user and group identifiers to decide if it has sufficient permissions to operate on a file, `kill` uses capabilities to permit a signal, and `setpgid` uses a process's session identifier.

Executing a system call may result in recording multiple rendezvous events. The categories of rendezvous points listed in Table 2.2 are not mutually exclusive. For example, running `open` on a file already opened by another process, will result in a rendezvous event for the global resource, for the process descriptor, and for the inode resource.

2.6 Sync Points

Asynchronous events cause nondeterminism arising from the timing of their occurrence. Replaying asynchronous events is challenging because it requires that a recorded event occur at the exact same place in the process's instruction stream as during recording. It is difficult because it could have occurred at an arbitrary location during the execution. The two predominant examples of asynchronous events are signal delivery and page ownership transfers for shared memory, described in §2.4.

Consider signal delivery. Signals are delivered in two steps. First, the sender process sends a signal

	Record (action) →	**Event log** →	**Replay (action)**	**User-space**
(a)	(do nothing)	(queue *sig* on *B*)	(do nothing)	*(A) kill(B, sig)*
	return(0)	*(A) syscall_ret(0)*	return(0)	
	...	...	...	...
	kill(B, sig)	*(B) signal_received(sig)*	kill(B, sig)	*(B) sync point*
(b)	(A) (stall)	(queue *ADDR* on B)	(A) (stall)	*(A) read page ADDR*
	...	...	...	...
	(B) (adjust PTE)	(B) *page_share_read(ADDR)*	(B) (adjust PTE)	*(B) sync point*
	(A) (adjust PTE)	(A) *page_share_read(ADDR)*	(A) (adjust PTE)	*(A) woken up*
	(A) read page ADDR		(A) read page ADDR	

Figure 2.3: **Asynchronous Events Record-Replay.** (a) The sender of a signal always skips the call and notifies the receiver instead; The receiver handles and logs the signal when it reaches a sync point. (b) Assume process B owns a page for writing. Process A faults reading from the page, notifies the owner, and blocks; When B reaches a sync point, it downgrades the page state (and PTE) to read-only, and logs a memory event; Finally, A updates its own PTE and resumes execution.

to the target process, which is marked as having a signal pending. Second, the target process detects the pending signals when it resumes from kernel space, and handles them. If the target process is executing in user space, an inter-processor interrupt will force it into kernel space, where it will detect the pending signals. Replaying this behavior requires interrupting the target process at the exact same instruction as during its original execution. This is difficult because the interrupt could have occurred at any time during execution.

Consider page ownership transfers for managing shared memory. As described in §2.4, a process requesting access to a shared memory page will page fault if it does not have the necessary ownership to read or write the page. This fault occurs asynchronously with the execution of the process that owns the page. Replaying this behavior requires interrupting the owner process at the exact same instruction at which ownership is transferred to the requesting process as during its original execution. This is difficult because the fault could have occurred at any time during its execution.

Attempting to address this problem while providing application transparent record-replay, previous approaches [22; 23; 41; 42] have relied on hardware providing a cycle accurate instruction counter [100]. The

respective counter value at which the asynchronous event occurs is logged so that during replay, the event can be replayed at the exact same counter value. The fundamental problem with this approach is that such counters are not available on many CPUs, and even when available, often do not have required degree of accuracy because they were not designed for this purpose. They work for performance measurements where occasional missed counts in various corner cases are not problematic, but do not work for record-replay where precise instruction counts are required.

SCRIBE takes a fundamentally different approach to address this problem by introducing a novel and efficient mechanism that makes asynchronous events much easier to record and replay by deferring their delivery until the nearest synchronous system event. This is done by introducing *sync points* to represent synchronous system events which are used for this purpose. Sync points are locations in a recorded process's execution which (1) cause the process to enter kernel space by executing the following instruction, and (2) are guaranteed to do so deterministically during replay (assuming a faithful execution prior to reaching there). Since SCRIBE interposes on these kernel entry points, it can easily record the occurrence and location of sync points. Calling a system call and triggering a trap due to division by zero are two examples of sync points. Certain page faults, namely due to invalid memory access, or due to memory sharing also qualify. However, page faults due to copy-on-write or memory paging do not satisfy the second requirement.

2.6.1 Signal Delivery

During recording, SCRIBE defers the delivery of an asynchronous signal until the target process is at a sync point. This allows SCRIBE to easily determine the exact instruction at which the signal is delivered. If the target process is in user space, SCRIBE queues the signal until the process reaches a sync point, such as a system call, and therefore synchronously enters the kernel. This effectively transforms the asynchronous nature of signals into synchronous behavior. Specifically, when a process enters kernel space, it first checks if it has any pending deferred signals. If so, SCRIBE will deliver them to the process and log a corresponding *signal_receive* event for each delivered signal, then force it to return to user space to handle them. In the case of a sync point due to a system call, it will also rewind the instruction pointer so that the process will re-issue the system call. If the target process is in kernel space, it is already at a sync point and the signal is delivered immediately.

Note that some signals are synchronous in that they are the direct result of an action of the process, like `SIGSEGV`, `SIGFPE`, `SIGBUS`. These occur while the process is in user space, and cannot be deferred for a

later time. They already force the process into kernel space and can therefore be delivered and handled on the spot. These signals do not need to be logged because they are deterministic, and will implicitly occur as part of the replay once the condition occurs that had triggered them in the original recording.

During replay, the sender process skips the system call that sends the signal and continues execution. Instead, signal delivery is deterministically replayed at the occurrence of sync points in the execution of the receiving process. When a process enters kernel space as it reaches a sync point, SCRIBE examines the next event in its log queue; if it finds a *signal_receive* event, it will deliver the designated signal to the process. The process will handle the signal as soon as it resumes to user space. Figure 2.3a illustrates record-replay of signals.

One set of signals, `SIGSTOP` and `SIGCONT`, are treated differently. Unlike other signals, which are replayed by arranging for the process to receive the desired signal, SCRIBE does not resend `SIGSTOP` as it would interfere with the replay. Because replay is performed in the context of the process, a stopped process will never check its queue for the corresponding `SIGCONT` signal. Instead SCRIBE maintains the process in a "stalled" state in kernel space, and examines the following events in the queue. The next event may be either another *signal_receive* event or a page ownership transition event, as discussed in §2.4. SCRIBE processes the remaining events in the queue until it encounters a `SIGCONT`, and then allows the process to resume execution. When a session that contains a stalled process prepares to go live, SCRIBE arranges to send the previously skipped `SIGSTOP` to the process, forcing the process into the proper kernel state.

2.6.2 Page Ownership Transfer

Page state transitions are allowed to only take place when SCRIBE can conveniently track, and later replay them. We draw the following analogy to signal delivery: the process that page faults and the page owner(s) are analogous to the sender and the receiver of a signal, respectively. SCRIBE converts asynchronous memory events into synchronous ones by deferring them until the owner process reaches a sync point.

When a process tries to access an owned page it notifies the owner and, unlike with signals, blocks until access is granted. Conversely, owner processes check for pending requests at every sync point and, if necessary, give up ownership. Figure 2.3b illustrates record-replay of memory interleaving. Note that page faults due to the memory interleaving under the CREW protocol contribute significantly to the pool of sync points, adding to system calls.

Although transfer of page ownership is always performed by the owner process(es), there is one excep-

tion to this rule due to interaction of blocking system calls and shared memory. When an owner of a page blocks inside a system call, it cannot transfer its page ownership to another process. This can cause long delays in ownership transfer, and even lead to a deadlock if, for example, the owner blocks on a read from a pipe, and the other process stalls on a memory access while attempting to write into the same pipe.

To address this problem, SCRIBE guarantees that user space shared memory is not accessed by an owner process when it is executing a system call. If another process needs to access a shared memory page owned by the calling process, SCRIBE can simply transfer ownership to the requesting process knowing that the original owner process will not access shared memory because it is executing a system call. There are no shared memory interleavings to track between the original owner process and the requesting process. SCRIBE can just identify the location in the original owner's instruction stream at which this ownership transfer occurs as being the occurrence of the system call, which it already logs.

More specifically, since various system calls transfer data between the kernel and user space which could involve a shared page owned by the calling process, SCRIBE uses an in-kernel staging area where it temporarily stores both input and output data. As a result, only the staging area, not user space memory, is accessed by the calling process during system calls. SCRIBE flags an owner that enters a system call as a *weak-owner* until the system call completes. This flag indicates that other processes may promptly revoke ownership of pages that it holds whose retention interval expired. If during the system call the owner also becomes blocked, SCRIBE flags it as a *sleep-owner* until it resumes execution. This flag indicates that other processes may promptly revoke ownership of any pages that it holds. PTEs of the requesting processes and the owner are updated promptly to reflect these actions.

2.6.3 Signature Record and Replay

Deferring signals and page ownership transfers may incur a performance penalty by increasing the latency of signal delivery and page faults on shared memory, respectively. SCRIBE's approach to recording and replaying asynchronous events is predicated on the assumption that sync points occur frequently enough in real applications, since they often enter kernel mode by executing system calls or causing page faults. Based on this assumption, we expect any performance overhead to be low in practice. §2.7 presents experimental results that validate our assumption.

In addition, for tracking shared memory, it assumes that real applications do not typically use user space-only spinlocks or related mechanisms. For example, consider a thread that reads from a memory location

in a busy loop until it finds a positive value, and another thread that intends to write a positive value to that location. Assume that the former thread becomes the owner of the page. The threads are now deadlocked, since the second thread waits for the first thread to give up the ownership for the page, and the first thread waits for the second one to change the value in the memory. This assumption may be incorrect for certain applications.

Although the likelihood of either scenario is not common in real applications, SCRIBE also provides a novel but more heavyweight mechanism to record and replay asynchronous events that were deferred for too long due to an unlikely absence of sync points. During recording, if a signal, or a page ownership transfer, has been deferred for a period that exceeds a predefined threshold, SCRIBE switches to a different mechanism. SCRIBE sends the target process a reserved signal that forces it into kernel mode. By using a reserved signal, we ensure that process execution does not depend on it in any way. It then creates a signature of the process: a checkpoint of the current user space context of just that process, namely its registers and writable memory pages.

A key observation here is that between sync points, the process is guaranteed to not have any interactions with the operating system, or any nondeterministic interactions with other processes, since its last sync point and until it is finally forced into kernel mode. Therefore, SCRIBE is also guaranteed not to have missed recording any nondeterministic interactions by forcing the process into kernel mode. Thus, forcing the process into kernel mode can be thought of as resetting the recording, and is logged as a *async_reset* event. By forcing the process into kernel mode, we effectively create a new sync point. The original pending signal or page ownership transfer can then be handled and its location with respect to the new sync point is precisely known.

As an optimization, SCRIBE can leverage special hardware features when available instead of performing heavyweight process signatures. On some Intel platforms, certain hardware performance counters can be used to count instructions deterministically [119], meaning that the counter values remain identical across replays. For example, on the Westmere, SandyBridge, and IvyBridge architectures, the retired conditional branch counter is not affected by hardware interrupts or nondeterministic page faults that could make an instruction be counted twice. By using similar techniques used in Mozilla rr [1], SCRIBE can accurately count the number of instructions executed after a sync point. This way, SCRIBE can precisely record when an asynchronous event is delivered in userspace.

During replay, the key issue is knowing when the process should consume the *async_reset* event. Other

Name	**Description**
apache-p	Apache 2.0.54, 8 processes, `prefork`
apache-t	Apache 2.0.54, 50 threads, `worker`
mysql	MySQL 5.0.60 database server
ssh-s	OpenSSH 5.1p1 (server)
ssh-c	OpenSSH 5.1p1 (client)
make	parallel compilation of Linux kernel
untar	untar of Linux 2.6.11.12 source tree
urandom	reading from `/dev/urandom`
editor	vim 7.1 text editor
firefox	Firefox 3.0.6 web browser in VNC
acroread	Adobe Acrobat Reader 8.1.3 in VNC
mplayer	Mplayer 1.0rc2 movie player in VNC
openoffice	OpenOffice 3.0.1 office suite in VNC

Table 2.3: **Application Scenarios.**

approaches suggested the use of hardware performance counters despite their shortcomings [17; 103]. Since SCRIBE is designed for commodity operating systems without base kernel changes, it does not have access to scheduling decisions and data that are essential for using performance counting; without it, it is impossible to accurately correlate performance counter data to individual processes that execute in user space.

SCRIBE takes a different approach. Starting at the last event in the log prior to the *async_reset* event, it will set a breakpoint at the instruction specified by the saved value of the program counter. The process will generate an exception each time that it reaches the instruction pointed to by the saved program counter, prompting SCRIBE to compare its current user space context, namely, registers and contents of writable memory pages with that of the *async_reset* event. The *async_reset* event occurs when the data at the replayed process matches that of the event. Once that happens, SCRIBE can remove the breakpoint and continue normal replay. Although this signature-based record and replay can be expensive, the overhead can be minimized by only recording differences in signatures. More importantly, forcing a sync point is rarely needed in practice for handling asynchronous events.

Name	Benchmark	Time
apache-p	`httperf 0.8` (rate=1500, num-calls=20)	189 s
apache-t	`httperf 0.8` (rate=1500, num-calls=20)	187 s
mysql	`sql-bench`	184 s
ssh-s	50 SSH sessions (10 concurrent), each emulates user typing 5K text file	53 s
ssh-c	50 SSH sessions (10 concurrent), each emulates user typing 5K text file	53 s
make	`make -j10` of the Linux kernel	101 s
untar	`gunzip linux-2.6.11.12.tar.gz | tar xf -`	2.8 s
urandom	`dd=/dev/random bs=1k count=10000 | lzma > /dev/null`	2.6 s
editor	`vim -S vi.script` to append 'hello world' 1000000 times	12.4 s
firefox	`SunSpider 0.9` JavaScript benchmark	120 s
acroread	open 190 KB PDF, close and exit	2.8 s
mplayer	play 10 MB 1280x720 HDTV video at 24 frames/s	30.8 s
openoffice	`Jungletest r27 (2009-03-08)` open document, export, close, and exit	4.9 s

Table 2.4: **Application Workloads.**

2.7 Performance Evaluation

We have implemented a SCRIBE prototype as a Linux kernel module and associated user-level tools. To demonstrate the effectiveness of our approach, we evaluated the ability and performance of our unoptimized prototype to record-replay real applications on commodity multiprocessors and operating systems.

We ran our experiments on an IBM HS20 eServer BladeCenter, each blade with dual 3.06 GHz Intel Xeon CPUs with hyperthreading, 2.5 GB RAM, a 40 GB local disk, interconnected with a Gigabit Ethernet switch. Each blade was running the Debian 3.1 distribution and the Linux 2.6.11.12 kernel and appears as a 4-CPU multiprocessor to the operating system. For application workloads that required clients and a server, we ran the clients on one blade and the server on another.

We recorded and replayed a wide range of real applications, listed in Table 2.3. The list includes (1) server applications such as Apache in both multi-process (`apache-p`) and multi-threaded (`apache-t`) configurations, MySQL (`mysql`), and an OpenSSH server (`ssh-s`), (2) utility programs such as SSH clients (`ssh-c`), make (`make`), untar (`untar`), compression programs such as gzip and lzma, and a vi editor (`editor`), and (3) graphical desktop applications such as Firefox (`firefox`), Acrobat Reader (`acroread`), MPlayer (`mplayer`), and OpenOffice (`openoffice`). To run the graphical applications

on the blade which lacks a monitor, we used VNC (TightVNC Server 1.3.9) to provide a virtual desktop.

We measured the performance of SCRIBE using the benchmark workloads listed in Table 2.4. Applications were all run with their default configurations. Workloads were selected to stress the system to provide a conservative measure of performance. For example, `firefox` runs the widely used SunSpider benchmark designed to measure real-world Web browser JavaScript performance. We also included benchmarks that emulate multiple interactive users such as `ssh-s` and `ssh-c`, which open multiple concurrent SSH sessions, each having an emulated user input text into a vi editor at world-record typing speed [110] to create a 5 KB file, then exiting. We focus on quantifying the performance overhead and storage requirements of running applications with SCRIBE in terms of the cost of continuously recording the execution, and speedup of replayed execution versus recorded execution. Previous work shows that the overhead of the virtual execution environment is small [61; 81].

Figure 2.4 shows the performance overhead of recording the application workloads. Performance is measured as completion time in all cases except for `apache-p` and `apache-t` which report performance in completed requests per second. No frames were dropped during logging of `mplayer` playback. Results are shown normalized to native execution without recording. Recording overhead was under 2.5% for server applications and under 7% for all desktop applications except for `openoffice`, which was 15%. For all desktop applications, there was no user noticeable degradation in interactive performance.

Figure 2.4 also shows the performance of replaying the applications workloads. Performance is measured as completion time, normalized to execution with recording. Replaying speedup relative to recording was at least 1 in all cases, and reached as much as a factor of 70 for `ssh-c`. The results demonstrate that SCRIBE can replay applications at least as fast as it records, as expected. This is useful for fault-tolerant systems to guarantee that replay on the backup does not slow down execution on the primary.

Two factors contribute to replay speedup: omitted in-kernel work due to system calls partially or entirely skipped (e.g. network output), and compressed time due to time waiting skipped at replay (e.g. timer expiration). Application that do neither perform the same work whether recording or replaying, and sustain speedups close to 1. This includes computation-intensive workloads such as `make`, `urandom`, `untar`, and `editor`. The speedup increases as the workload exhibits more idle time in sleeping or blocking (mostly waiting for input events). For instance, `mplayer` spends about 23% of the time sleeping during recording, and its replay speedup is roughly 1.3. Replay speedup is noticeably larger for workloads that spend much of their time sleeping: 3.9 for `acrobat`, 7.1 for `apache-p`, and 5.8 for `apache-t`. Interactive workloads

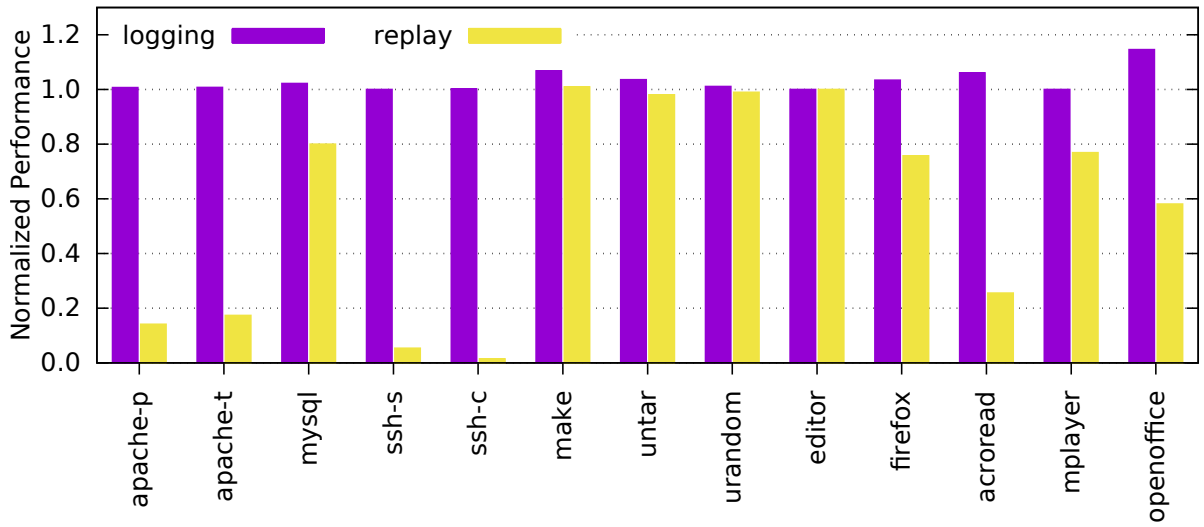


Figure 2.4: **Recording Runtime Overhead.**

obtained the largest speedups: 19 for `ssh-s` and 70 for `ssh-c`.

Figure 2.5 shows the storage growth rate of recording. Storage requirements are decomposed into memory-related events (`memory`), nondeterministic input data returned by system calls (`input data`), and other data which is primarily system call return values and rendezvous points (`syscalls`). The storage growth rates ranged from 100 KB/s for `ssh-c` to almost 1.9 MB/s for `mysql`. These storage requirements are quite modest. When compressed using lzma, storage growth rates dropped to between 1 to 90 KB/s for all scenarios except `urandom`, whose storage growth rate remained a bit over 1.1 MB/s. Most of the log of `urandom` is due to input of random data, which does not compress well.

Figure 2.6 shows the average number of processes and threads running for each application scenario. The sum of the two is the average number of total Linux tasks running. All workloads except `editor` consisted of multiple processes or threads, demonstrating SCRIBE's ability to record and replay real multi-process and multi-threaded application workloads. Five of the scenarios used threads: `apache-t`, `mysql`, `ssh-c`, `firefox`, and `openoffice`. For all of these scenarios except `ssh-c`, this correlates with the majority of the log storage consisting of memory events, as shown in Figure 2.5. The threads in `ssh-c` are used in the benchmark to manage concurrent sessions. They involve very little contention over shared memory, and therefore do not contribute much to the log size. Conversely, `apache-p` shows mild shared memory activity despite being a multi-process application rather than multi-threaded.

Figure 2.7 shows the time interval between consecutive per process sync points for each application

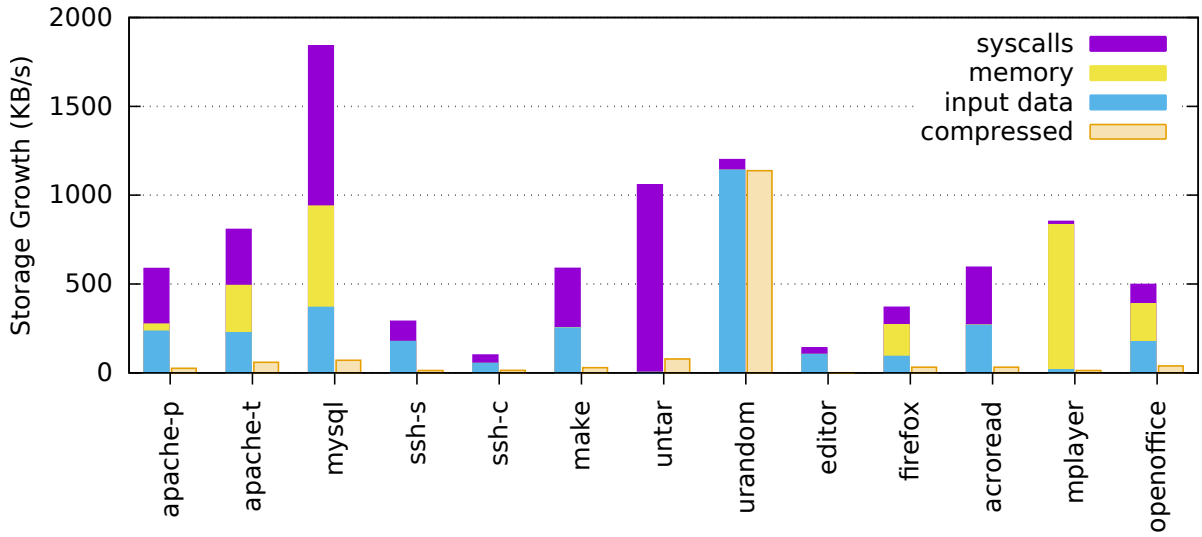


Figure 2.5: **Recording Storage Growth.**

scenario. The average time interval is measured per process then averaged over all processes. It is at most 30 μs for all scenarios except `make`, `urandom` and `editor`, for which it is less than 500 μs. These three are CPU intensive workloads that produce sync points only due to system calls. The maximum time interval between sync points for almost all application workloads was less than 100 ms, which is also not large and similar to the scheduling time quantum in Linux. The maximum time interval for three application workloads, `make`, `firefox`, and `openoffice`, was higher, but only occurred once, during the startup of each application. If we exclude these outliers and compute the 99th percentile of the time interval between sync points, the time interval is less than 10 ms.

Figure 2.7 also shows the average length of sync points per process for each application scenario. It is at least 300 μs for all scenarios except `untar` and `mplayer`, in which it is over 50 μs. More importantly, in all workloads the average time spent at a sync point is significantly larger—over an order of magnitude in most cases—than the time spent between sync point, or outside sync points. Processes persist longer at sync points whenever, for example, they block on I/O in a system call or wait for page ownership transfer. During the time intervals within sync points, asynchronous events for a process are delivered instantly and need not be deferred. In other words, on average, most of the time asynchronous events can be delivered promptly; and if not, then they are delayed for a short period. This establishes the empirical grounds for SCRIBE's reliance on sync points to successfully convert asynchronous events to synchronous ones in a timely manner.

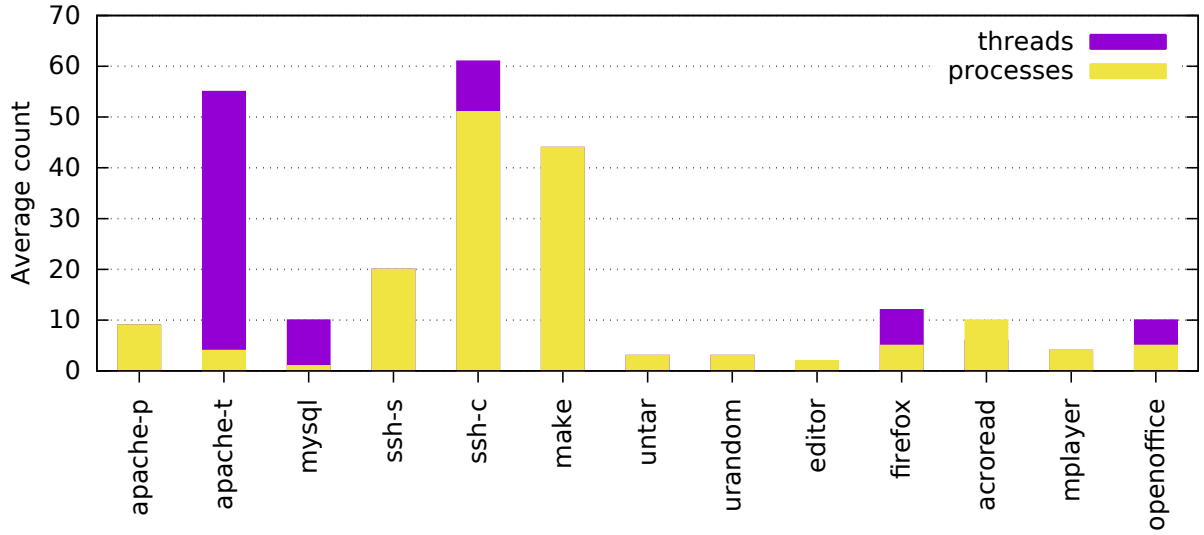


Figure 2.6: **Number of Processes and Threads.**

Figure 2.8 shows the total number of signals and shared memory page faults due to SCRIBE's page ownership management mechanism for each application scenario. Page faults not due to SCRIBE are not included. The totals are decomposed into those that are handled instantly versus those that need to be deferred until a sync point is reached. The measurements show that SCRIBE provides low-overhead execution recording even in the presence of a large number of asynchronous events. Nearly all asynchronous events of either type are handled instantly as they arrive, because the process that is the target of these events is already executing in the kernel at a sync point. Sync points not only happen frequently enough, but also endure long enough, that the vast majority of asynchronous events can be handled immediately without being deferred.

Observe in Figure 2.8 that asynchronous events due to shared memory page faults predominate over signals in scenarios that involve multiple threads or shared memory. In these scenarios, page faults due to SCRIBE's page ownership management occur in larger numbers, and, since they themselves are sync points, they contribute to the pool of available sync points. The fraction of sync points due to shared memory page faults of the total number of sync points ranges from 10% in `apache-p`, to 30% in `mysql`, `apache-t`, `firefox`, and `openoffice`, and up to 50% in `mplayer`. In other words, applications that need sync points for shared memory accesses are also likely to have sync points more frequently.

Figure 2.9 shows the amount of delay incurred for signals and shared memory accesses. Signals were delayed at most 100 μs on average, except `ssh-c`, which reached 220 μs. The average delay for only those few deferred signals that could not be handled instantly was at most 1 ms. CPU intensive workloads without

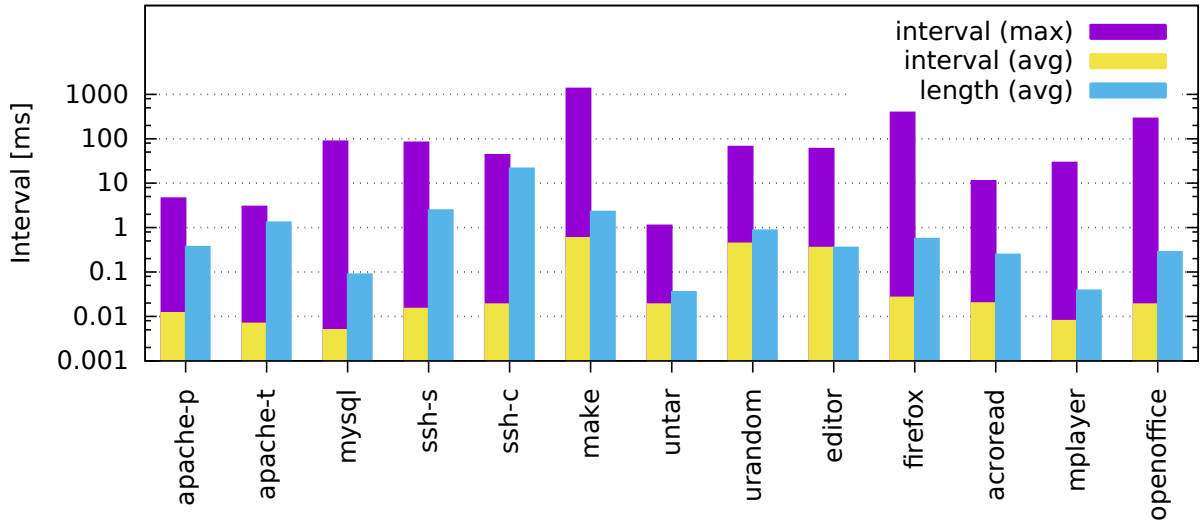


Figure 2.7: **Sync Points Interval and Length.**

shared memory produce sync points only due to system calls, and sustain longer delays for deferred signals. For example, in `make`, 135 `SIGCHLD` signals were deferred as the parent process waited to be scheduled while compilations occupied the CPUs. Only when it was scheduled, it reached a sync point and handled the signal. However, even without SCRIBE, when the signal is delivered instantly, the parent process would only handle the signal after a comparable delay since it would still wait to be scheduled. In multi-threaded workloads, the delays for signals are longer, despite the addition of sync points due to shared memory accesses. This is because our prototype only considered sync points due to system calls for deferred signals. The delays would probably be more comparable to those for shared memory access if sync points due to shared memory were also used.

Unlike with signals, when a shared memory event occurs, the process that faulted blocks until access is granted. Thus, whether memory events are delayed and for how long is pivotal for the performance of the system. Fortunately, shared memory accesses introduce numerous additional sync points due to page ownership transfers. The average delay for shared memory accesses was less than 25 μs. If we consider only deferred shared memory accesses that could not be handled instantly, the average delay increases modestly to at most 60 μs. These delays are comparable to the native service time of a page fault. SCRIBE's sync points convert page ownership transfers from asynchronous events to synchronous events with negligible impact on page fault performance, since most asynchronous events are handled instantaneously.

Finally, through all the executions of the application scenarios, we have never observed a situation

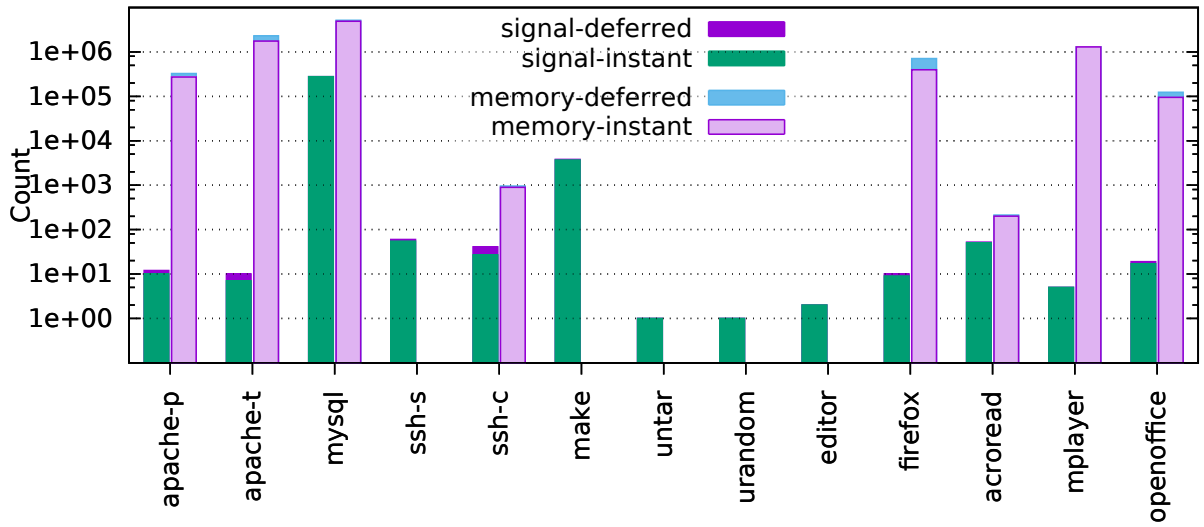


Figure 2.8: **Count of Signals and Memory.**

in which a process failed to reach a sync point in a reasonable time, or at all. Although SCRIBE has a mechanism in place to deal with delays that become too large, we did not witness a need for this functionality in practice. Our experiences and results demonstrate that sync points occur frequently and are useful for enabling deterministic replay.

2.8 Related Work

Replaying program execution has been of interest for over 40 years [14]. Hardware mechanisms [10; 39; 54; 71; 72; 75; 76; 124] face a high implementation barrier and do not support record-replay on commodity hardware. Virtual machine mechanisms [23; 41; 42; 117] require replaying operating system execution just to replay application execution. Almost none of them support replaying multiprocessor virtual machines, and the ones that do incur an order of magnitude worse overhead for common applications like compilation due to kernel-level sharing, such as writing files to the same directory [42]. Most of application and library mechanisms [50; 52; 79; 96] cannot provide transparent record-replay for unmodified applications. Only one practical userspace solution exists, Mozilla rr [1]. While it provides a comprehensive set of features, it does not support multiprocessor record-replay. Programming language mechanisms [29; 64; 95] do not support widely-used applications written in languages that do not provide record-replay primitives. Unlike these approaches, SCRIBE is an operating system mechanism. It works at a higher-level abstraction than hardware or virtual machine approaches to reduce recording overhead. It works at a lower-level abstraction

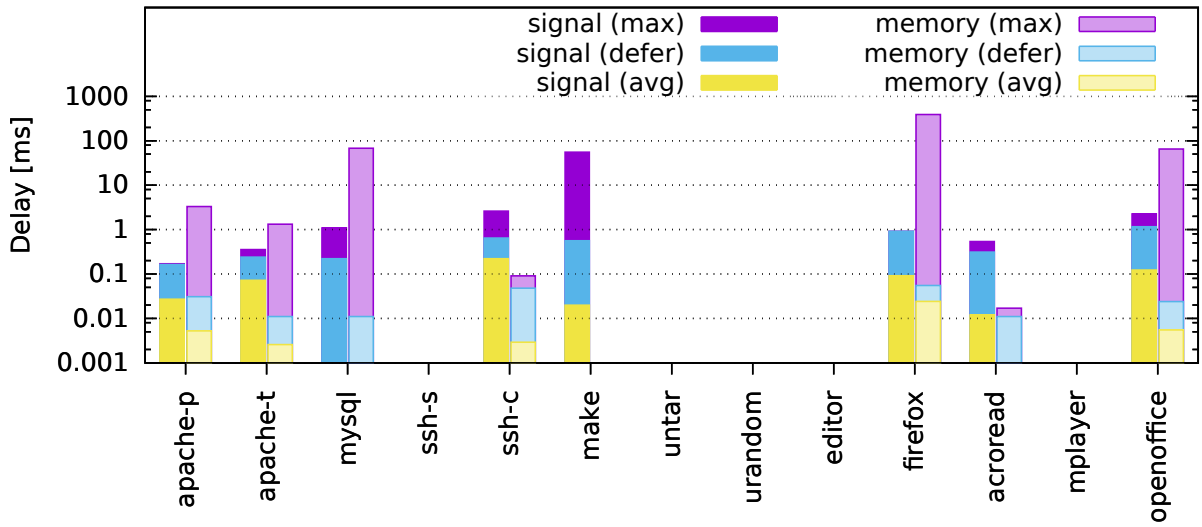


Figure 2.9: **Delay of Signals and Memory.**

than application, library, and programming language approaches to provide transparent record-replay for unmodified applications.

Other operating system mechanisms have also been proposed [17; 22; 102; 109] that interpose between applications and the operating system. None of them provides record-replay for multi-threaded and multi-process applications. In fact, only TFT [22] shows any record-replay results for real applications, namely `gzip`, a single process application, but overhead was quite high. Unlike SCRIBE, TFT is only designed to replay a single process. Debugging using deterministic replay (DUDR) [109] presents only a paper design with no implementation or evaluation, while Flashback [102] and RR [17] are largely incomplete with no results beyond those for a single, simple test program. In contrast, SCRIBE demonstrates for the first time that record-replay of real multi-threaded and multi-process applications is possible using an operating system approach.

A key issue for operating system mechanisms is replaying the in-kernel side effects of system calls. This must be done for at least some system calls in all replay systems. Previous approaches do not solve the important problem of nondeterminism arising from the order of execution of related system calls. TFT only replays a single process, so this issue does not arise. DUDR and Flashback hypothesize counting instructions to know when context switches occur to track exact scheduling order to know the order of system call execution among processes. However, they provide no mechanism for obtaining and using the required cycle accurate counters, and the approach itself does not work for multiprocessors. RR suggests instrument-

ing the system call interface, but provides no actual mechanism to do it. In contrast, SCRIBE provides a new mechanism using rendezvous points that solves this problem without tracking exact scheduling order. SCRIBE's mechanism does not require hardware support and works for multiprocessor systems.

Record-replay systems must record the exact location in an instruction stream at which an asynchronous event occurs. This can be done by adding hardware support, modifying applications, or writing applications with new language primitives to record exactly when the application receives the event. To do this on commodity hardware without application changes, all previous approaches that deal with this issue [23; 41; 42; 100] rely on the existence of a cycle accurate instruction counter. To deal with interrupt lag [103], replay is done by interrupting execution some time before the asynchronous event should occur, setting a breakpoint on the instruction at which it should occur, then stopping at every breakpoint to see if the instruction counter matches the recorded value. When they match, the asynchronous event is delivered. In contrast, SCRIBE introduces a fundamentally different mechanism based on sync points that does not rely on hardware performance counters.

TFT [22] proposed recording in periodic epochs for fault tolerance, and then deferring the delivery of signals sent in each epoch until the respective epoch ends. Epochs are created by instrumenting applications to use counters to periodically return control to TFT. This also makes it easier to determine when signals are delivered since they are delivered at well-defined epoch boundaries. The idea is similar to SCRIBE's notion of deferring signal delivery until sync points. But, unlike TFT, SCRIBE does not require instrumenting applications and does not define sync points based on any measure of time or instruction counts. Instead, sync points are based on system calls, page faults, and traps that occur as part of normal application execution. Unlike TFT which only supports replaying a single process, SCRIBE uses sync points to enable replay of multi-process and multi-threaded applications on multiprocessors.

Besides SCRIBE, only SMP-ReVirt [42] can transparently replay multiprocessor workloads that use shared memory. SMP-ReVirt replays multiprocessor virtual machines where multiple CPUs may access shared memory. It uses standard page protection to detect memory races, and the concurrent read, exclusive write (CREW) protocol [34; 64]. To record exactly when page access permissions switch from one CPU to another, SMP-ReVirt records counter values in the same manner as it does for handling other asynchronous events. RR [17] proposes a mechanism similar to SMP-ReVirt, but notes problems with inaccuracy of hardware counters on modern CPUs and has no record-replay results for any applications. In contrast, SCRIBE avoids counter inaccuracies and introduces sync points based on the assumption that real applications per-

form frequent system activities that involve the kernel. This assumption is the antithesis of SMP-ReVirt's virtual machine approach which must also record kernel execution. For example, SMP-ReVirt incurs an order of magnitude worse overhead than SCRIBE for kernel compilation due to frequent system activities that result in kernel-level sharing. While SMP-ReVirt can provide whole system replay, SCRIBE can provide much more efficient application replay.

2.9 Summary

SCRIBE is the first operating system mechanism to provide transparent, deterministic execution record and replay of multi-threaded and multi-process applications on commodity multiprocessors and operating systems. SCRIBE records and replays multiple processes by accounting for nondeterministic interactions among processes and their execution environment. SCRIBE introduces *rendezvous points* to ensure correct partial ordering of execution based on system call dependencies, and *sync points* to convert asynchronous interactions that can occur at arbitrary times into synchronous events that are much easier to record and replay. Using these two mechanisms, SCRIBE overcomes the need of enforcing the original application scheduling. While a replayed execution is different from the original one from a kernel perspective, they appear identical from an application perspective. By relaxing the amount of determinism replicated at the kernel level, SCRIBE recording overhead stays modest for various server and desktop applications without sacrificing faithfulness at the application level. SCRIBE can transition an application to running live at any time, an instrumental for certain record-replay applications such as fault-tolerance, or any form of debugging that requires the replayed instance to go live. We built RACEPRO, a process race detection system based on the SCRIBE engine, that leverage this go-live feature when performing race triage. We describe RACEPRO in the next chapter and show how it generates new executions based on previously recorded executions.

Chapter 3

RACEPRO: Detection of Process Races in Deployed Systems

3.1 Introduction

After presenting SCRIBE, our transparent record-replay engine most suited for application debugging by capturing and reproducing hard to find bugs, we explore ways to reveal dormant bugs in applications, and catch them before they happen. Specifically, we explore the effect of bugs due to harmful process races in software, and how to detect them using record-replay mechanisms.

While thread races have drawn much attention from the research community [35; 43; 97; 123; 128], little has been done for *process races*, where multiple processes access an operating system (OS) resource such as a file or device without proper synchronization. Process races are much broader than time-of-check-to-time-of-use (TOCTOU) races or signal races [129]. A typical TOCTOU race is an atomicity violation where the permission check and the use of a resource are not atomic, so that a malicious process may slip in. A signal race is often triggered when an attacker delivers two signals consecutively to a process to interrupt and reenter a non-reentrant signal handler. In contrast, a process race may be any form of race. Some real examples include a shutdown script that unmounts a file system before another process writes its data, `ps | grep X` shows N or $N+1$ lines depending on the timing of the two commands, and `make -j` failures.

To better understand process races, we present the first study of real process races. We study hundreds of real applications across six Linux distributions and show that process races are numerous and a real threat to reliability and security. For example, a simple search on Ubuntu's software management site [63] returns

hundreds of process races. Compared to thread races that typically corrupt volatile application memory, process races are arguably more dangerous because they often corrupt persistent and system resources. Our study also reveals that some of their characteristics hint towards potential detection methods.

We then present RACEPRO, the first system for automatically detecting process races beyond TOCTOU and signal races. RACEPRO faces three key challenges. The first is scope: process races are extremely heterogeneous. They may involve many different programs. These programs may be written in different programming languages, run within different processes or threads, and access diverse resources. Existing detectors for thread or TOCTOU races are unlikely to work well with this heterogeneity.

The second challenge is coverage: although process races are numerous, each particular process race tends to be highly elusive. They are timing-dependent, and tend to surface only in rare executions. Arguably worse than thread races, they may occur only under specific software, hardware, and user configurations at specific sites. It is hopeless to rely on a few software vendors and beta testing sites to create all possible configurations and executions for checking.

The third challenge is algorithmic: what race detection algorithm can be used for detecting process races? Existing algorithms assume well-defined load and store instructions and thread synchronization primitives. However, the effects of system calls are often under-specified and process synchronization primitives are very different from those used in shared memory. For instance, what shared objects does `execve` access? In addition to reading the inode of the executed binary, an obvious yet incomplete answer, `execve` also conceptually writes to `/proc`, which is the root cause of the `ps | grep X` race (§3.5). Similarly, a thread-join returns only when the thread being waited for exits, but `wait` may return when any child process exits or any signal arrives. Besides fork-wait, processes can also synchronize using pipes, signals, `ptrace`, etc. Missing the (nuanced) semantics of these system calls can lead to false positives where races that do not exist are mistakenly identified and, even worse, false negatives where harmful races are not detected.

RACEPRO addresses these challenges with four ideas. First, it checks deployed systems *in vivo*. While a deployed system is running, RACEPRO records the execution without doing any checking. RACEPRO then systematically checks this recorded execution for races *offline*, when the deployed system is idle or by replicating the execution to a dedicated checking machine. By checking deployed systems, RACEPRO mitigates the coverage challenge because all user machines together can create a much larger and more diverse set of configurations and executions for checking. Alternatively, if a configuration or execution never occurs, it is probably not worth checking. By decoupling recording and checking [30], RACEPRO

reduces its performance overhead on the deployed systems.

Second, RACEPRO records a deployed system as a system-wide, deterministic execution of multiple processes and threads. RACEPRO uses lightweight OS mechanisms developed in SCRIBE (§2) to transparently and efficiently record nondeterministic interactions such as related system calls, signals, and shared memory accesses. No source code or modifications of the checked applications are required, mitigating the scope challenge. Moreover, since processes access shared OS resources through system calls, this information is recorded at the OS-level so that RACEPRO can use it to detect races regardless of higher level program semantics.

Third, to detect process races in a recorded execution, RACEPRO models each system call by what we call *load and store micro-operations* to shared kernel objects. Because these two operations are well-understood by existing race detection algorithms, RACEPRO can leverage these algorithms, mitigating the algorithmic challenge. To reduce manual annotation overhead, RACEPRO automatically infers the micro-operations a system call does by tracking how it accesses shared kernel objects, such as inodes. Given these micro-operations, RACEPRO detects *load-store races* when two concurrent system calls access a common kernel object and at least one system call stores to the object. In addition, it detects *wait-wakeup races* such as when two child processes terminate simultaneously so that either may wake up a waiting parent. To our knowledge, no previous algorithm directly handles wait-wakeup races.

Fourth, to reduce false positives and negatives, RACEPRO uses replay and go-live to validate detected races, a core feature of SCRIBE. A race detected based on the micro-operations may be either *benign* or *harmful*, depending on whether it leads to a *failure*, such as a segmentation fault or a program abort. RACEPRO considers a change in the order of the system calls involved in a race to be an *execution branch*. To check whether this branch leads to a failure, RACEPRO replays the recorded execution until the *reordered* system calls then resumes live execution. It then runs a set of built-in or user-provided checkers on the live execution to detect failures, and emits a bug report only when a real failure is detected. By checking many execution branches, RACEPRO reduces false negatives. By reporting only harmful races, it reduces false positives.

RACEPRO heavily relies on record-replay techniques to perform its tasks. However, deterministic record-replay is not sufficient to perform validation of detected races. RACEPRO goes beyond replaying an execution verbatim, as it relies on the ability to modify the recorded execution and evaluate the effect of the modification. Even though reordering system calls can be seen as a small modification, the effect can be

Distribution	Pages		Bugs		
	Returned	Sampled	Total	Process	Thread
Ubuntu	3330	300	45	42 (1)	3
Fedora/RedHat	1070	100	52	30 (10)	22
Gentoo	2360	60	31	23 (10)	8
Debian	768	40	17	12 (4)	5
CentOS	1500	40	5	2 (0)	3
Total	9028	540	150	109 (25)	41

Table 3.1: **Summary of Collected Pages and Bugs.**

significant. As soon as the modifications are executed by the application, RACEPRO allow the application to go-live. This alleviate the need of having a replayer tolerant to divergence from the original execution.

We have implemented RACEPRO in Linux as a set of kernel components for record, replay, and go-live, and a user-space exploration engine for systematically checking execution branches. Our experimental results show that RACEPRO can be used in production environments with only modest recording overhead, less than 2.5% for server and 15% for desktop applications. Furthermore, we show that RACEPRO can detect 10 real bugs due to process races in widespread Linux distributions.

This chapter is organized as follows. §3.2 presents a study of process races and several process race examples. §3.3 presents an overview of the RACEPRO architecture. §3.4 describes the execution recording mechanism. §3.5 describes the system call modeling using micro-operations and the race detection algorithm. §3.6 describes how replay and go-live are used to determine harmful races. §3.7 presents experimental results. §3.9 discusses related work. Finally, §3.10 presents a summary and concluding remarks of this chapter.

3.2 Process Race Study

We conducted a study of real process races with two key questions in mind. First, are process races a real problem? Second, what are their characteristics that may hint towards how to detect them? We collected bugs from six widespread Linux distributions, namely Ubuntu, RedHat, Fedora, Gentoo, Debian, and CentOS. For each distribution, we launched a search query of "race" on the distribution's software management

website. We manually examined a random sample of the returned pages, identified all unique bugs in the sampled pages, and classified these bugs based on whether they resulted in process or thread races. Raw data of the studied bugs is available [90]. §3.2.1 presents our findings. §3.2.2 describes four process race examples from the most serious to the least.

3.2.1 Findings

Table 3.1 summarizes the collected pages and bugs; Fedora and Redhat results are combined as they share the same management website. For each distribution, we show the number of pages returned for our query (Returned), the number of pages sampled and manually examined (Sampled), the number of process races (Process) and the subset of which were TOCTOU races, the number of thread races (Thread), and the total number of bugs in the sampled pages (Total).

Process races are numerous. Of the 150 sampled bugs, 109 resulted in process races, a dominating majority; the other 41 bugs resulted in thread races. However, thread races are likely underrepresented because the websites we searched are heavily used by Linux distribution maintainers, not developers of individual applications. Of the 109 process races, 84 are not TOCTOU races and therefore cannot be detected by existing TOCTOU detectors. Based on this sample, the 7,498 pages that our simple search returned may extrapolate to over 1,500 process races. Note that our counting is very conservative: the sampled pages contain an additional 58 likely process races, but the pages did not contain enough information for us to understand the cause, so we did not include them in Table 3.1.

Process races are dangerous. Compared to thread races that typically corrupt volatile application memory, process races are arguably more dangerous because they often corrupt persistent and system resources. Indeed, the sampled process races caused security breaches, files and databases to become corrupted, programs to read garbage, and processes to get stuck in infinite loops. The top right graph in Figure 3.1 summarizes the effects of all process races from Table 3.1.

Process races are heterogeneous. The sampled process races spread across over 200 programs, ranging from server applications such as MySQL, to desktop applications such as OpenOffice, to shell scripts in Upstart [115], an event-driven replacement of System V `init` scripts. Figure 3.1 breaks down the process races by packages, processes, and programming languages involved. Over half of the 109 process races, including all examples described in §3.2.2, require interactions of at least two programs. These programs are written in different programming languages such as C, Java, PHP, and shell scripts, run in multiple

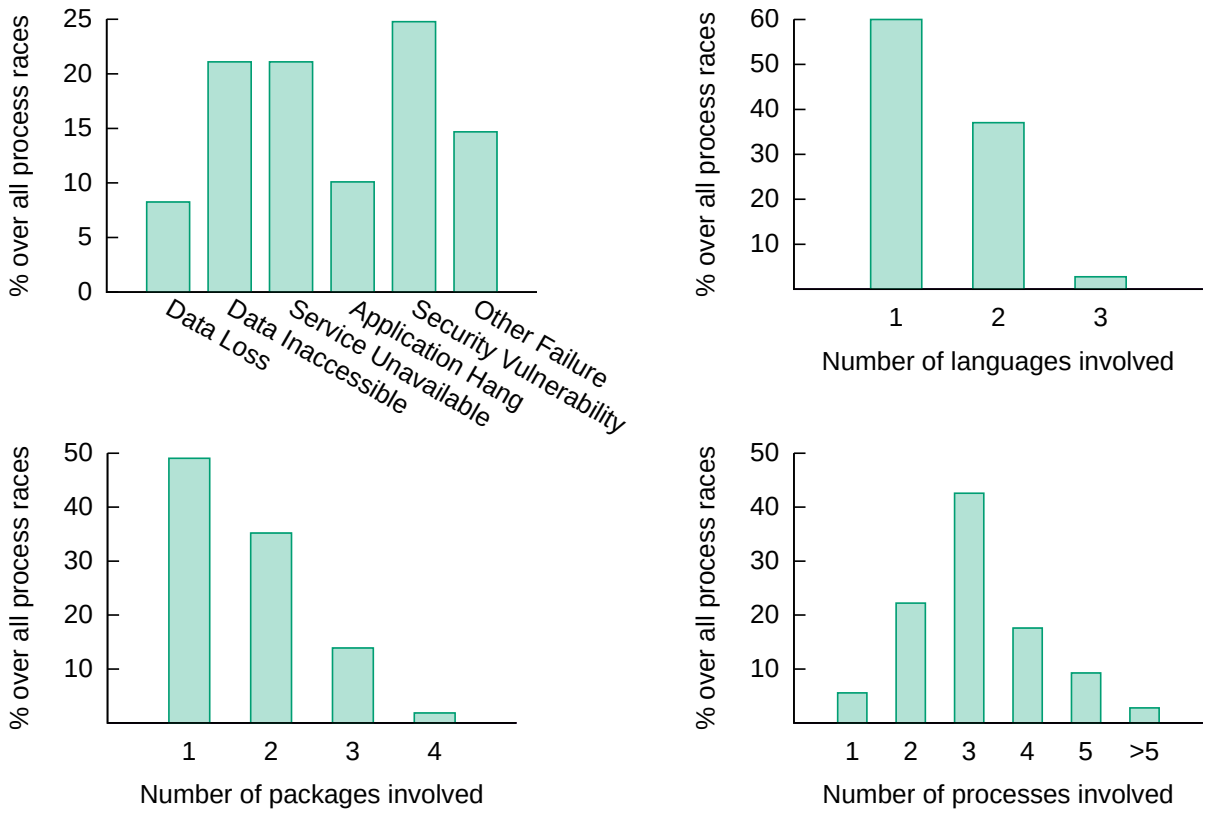


Figure 3.1: **Process Races Breakdown.** X axis shows the race effect, programming languages, the number of software packages, or processes involved. Y axis shows the percentage of process races that involve the specific effect, languages, packages, or processes. To avoid inflating the number of processes, we count a run of a shell script as one process. (Each external command in a script causes a `fork`.)

processes, synchronize via `fork` and `wait`, pipes, sockets, and signals, and access resources such as files, devices, process status, and mount points.

This heterogeneity makes it difficult to apply existing detection methods for thread races or TOCTOU races to process races. For instance, static thread race detectors [43] work only with one program written in one language, and dynamic thread race detectors [128] work only with one process. To handle this heterogeneity, RACEPRO's race detection should be system-wide.

Process races are highly elusive. Many of the process races, including Bug 1 and 3 described in §3.2.2, occur only due to site-specific software, hardware, and user configurations. Moreover, many of the sampled process races, including all of those described in §3.2.2, occur only due to rare runtime factors. For example,

```
child = fork()
setjmp(loc)
p = wait(...) [blocks...]
... // child exits
p = wait(...) [...returns]
... // signaled
longjmp(loc)
p = wait(...) // error (no child)
```

Figure 3.2: **dash-MySQL Race.**

```
fd = open(H,RDONLY);
read(fd, buf, ...);
close(fd);
... // update buf
... // do work
fd = open(H,WRONLY|TRUNC);
write(fd, buf, ...);
close(fd);
```

Figure 3.3: **bash Race.**

Bug 1 only occurs when a database shutdown takes longer than usual, and Bug 2 only occurs when a signal is delivered right after a child process exited. These bugs illustrate the advantage of checking deployed systems, so that we can rely on real users to create the diverse configurations and executions to check.

Process race patterns. Classified by the causes, the 109 process races fall into two categories. Over two thirds (79) are execution order violations [67], such as Bug 1, 3, and 4 in §3.2.2, where a set of events are supposed to occur in a fixed order, but no synchronization operations enforce the order. Less than one third (30) are atomicity violations, including all TOCTOU bugs; most of them are the simplest load-store races, such as Bug 2 in §3.2.2. Few programs we studied use standard locks (*e.g.*, `flock`) to synchronize file system accesses among processes. These patterns suggest that a lockset-based race detection algorithm is unlikely to work well for detecting process races. Moreover, it is crucial to use an algorithm that can detect order violations.

3.2.2 Process Race Examples

Bug 1: Upstart-MySQL. `mysqld` does not cleanly terminate during system shutdown, and the file system becomes corrupted. This failure is due to an execution order violation where `S20sendsigs`, the shutdown script that terminates processes, does not wait long enough for MySQL to cleanly shutdown. The script then fails to unmount the file system which is still in use, so it proceeds to reboot the system without cleanly unmounting the file system. Its occurrence requires a combination of many factors, including the mixed use of Systems V initialization scripts and Upstart, a misconfiguration so that `S20sendsigs` does not wait for daemons started by Upstart, insufficient dependencies specified in MySQL's Upstart configuration file, and

a large MySQL database that takes a long time to shut down.

Bug 2: dash-MySQL. The shell wrapper `mysql_safe` of the MySQL server daemon `mysqld` goes into an infinite loop with 100% CPU usage after a MySQL update. This failure is due to an atomicity violation in `dash`, a small shell Debian uses to run daemons [37]. It occurs when `dash` is interrupted by a signal unexpectedly. Figure 3.2 shows the event sequence causing this race. To run a new background job, `dash` forks a child process and adds it to the job list of `dash`. It then calls `setjmp` to save an execution context and waits for the child to exit. After the child exits, `wait` returns, and `dash` is supposed to remove the child from the job list. However, if a signal is delivered at this time, `dash`'s signal handler will call `longjmp` to go back to the saved context, and the subsequent `wait` call will fail because the child's exit status has been collected by the previous `wait` call. The job list is still not empty, so `dash` gets stuck waiting for the nonexistent child to exit. Although this bug is in `dash`, it is triggered in practice by a combination of `dash`, the `mysql_safe` wrapper, and `mysqld`.

Bug 3: Mutt-OpenOffice. OpenOffice displays garbage when a user tries to open a Microsoft (MS) Word attachment in the Mutt mail client. This failure is due to an execution order violation when `mutt` prematurely overwrites the contents of a file before OpenOffice uses this file. It involves a combination of Mutt, OpenOffice, a user configuration entry in Mutt, and the `openoffice` shell script wrapper. The user first configures Mutt to use the `openoffice` wrapper to open MS Word attachments. To show an attachment, `mutt` saves the attachment to a temporary file, spawns the configured viewer in a new process, and waits for the viewer process to exit. The `openoffice` wrapper spawns the actual OpenOffice binary and exits at once. `mutt` mistakes this exit as the termination of the actual viewer, and overwrites the temporary file holding the attachment with all zeros, presumably for privacy reasons.

Bug 4: bash. The `bash` shell history is corrupted. This failure is due to an atomicity violation when multiple `bash` shells write concurrently to `.bash_history` without synchronization. When `bash` appends to the history file, it correctly uses `O_APPEND`. However, it also occasionally reads back the history file and overwrites it, presumably to keep the history file under a user-specified size. Figure 3.3 shows this problematic sequence of system calls. `bash` also runs this sequence when it exits. When multiple `bash` processes exit at the same time, the history file may be corrupted.

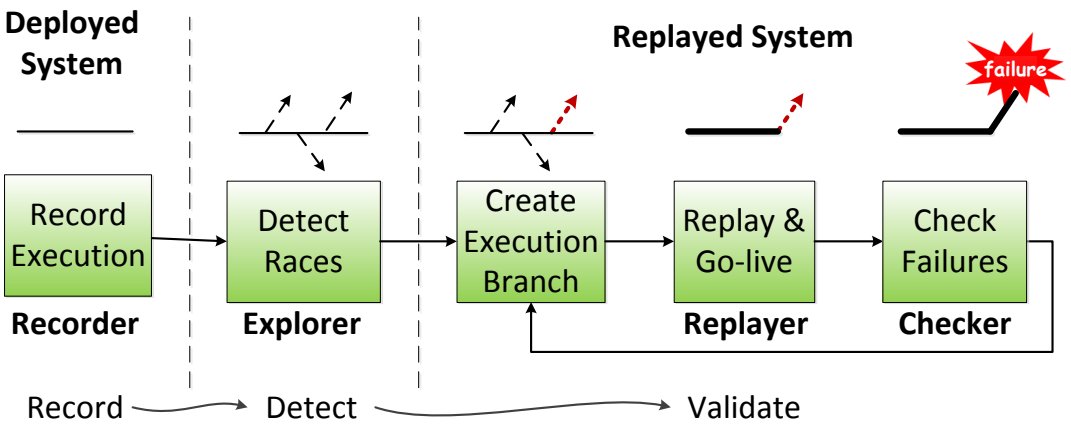


Figure 3.4: **RACEPRO Workflow.** Thin solid lines represent recorded executions; thick solid lines represent replayed executions. Dashed arrows represent potentially buggy execution branches. The dotted thick arrow represents the branch RACEPRO selects to explore.

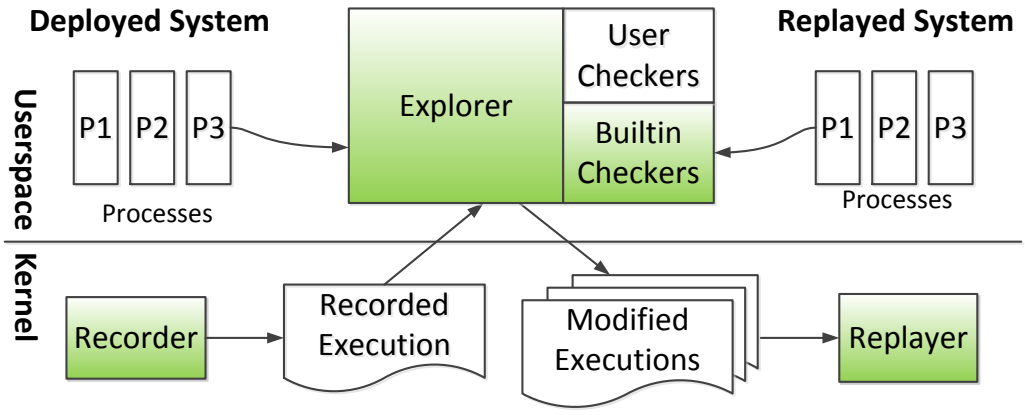


Figure 3.5: **RACEPRO Architecture.** Components are shaded. The recorder and the replayer run in kernel-space, and the explorer and the checkers run in user-space. Recorded executions and modified executions are stored in files.

3.3 Architecture Overview

RACEPRO is designed to automatically detect process races using the workflow shown in Figure 3.4. It consists of three steps, the first of which runs on the deployed system, while the latter two can run elsewhere on a separate replay system to avoid any performance impact on the deployed system. First, a *recorder* records the execution of a deployed system while the system is running and stores the recording in a log file. Second, an *explorer* reads the log and detects load-store and wait-wakeup races in the recorded execution. Third, each race is validated to determine if it is harmful. An execution branch of the recorded execution corresponding to each race is computed by systematically changing the order of system calls involved in the race. For each execution branch, a modified log is constructed that is used to replay execution with the changed order of system calls. A *replayer* replays the respective modified log up to the occurrence of the race, then causes it to resume live execution from that point onward. A set of built-in and user-provided *checkers* then check whether the execution results in misbehavior or a failure such as a segmentation fault. By examining the effects of a live execution, we distinguish harmful races from false or benign ones, thus reducing false positives [77; 97]. The live part of the re-execution is also recorded, so that users can deterministically replay detected bugs for debugging.

Figure 3.5 shows the RACEPRO architecture used to support its workflow. Of the four main architectural components, the recorder and the replayer run in kernel-space, and the explorer and checkers run in user-space. We will describe how RACEPRO records executions (§3.4) and detects (§3.5) and validates (§3.6) races using these components.

3.4 Recording Executions

RACEPRO's record-replay functionality builds on SCRIBE. This approach provides four key benefits for detecting process races. First, RACEPRO's recorder can record the execution of multiple processes and threads with low overhead on a deployed system so that the replayer can later deterministically replay that execution. This makes RACEPRO's *in vivo* checking approach possible by minimizing the performance impact of recording deployed systems. Second, RACEPRO's record-replay is application-transparent; it does not require changing, relinking, or recompiling applications or libraries. This enables RACEPRO to detect process races that are extremely heterogeneous involving many different programs written in different program languages. Third, RACEPRO's recorder operates at the OS-level to log sufficiently fine-grained accesses

to shared kernel objects so that RACEPRO's explorer can detect races regardless of high-level program semantics (§3.5). Finally, RACEPRO's record-replay records executions such that it can later transition from controlled replay of the recording to live execution at any point. This enables RACEPRO to distinguish harmful races from benign ones by allowing checkers to monitor an application for failures (§3.6.2).

To record the execution of multiprocess and multithreaded applications, RACEPRO records all non-deterministic interactions between applications and the OS and saves the recording as a log file. Signals and shared memory are recorded with the help of sync points as described in §2.6. We highlight how key interactions involving system calls are handled.

Unlike previous work [52; 102] that records and replays a total order of system calls, RACEPRO records and replays a partial order of system calls for speed. RACEPRO enforces no ordering constraints among system calls during record-replay unless they access the same kernel object and at least one of them modifies it, such as a `write` and a `read` on the same file. In that case, RACEPRO records the order in the kernel in which the object is accessed by the system calls and later replays the exact same order of accesses. This is done by piggybacking on the synchronization code that the kernel already has for serializing accesses to shared objects. These tracked accesses also help detect process races in a recorded execution (§3.5).

Table 3.2 lists the kernel objects tracked by RACEPRO. Most of the entries correspond one-to-one to specific low-level kernel resources, including inodes, files, file-tables, memory maps, and process credentials. The global entry corresponds to system-wide kernel objects, such as the hostname, file system mounts, system time, and network interfaces. For each such system-wide resource there is a unique global kernel object used to track accesses to that resource. The last two entries in the table, pid and ppid, provide a synchronization point to track dependencies on process states. For example, the pid entry of a process is used to track instances where the process is referenced by another process, *e.g.*, through a system call that references the process ID or through the `/proc` file system. The ppid entry is used to track when an orphan process is re-parented, which is visible through the `getppid` system call. Both pid and ppid correspond to identifiers that are visible to processes but cannot be modified explicitly by processes.

The recorder only tracks kernel objects whose state is visible to user-space processes, either directly or indirectly. For example, inode state is accessible via the system call `lstat`, and file-table state is visible through resolving of file descriptor in many system calls. RACEPRO does not track accesses to kernel objects which are entirely invisible to user-space. This avoids tracking superfluous accesses that may pollute the race detection results with unnecessary dependencies. For example, both the `fork` and `exit` system calls

Object	Description
inode	file, directory, socket, pipe, tty, pty, device
file	file handle of an open file
file-table	process file table
mmap	process memory map
cred	process credentials and capabilities, *e.g.*, user ID
global	system-wide properties (*e.g.*, hostname, mounts)
pid	process ID (access to process and `/proc`)
ppid	parent process ID (synchronize `exit`/`getppid`)

Table 3.2: **Shared Kernel Objects Tracked.**

access the kernel process table, but the order is unimportant to user-space. It only matters that the lifespan of processes is observed correctly, which is already tracked and enforced via the pid resource. If RACEPRO tracked accesses to the kernel process table, it would mistakenly conclude that every two `fork` system calls are "racy" because they all modify a common resource (§3.5). One complication with this approach is that if the kernel object in question controls assignment of identifiers (*e.g.*, process ID in the `fork` example), it may assign different identifiers during replay because the original order of accesses is not enforced. To address this problem, RACEPRO virtualizes identifiers such as process IDs to ensure the same values are allocated during replay as in the recording.

3.5 Detecting Process Races

RACEPRO flags a set of system calls as a race if (1) they are *concurrent* and therefore could have executed in a different order than the order recorded, (2) they access a common resource such that reordering the accesses may change the outcome of the execution. To determine whether a set of system calls are concurrent, RACEPRO constructs a happens-before [62] graph for the recorded execution (§3.5.1). To determine whether a set of system calls access common resources, RACEPRO obtains the shared kernel resources accessed by system calls from the log file and models the system calls as *load* and *store* micro-operations (§3.5.2) on those resources. RACEPRO then runs a set of happens-before based race detection algorithms to detect load-store and wait-wakeup races (§3.5.3).

3.5.1 The Happens-Before Graph

We define a partial ordering on the execution of system calls called *inherent* happens-before relations. We say that system call S_1 *inherently* happens-before system call S_2 if (1) S_1 accesses some resource before S_2 accesses that resource, (2) there is a dependency such that S_2 would not occur or complete unless S_1 completes, and (3) the dependency must be inferable from the system call semantics. For example, a `fork` that creates a child process inherently happens-before any system call in the child process, and a `write` to a pipe inherently happens-before a blocking `read` from the pipe. On the other hand, there is no inherent happens-before relation between a `read` and subsequent `write` to the same file.

RACEPRO constructs the happens-before graph using only inherent happens-before relations, as they represent the basic constraints on the ordering of system calls. Given a recorded execution, RACEPRO constructs a happens-before graph for all recorded system call events by considering pairs of such events. If two events S_1 and S_2 occur in the same process and S_2 is the next system call event that occurs after S_1, RACEPRO adds a directed edge $S_1 \rightarrow S_2$ in the happens-before graph. If two events S_1 and S_2 occur in two different processes, RACEPRO adds a directed edge $S_1 \rightarrow S_2$ in four cases:

- S_1 is a `fork` call, and S_2 is the corresponding `fork` return in the child process;
- S_1 is the `exit` of a child process, and S_2 is the corresponding `wait` in the parent;
- S_1 is a `kill` call, and S_2 is the corresponding signal delivery in the target process; or
- S_1 is a stream (*e.g.*, pipe or socket) write, and S_2 is a read from the same stream and the data written and the data read overlap.

We say that event S_1 *happens-before* S_2 with respect to a happens-before graph iff there is a directed path from S_1 to S_2 in the happens-before graph. Two events are *concurrent* with respect to a happens-before graph iff neither happens before the other.

RACEPRO also computes the vector-clocks [69] for all the system calls in the happens-before graph. By definition, the vector-clock of S_1 is earlier than the vector-clock of S_2 iff S_1 *happens-before* S_2 with respect to the graph, so comparing the vector-clocks of system calls is a fast and efficient way to test whether they are concurrent.

Our definition of inherent happens-before does not capture all dependencies that may constrain execution ordering. It may be missing happens-before edges that depend on the behavior of the application but cannot

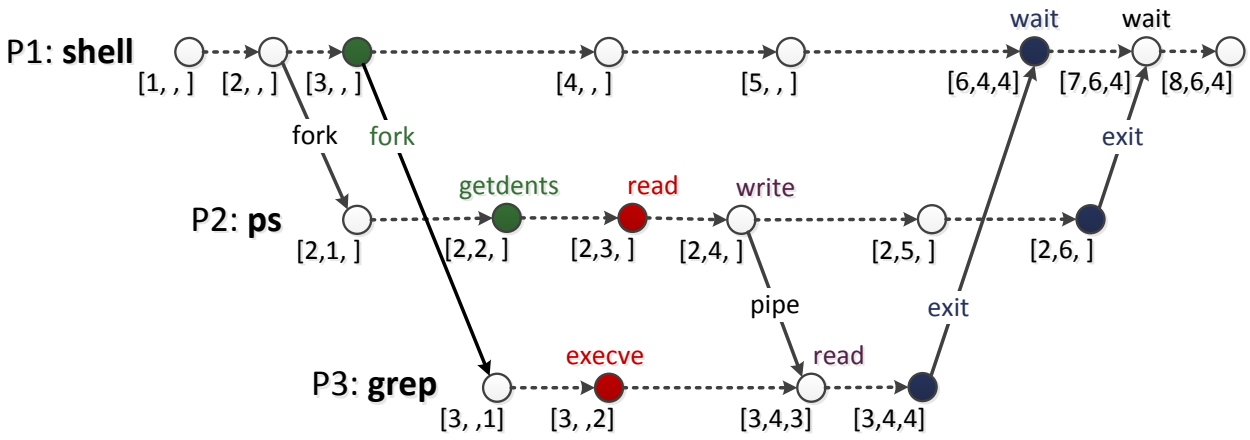


Figure 3.6: **The Happens-Before Graph for `ps | grep X`.** $P_{i=1,2,3}$ represent the processes involved. $[i, j, k]$ represent vector-clocks. The `read` of process P_2 and the `execve` of P_3 form a load-store race (§3.5.3), and so do the second `fork` of P_1 and the `getdents` (read directory entries) of P_2. The first `wait` of P_1 and the `exits` of P_2 and P_3 form a wait-wakeups race (§3.5.3). For clarity, not all system calls are shown.

be directly inferred from the semantics of the system calls involved. For example, the graph does not capture dependencies between processes via shared memory. It also does not capture dependencies caused by contents written to and read from files. For example, one can implement a fork-join primitive using read and write operations on a file. In some cases, such inaccuracies may make RACEPRO more conservative in flagging racy system calls and thereby identify impossible races. However, such cases will be filtered later by RACEPRO's validation step (§3.6) and will not be reported.

Figure 3.6 shows the happens-before graph for the example command `ps | grep X`. This command creates two child processes that access `grep`'s entry in the `/proc` directory: the process that runs `grep` modifies its command-line data when executed, and the process that runs `ps` reads that data. A race exists because both processes access the common resource in an arbitrary order, and the end result can be either N or $N + 1$ lines depending on that order.

Consider the `execve` system call in process P_3 and the `read` system call in process P_2. These two system calls are concurrent because there is no directed path between them in the graph. They both access a shared resource, namely, the inode of the file `cmd_line` in the directory corresponding to P_3 in `/proc`.

Therefore, these system calls are racy: depending on the precise execution order, `read` may or may not observe the new command line with the string "X". Similarly, the second `fork` in process P_1 and the `getdents` in process P_3 are also racy: `getdents` may or may not observe the newly created entry for process P_3 in the `/proc` directory.

In contrast, consider the pipe between P_2 and P_3. This pipe is a shared resource accessed by their `write` and `read` system calls, respectively. However, these two system calls are not racy because they are not concurrent. There exists a happens-before edge in the graph because a read from the pipe will block until data is available after a write to it.

3.5.2 Modeling Effects of System Calls

Existing algorithms for detecting memory races among threads rely on identifying concurrent load and store instructions to shared memory. To leverage such race detection algorithms, RACEPRO models the effects of a system call on the kernel objects that it may access using two micro-operations: *load* and *store*. These micro-operations are analogous to the traditional load and store instructions that are well-understood by the existing algorithms, except our micro-operations refer to shared kernel objects, such as inodes and memory maps, instead of an application's real shared memory.

More formally, we associate an abstract memory range with each kernel object. The effect of a system call on a kernel object depends on its semantics. If the system call only observes the object's state, we use a *load(obj,range)* operation. If it may also modify the object's state, we use a *store(obj,range)* operation. The argument *obj* indicates the affected kernel object, and the argument *range* indicates the ranges being accessed within that object's abstract memory. A single system call may access multiple kernel objects or even the same kernel object multiple times within the course of its execution.

We use a memory range for a shared kernel object instead of a single memory location because system calls often access different properties of an object or ranges of the object data. For instance, `lstat` reads the meta-data of files, while `write` writes the contents of files. They access a common object, but because they access distinct properties of that object, we do not consider them to race. Likewise, `read` and `write` system calls to non-overlapping regions in the same file do not race.

Memory ranges are particularly useful to model pathnames. Pathname creation and deletion change the parent directory structure and may race with reading its contents, but pathname creation, deletion, and lookup may only race with each other if given the same pathname. For example, both `creat(/tmp/a)`

and `unlink(/tmp/b)` may race with a `getdents` on `/tmp`, but are unrelated to each other or to an `lstat(/tmp/c)`. Modeling all pathname accesses using a single location on the parent directory's inode is too restrictive. Instead, we assign a unique memory location in the parent directory's inode for each possible pathname. We then model pathname creation and deletion system calls as stores to the designated location, pathname lookup system calls as loads from that location, and read directory system calls as loads from the entire pathname space under that directory.

Memory ranges are also useful to model *wait system calls* which may block on events and *wakeup system calls* which may trigger events. Example wait and wakeup system calls include `wait` and `exit`, respectively, and a blocking `read` from a pipe and a `write` to the pipe, respectively. To model the effect of wait and wakeup system calls, we use a special location in the abstract memory of the resource involved. Wait system calls are modeled as *loads* from that location, and wakeup system calls are modeled as *stores* to that location. For instance, the `exit` system call does a *store* to the special location associated with the parent process ID, and the `getppid` system call does a *load* from the same location.

Table 3.3 shows the template of micro-operations that RACEPRO uses to model nine common system calls: `open`, `write`, `read`, `getdents`, `execve`, `clone` (fork a process), `exit`, `wait`, and `getppid`. The `open` system call accesses several resources. It stores to the process file-table to allocate a new file descriptor, loads from the inodes of the directories corresponding to the path components, stores to the inode of the parent directory if the file is being created or loads from the file's inode otherwise, and stores to the entire data range of the inode if the file is being truncated.

The `write`, `read`, and `getdents` system calls access three resources: process file-table, file handle, and inode. `write` loads from the process file-table to locate the file handle, stores to the file handle to update the file position, stores to the meta-data of the file's inode in the file system, and stores to the affected data range of the file's inode. The last two micro-operations both affect the file's inode, but at different offsets. `read` from a regular file and `getdents` are similar to `write`, except that they load from the respective file's or directory's inode. `read` from a stream, such as a socket or a pipe, is also similar, except that it consumes data and thus modifies the inode's state, so it is modeled as a store to the corresponding inode.

The `execve` system call accesses several resources. It loads from the inodes of the directories corresponding to the path components. It also stores to the inodes of the `status` and `cmdline` files in the `/proc` directory entry of the process, to reflect the newly executed program name and command line.

Syscall	Micro-Op	Kernel Object
open	*store*	file-table
	load	inodes of path components
	store	inode of directory, if O_CREAT
	load	inode of file, if no O_CREAT
	store	data of file (range), if O_TRUNC
write	*load*	process file-table
	store	file handle of file
	store	inode of file
	store	data of file (range)
read	*load*	process file-table
	store	file handle of file
	load	inode of file, if regular file
	store	inode of file, if a stream
	load	data of file (range)
getdents	*load*	process file-table
	store	file handle of directory
	load	inode of directory
	load	data of directory (range)
execve	*load*	inodes of path components
	store	data of /proc/self/status
	store	data of /proc/self/cmdline
clone	*load*	process memory map
	store	data of /proc directory
exit	*store*	'pid' of self
	store	'ppid' of re-parented children
wait	*store*	data of /proc directory
	load	'pid' of reaped child
getppid	*load*	'ppid' of self

Table 3.3: **Micro-Operations of Common System Calls.**

The `clone`, `exit`, and `wait` system calls access two resources. `clone` loads from the process's memory map to create a copy for the newborn child, and stores to the `/proc` directory inode to reflect the existence of a new entry in it. `exit` stores to the pid resource of the current process to set the zombie state, and stores to the ppid resource of its children to reparent them to init. `wait` stores to the reaped child's pid resource to change its state from zombie to dead, and stores to the `/proc` directory inode to remove the reaped child's entry. RACEPRO detects races between `exit` and `wait` based on accesses to the exiting child's pid resource. Similarly, `getppid` loads from the current process's ppid resource, and RACEPRO detects races between `exit` and `getppid` based on accesses to the ppid resource.

To account for system calls that operate on streams of data, such as reads and writes on pipes and sockets, we maintain a virtual write-offset and read-offset for such resources. These offsets are advanced in response to write and read operations, respectively. Consider a stream object with write-offset L_W and read-offset L_R. A `write(fd,buf,n)` is modeled as a *store* to the memory range $[L_W..L_W + n]$ of the object, and also advances L_W by n. A `read(fd,buf,n)` is modeled as a *load* from the memory range $[L_R..L_R+\tilde{n}]$, where $\tilde{n} = min(L_W - L_R, n)$, and also advances L_R by $\tilde{n}$.

To account for the effects of signal delivery and handling, we model signals in a way that reflects the possibility of a signal to affect any system call, not just the one system call that was actually affected in the recording. We associate a unique abstract memory location with each signal. A `kill` system call that sends a signal is modeled as a *store* to this location. Each system call in the target process is considered to access that location, and therefore modeled as a *load* from all the signals. This method ensures that any system call that may be affected by a signal would access the shared object that represents that signal.

3.5.3 Race Detection Algorithms

Building on the happens-before graph and the modeling of system calls as micro-operations, RACEPRO detects three types of process races: load-store races (§3.5.3), wait-wakeups races (§3.5.3), and wakeup-waits races (§3.5.3). RACEPRO may also be extended to detect other types of races (§3.5.3).

Load-Store Races A load-store race occurs when two system calls concurrently access the same shared object and at least one is a *store* operation. In this case, the two system calls could have executed in the reverse order. RACEPRO flags two system calls as a load-store race if (1) they are concurrent; (2) they access the same shared kernel object, and (3) at least one access is a *store*. In the `ps | grep X` example

shown in Figure 3.6, the system calls `read` and `execve` are flagged as a race because they are concurrent, they access the same resource, and at least one, `execve`, does a *store*. In contrast, the system call `exit` of P_3 also stores to the same resource, but is not flagged as a race because it is not concurrent with any of them as `read` happens-before `exit` and `execve` happens-before `exit`.

RACEPRO detects load-store races using a straightforward happens-before-based race detection algorithm. We chose a happens-before over lockset because processes rarely use standard locks (§3.2). RACEPRO iterates through all the shared kernel objects in the recording. For each shared object, it considers the set of all accesses to that object by all system calls, and divides this set into per-process lists, such that the list L_i of process P_i contains all the accesses performed by that process. RACEPRO now looks at all pairs of processes, P_i, P_j, $i \neq j$, and considers their accesses to the object. For each access $S_n \in L_i$, it scans through the accesses $S_m \in L_j$. If the vector-clocks of S_n and S_m are concurrent, the pair of system calls is marked as a race. If $S_n \rightarrow S_m$, then $S_n \rightarrow S_{m+k}$, so the scan is aborted and the next access $S_{n+1} \in L_i$ is considered. If $S_m \rightarrow S_n$, then $S_m \rightarrow S_{n+k}$, so $S_{m+1} \in L_j$ is saved so that the next scan of accesses from L_j will start from S_{m+1}, since we know that earlier events happened-before all remaining accesses in L_i.

Because system calls may access more than one shared object during their execution, it is possible that the same pair of system calls will be marked more than once. For example, two `write` system calls from different processes to the same location in the same file will be marked twice, once when the meta-data of the inode is considered, and once when the data of the file is considered. Because RACEPRO detects and later validates (§3.6) races at the granularity of system calls, it only reports the respective pair of system calls once.

RACEPRO may produce a myriad of races, which can take a long time to produce and later validate. To address this concern, RACEPRO prioritizes which races to examine in two ways. First, RACEPRO may defer or entirely skip races that are less likely to prove harmful, depending on the system calls and resource involved. For example, when analyzing the execution of a parallel compilation, resources related to visual output may be skipped: although many processes may be writing to the standard output, races, if they exist, are likely to be benign. Second, RACEPRO ranks pairs of system calls according to their distance from each other in the happens-before graph, and examines nearer system calls first.

Wait-Wakeups Races A wait-wakeups race occurs when a wait system call may be woken up by more than a single matching wakeup system call. If the wakeup system calls executed in a different order, the

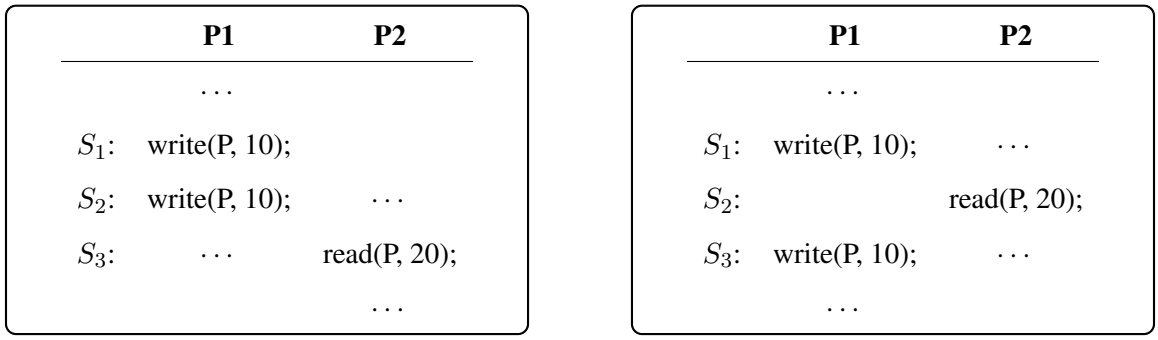
wait system call could have picked a different wakeup than in the original execution. Wait-wakeups races involve at least three system calls. For instance, a `wait` system call which does not indicate a specific process identifier to wait for will complete if any of its children terminate. Likewise, a blocking `read` from a stream will complete after any `write` to the stream.

In these cases, the wait system call essentially uses a *wildcard* argument for the wakeup condition so that there can be multiple system calls that match the wakeup condition depending on their order of execution. The wait-wakeups race requires a wildcard, otherwise there is only a single matching system call, and thus a single execution order. For instance, a `wait` system call that requests a specific process identifier must be matched by the exit of that process. In this case, the wait-wakeup relationship implies an inherent happens-before edge in the happens-before graph, since the two system calls must always occur in that order.

RACEPRO flags three system calls as a wait-wakeups race if (1) one is a wait system call, (2) the other two are wakeup system calls that match the wait condition, and (3) the wait system call did not happen-before any of the wakeup system calls. In the `ps | grep X` example shown in Figure 3.6, the two `exit` system calls of P_2 and P_3 and the first `wait` system call of P_1 are flagged as a wait-wakeups race since both `exit` calls are concurrent and can match the `wait`. In contrast, the `write` and `read` system calls to and from the pipe are not flagged as a race, because there does not exist a second wakeup system call that matches the `read`.

RACEPRO detects wait-wakeups races using an algorithm that builds on the load-store race detection algorithm, with three main differences. First, the algorithm considers only those accesses that correspond to wait and wakeup system calls by looking only at locations in the abstract memory reserved for wait and wakeup actions. Second, it considers only pairs of accesses where one is a *load* and the other is a *store*, corresponding to one wait and one wakeup system calls. The wait system call must not happen-before the wakeup system call. Third, for each candidate pair of wait and wakeup system calls S_1 and S_2, RACEPRO narrows its search to the remaining wakeup system calls that match the wait system call by looking for system calls that store to the same abstract memory location. For each matching wakeup system call S_3, RACEPRO checks whether it would form a wait-wakeups race together with S_1 and S_2.

The relative order of the wakeup system calls may matter if their effect on the resource is cumulative. For instance, Figure 3.7 depicts a cumulative wait-wakeups scenario in which the order of two `write` system calls to the same stream determines what a matching `read` would observe. A `read` from a stream may return less data than requested if the data in the buffer is insufficient. In Figure 3.7a, a blocking `read` occurs



(a) (b)

Figure 3.7: **Wait-Wakeups Races in Streams.**

after two `writes` and consumes their cumulative data. However, in Figure 3.7b, the `read` occurs before the second `write` and returns the data only from the first `write`. Note that S_2 and S_3 in Figure 3.7a do not form a load-store race as S_2 inherently happens-before S_3. Thus, RACEPRO flags either case as a wait-wakeups race. The relative order of the wakeup system calls does not matter if their effect on the resource is not cumulative, such as with `wait` and `exit` system calls.

Wakeup-Waits Races A wakeup-waits race occurs when a wakeup system call may wake up more than a single matching wait system call. Like wait-wakeups races, wakeup-waits races involve at least three system calls. For example, a `connect` system call to a listening socket will wake up any processes which may have a pending `accept` on that socket; the popular Apache Web server uses this method to balance incoming requests. As another example, a signal sent to a process may interrupt the process during a system call. Depending on the exact timing of events, the signal may be delivered at different times and interrupt different system calls.

Some wakeup system calls only affect the first matching wait system call that gets executed; that system call "consumes" the wakeup and the remaining wait system calls must wait for a subsequent wakeup. Examples include `connect` and `accept` system calls, and `read` and `write` system calls on streams. In contrast, when two processes monitor the same file using the `select` system call, a file state change will notify both processes equally. Even in this case, a race exists as the behavior depends on which wait system calls executes first.

RACEPRO flags three system calls as a wakeup-waits race if (1) one is a wakeup system call, (2) the other two are wait system calls that match the wakeup, (3) the wait system calls did not happen-before the wakeup

system call. To detect wakeup-waits races, RACEPRO builds on the wait-wakeups race detection algorithm with one difference. For each candidate pair of wait and wakeup system calls S_1 and S_2, RACEPRO narrows its search to the remaining wait system calls that match the wakeup system call by looking for system calls that load from the same abstract memory location. For each matching wait system call S_3, RACEPRO checks whether it would form a wakeup-waits race together with S_1 and S_2.

Many-System-Calls Races RACEPRO's algorithms handle races that involve two system calls for load-store races, and three system calls for both wait-wakeups and wakeup-waits races. However, it is also possible that a race involves more system calls. For example, consider a load-store race that comprises a sequence of four system calls that only if executed in the reverse order, from last to first, will produce a bug. RACEPRO's algorithm will not detect this load-store race since it only considers one pair of system calls at a time. To detect such races, the algorithms can be extended to consider more system calls at a time and more complex patterns of races. An alternative approach is to apply RACEPRO's analysis recursively on modified executions (§3.6.2).

3.6 Validating Races

A detected process race may be either benign or harmful, depending on whether it leads to a failure. For instance, consider the `ps | grep X` example again which may output either N or $N+1$ lines. When run from the command line, this race is usually benign since most users will automatically recognize and ignore the difference. However, for applications that rely on one specific output, this race can be harmful and lead to a failure (§3.7).

To avoid false positives, RACEPRO validates whether detected races are harmful and reports only harmful races as bugs. For each race, it creates an execution branch in which the racy system calls, which we refer to as *anchor* system calls, would occur in a different order from the original recorded execution (§3.6.1). It replays the modified execution until the race occurs, then makes the execution go-live (§3.6.2). It checks the live execution for failures (§3.6.3), and, if found, reports the race as a bug.

3.6.1 Creating Execution Branches

RACEPRO does not replay the original recorded execution, but instead replays an execution branch built from the original execution in a controlled way. The execution branch is a truncated and modified version

of the original log file. Replaying such modified log file is a departure from deterministic record-replay presented in the previous chapter with SCRIBE as we no longer aim to reproduce a past execution verbatim. Given a detected race which, based on its type, involves two or three anchor system calls, RACEPRO creates an execution branch in two steps. First, it copies the sequence of log events from the original execution recording *up to* the anchor system calls. Then, it adds the anchor system calls with suitable ordering constraints so that they will be replayed in an order that makes the race resolve differently than in the original recorded execution. The rest of the log events from the original execution are not included in the modified version.

A key requirement in the first step above is that the definition of *up to* must form a *consistent cut* [69] across all the processes to avoid deadlocks in replay. A consistent cut is a set of system calls, one from each process, that includes the anchor system calls, such that all system calls and other log events that occurred before this set are on one side of the cut. For instance, if S_1 in process P_1 happens-before S_2 in process P_2 and we include S_2 in the consistent cut, then we must also include S_1 in the cut.

To compute a consistent cut for a set of anchor system calls, RACEPRO simply merges the vector-clocks of the anchor system calls into a unified vector-clock by taking the latest clock value for each process. In the resulting vector-clock, the clock value for each process indicates the last observed happens-before path from that process to any of the anchor system calls. By definition, the source of this happens-before edge is also the last system call of that process that must be included in the cut. For instance, the unified vector-clock for the `read` and `execve` race in Figure 3.6 is $[3, 3, 2]$, and the consistent cut includes the second `fork` of P_1, `read` of P_2, and `execve` of P_3.

Given a consistent cut, RACEPRO copies the log events of each process, except the anchor system calls, until the clock value for that process is reached. It then adds the anchors in a particular order. For load-store races, there are two anchor system calls. To generate the execution branch, RACEPRO simply flips the order of the anchors compared to the original execution; it first adds the system call that occurred *second* in the original execution, followed by the one that occurred *first*. It also adds an ordering constraint to ensure that they will be replayed in that order.

For wait-wakeups races, there are three anchor system calls: two wakeup system calls and a wait system call. To generate the execution branch, RACEPRO first adds both wakeup system calls, then adds a modified version of the wait system call in which its wildcard argument is replaced with a specific argument that will match the wakeup system call that was *not* picked in the original execution. For example, consider a

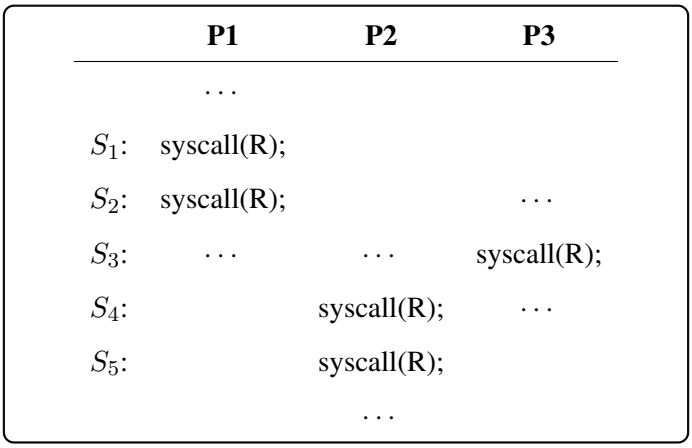


Figure 3.8: **Replay Divergence Due to Reordering.**

race with two child processes in `exit`, either of which may wake up a parent process in `wait`. RACEPRO first adds both `exit` system calls, then the `wait` system call modified such that its wildcard argument is replaced by a specific argument that will cause this `wait` to pick the `exit` of the child that was not picked in the original execution. It also adds a constraint to ensure that the parent will execute after that child's `exit`. The other child is not constrained.

For wakeup-waits races, there are also three anchor system calls: one wakeup system call and two wait system calls. To generate the execution branch, RACEPRO simply flips the order of the two wait system calls compared to the original execution. Races that involve signals, which may be delivered earlier or later than in the original execution, are handled differently. To generate an execution branch for a signal to be delivered earlier, RACEPRO simply inserts the signal delivery event at an earlier location which is thereby considered one of the anchors of the consistent cut. In contrast, delivering a signal arbitrarily later is likely to cause replay divergence (§3.6.2). Instead, RACEPRO only considers delivering a signal later if it interrupted a system call in the recorded execution, in which case the signal is instead delivered promptly after the corresponding system call completes when replayed.

Reordering of the anchor system calls may also imply reordering of additional system calls that also access the same resources. Consider the execution scenario depicted in Figure 3.8, which involves three processes and five system calls that access the same resource. The system calls S_1 and S_5 form a load-store race. To generate the modified execution for this race, RACEPRO will make the following changes: (1) it will include S_1 but not S_2, because system calls following the anchors remain outside the cut and are truncated; (2) it will reorder S_5, and therefore S_4 too, with respect to S_1; and (3) depending on the consistent cut, it

will either exclude S_3 or reorder S_3 with respect to S_1. RACEPRO adjusts the modified recording so that it will enforce the new partial order of system calls instead of the partial order of system calls in the original execution.

3.6.2 Replaying Execution Branches and Going Live

RACEPRO's replayer provides deterministic replay of the originally recorded execution and also ensures that successful replay of a modified execution is also deterministic. Given a modified execution, RACEPRO replays each recorded event while preserving the partial order indicated by the recording. The last events replayed are the anchor system calls. To force races to resolve as desired, RACEPRO replays the anchor system calls serially, one by one, while holding the remaining processes inactive. From that point onward, it allows the processes to go live to resume normal execution.

Go Live. The ability to go live by resuming live execution from a replay is fundamental for allowing RACEPRO to validate whether races manifest into real bugs or not, and thereby avoid reporting false-positives. To go live, RACEPRO faces two challenges. First, RACEPRO must ensure that replayed processes perceive the underlying system to be the same as at the time of recording. For example, system identifiers such as process IDs must remain the same for processes to run correctly after they transition to live execution. RACEPRO leverages OS virtualization to encapsulate processes in a virtual execution environment that provides the same private, virtualized view of the system when the session is replayed or goes live as when it was recorded. Processes only see virtual identifiers that always stay the same so that the session can go live at any time. Second, RACEPRO needs to not only replay the application state in user-space, but also the corresponding state that is internally maintained by the operating system for the processes. For example, actions such as creating a pipe and writing to it must be done as is so that the pipe exists and has suitable state should the process transition to live execution.

RACEPRO works best when a go-live execution requests no inputs from users or external processes; such executions include parallel make, parallel boot, and executions of non-interactive programs. If a go-live execution requests external inputs, RACEPRO tries to replay the inputs recorded from the original execution. Currently RACEPRO replays standard inputs from users and pipe or socket data received from external processes. It does not replay data read from the file system. Instead, it checkpoints the file system before recording an execution and restores to this checkpoint before each replay, using unionfs [88], which has low overhead. Replaying inputs may not always work because the go-live execution differs from the original

execution, but we have not found it a problem in our evaluation because tightly coupled processes should be recorded together anyway.

RACEPRO can be applied recursively to detect races involving more system calls (§3.5.3). Since it already records the go-live portion of modified executions, doing so is as easy as running the same detection logic on these new recordings. This essentially turns RACEPRO into a model checker [45]. However, we leave this mode off by default because exhaustive model checking is quite expensive and it is probably more desirable to spend limited checking resources on real executions over the fake checking-generated executions.

Replay Divergence. RACEPRO's replayer may not be able to replay some execution branches due to *replay divergence*. This can result from trying to replay a modified recording instead of the original recording. Replay divergence occurs when there is a mismatch between the actual actions of a replayed process and what is scripted in the execution recording. The mismatch could be between the actual system call and the expected system call or, even if the system calls match, between the resources actually accessed by the system call and the resources expected to be accessed. When a divergence failure occurs for some execution branch, RACEPRO does not flag the corresponding race as a bug because it lacks evidence to that end.

Divergence is commonly caused when the reordering of the anchor system calls implies reordering of additional system calls that also access the same resources. Consider again the execution scenario depicted in Figure 3.8 in which the system calls S_1 and S_5 form a load-store race and the modified execution branch reorders the systems calls as S_3, S_4, S_5, and S_1 while dropping S_2 as being outside the cut. A replay divergence may occur if the execution of S_5 depended on S_2 which was dropped out, or if the execution of S_4 depends on S_1 which was reordered with respect to S_4. Figure 3.9a illustrates the former scenario. Reordering the two `creat` system calls would cause P_2 to call `unlink` before P_1's `creat`. The call will fail and P_2 will not call `creat` and thus diverge from the recorded execution.

Divergence can also be caused when processes rely on a specific execution ordering of system calls in a way that is not tracked by RACEPRO. Figure 3.9b illustrates one such scenario where process P_1 executes system call S_1 to write data to a file, and process P_2's execution depends on data read from file by S_2. If P_2 depends on the specific data written by S_1, then reordering S_1 and S_2 will almost certainly cause a divergence. Were the dependency on the file's content considered an inherent happens-before $S_1 \rightarrow S_2$, RACEPRO's explorer would not have flagged the race in the first place. However, it is prohibitively expensive, and in some cases impossible, to track generic semantics of applications.

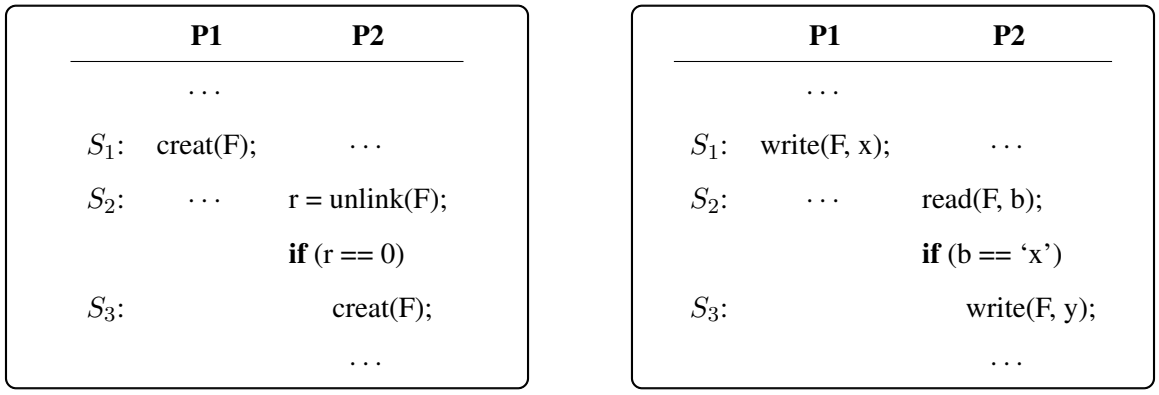


Figure 3.9: **Replay Divergence Examples.**

Another cause for divergence is use of shared memory. Recall that shared memory accesses are tracked by the recorder and enforced by the replayer. However, reordering of system calls may lead to reordering of shared memory accesses as well, which will certainly lead to replay divergence. RACEPRO mitigates this effect by permitting *relaxed* execution from where the reordering takes place. In this mode the replayer does not enforce memory access ordering, but continues to enforce other ordering constraints such as partial ordering of system calls. This improves the chances that the replayed execution reach the point of go-live. However, accesses to shared memory may now resolve arbitrarily and still cause divergence. For this reason RACEPRO is likely to be less effective in finding races on OS resources between threads of the same process. We believe that such races are relatively unlikely to occur.

Replay divergence is reportedly a serious problem for a previous race classifier [77], where it can occur for two reasons: the race being validated does occur and causes the execution to run code or access data not recorded originally, or the race being validated cannot occur and is a false positive. In contrast, replay divergence actually *helps* RACEPRO to distinguish root-cause races from other races. By relying on a replay followed by transition to live execution, RACEPRO is no longer concerned with the first scenario. If replay diverges, RACEPRO can tell that the race is a false positive and discard it.

Moreover, if the divergence is not due to untracked interactions or shared memory discussed above (or file locking, also untracked by RACEPRO), then there must exist another race that is "tighter" than the one being validated. The other race may involve the same resource or a different one. For example, in Figure 3.9b the race between S_1 and S_3 causes divergence because of another race between S_1 and S_2. The latter race is "tighter" in the sense that S_2 is closer to S_1 because $S_2 \rightarrow S_3$; the race between S_1 and S_2

Bug ID	Description
debian-294579	concurrent `adduser` processes read and write `/etc/passwd` without synchronization, corrupting this file
debian-438076	`mv` unlinks the target file before calling atomic `rename`, violating the atomicity requirement on `mv`
debian-399930	`logrotate` creates a new file then sets it writable, but deamons may observe it without write permissions
redhat-54127	`ps | grep` race causes a wrong version of `licq 7.3` to be started
launchpad-596064	`upstart` does not wait until `smbd` creates a directory before spawning `nmbd`, which requires that directory
launchpad-10809	`bash` updates the history file without synchronization, corrupting this file
new-1	`tcsh 6.17` updates the history file without synchronization, even when "savehist merge" is set
new-2	`updatedb` removes old database before renaming the new one, so `locate` finds nothing (`findutils 4.4.2`)
new-3	concurrent `updatedb` processes may cause the database to be empty
new-4	incorrect dependencies in Makefile of `abr2gbr 1.0.3` may causes compilation failure

Table 3.4: **Bugs Found by RACEPRO.** Bugs are identified by "distribution - bug ID". New bugs are identified as "new - bug number"

.

subsumes the race between S_1 and S_3. In other words, discarding races that cause replay divergence helps RACEPRO to find root-cause races. We believe the go-live mechanism can benefit existing replay-based thread-race classifiers.

3.6.3 Checking Execution Branches

When the replay of an execution branch switches to live execution, RACEPRO no longer controls the execution. Rather, it records the execution from that point on, and activates a checker to monitor the execution for failures or incorrect behavior. If the checker detects a failure that did not occur during recording, it reports a bug and saves the combined execution recording, consisting of the original recording followed by the new

recording, so that users can deterministically replay it for debugging.

RACEPRO provides a set of built-in checkers to detect bad application behavior. The built-in checker can detect erroneous behavior such as segmentation faults, infinite loops (via timeouts), error messages in system logs, and failed commands with non-zero exit status. In addition, RACEPRO can also run system-provided checker programs such as `fsck`.

Moreover, RACEPRO allows users to plug in domain-specific checkers. To do so, a user need only provide a program or even a shell script that will run concurrently along the live execution. For instance, such scripts could compare the output produced by a modified execution to that of the original execution, and flag significant differences as errors. It is also possible to use existing test-suites already provided with many application packages. These test-suites are particularly handy if the target application is a server. For instance, both the Apache web server and the MySQL database server are shipped with basic though useful test suites, which could be executed against a modified server. Finally it may also compare the output of the go-live execution with a linearized run [46].

By running checkers on live executions, RACEPRO guarantees that observed failures always correspond to real executions, thus eliminating false positives if the checkers are accurate. Moreover, the process races RACEPRO detects are often the root cause of the failures, aiding developers in diagnosis. In rare cases, after a modified execution goes live, it may encounter an unrelated bug. RACEPRO still provides an execution recording useful for debugging, but without pointing out the root-cause.

As in many other checking frameworks, RACEPRO can detect only what is checked. Although its built-in checkers can detect many errors (§3.7.1), it may miss domain-specific "silent" corruptions. Fortunately, recent work has developed techniques to check advanced properties such as conflict serializability or linearizability [46], which RACEPRO can leverage.

RACEPRO may have false negatives. A main source is that RACEPRO is a dynamic tool, thus it may miss bugs in the executions that do not occur. Fortunately, by checking deployed systems, RACEPRO increases its checking coverage. A second source is checker inaccuracy. If a checker is too permissive or no checker is provided to check for certain failures, RACEPRO would miss bugs.

While our checkers benefit from running on live executions, RACEPRO is unable to replay production workload recorded after the race is triggered. RACEPRO would benefit from a built-in checker that would replay the rest of the recorded execution. The ability to replay production workload after the race is triggered would provide confidence that the race is benign. Replaying a recorded execution after introducing modi-

Name	Statistics			Number of Races			
	Processes	Syscalls	Resources	Detected	Diverged	Benign	Harmful
debian-294579	19	5275	658	4232	3019	1171	42
debian-438076	21	1688	213	50	0	46	4
debian-399930	10	1536	279	17	0	13	4
redhat-54127	14	1298	229	35	15	16	4
launchpad-596064	34	5564	722	272	267	3	2
launchpad-10809	13	1890	205	143	117	16	10
new-1	12	2569	201	137	90	33	14
new-2	47	2621	467	82	13	27	42
new-3	30	4361	2981	17	0	13	4
new-4	19	4672	716	8	0	7	1

Table 3.5: **Bug Detection Statistics.** *Processes* is the number of processes, *Syscalls* the number of system calls occured, and *Resources* the number of distinct shared resources tracked in the recorded executions. For races, *Detected* is the number of races detected by RACEPRO, *Diverged* the races for which the replay diverged (*i.e.*, false positive), *Benign* the benign races, and *Harmful* harmful races that led to failures.

fications is difficult due to the unknown behavior and side-effects of the application during the divergence. We explore the feasibility of such mutable replay capability in the next chapter with DORA.

3.7 Experimental Results

We have implemented a RACEPRO prototype in Linux. The prototype consists of Linux kernel components for record, replay, and go-live, and a Python user-space exploration engine for detecting and validating races. The current prototype has several limitations. For replaying executions and isolating the side effects of replay, RACEPRO must checkpoint system states. It currently checkpoints only file system states, though switching to better checkpoint mechanism [81] is straightforward. RACEPRO detects idle state simply by reading `/proc/loadavg`, and can benefit from a more sophisticated idle detection algorithm [114].

Using the RACEPRO prototype, we demonstrated its functionality in finding known and unknown bugs, and measured its performance overhead. For our experiments, the software used for RACEPRO was Linux kernel 2.6.35, Python 2.6.6, Cython 0.14, Networkx 1.1-2, and UnionFs-Fuse 0.23.

3.7.1 Bugs Found

We evaluated RACEPRO's effectiveness by testing to see if it could find both known and unknown bugs. To find known bugs, we used RACEPRO on 6 bugs from our study. Bugs were selected based on whether we could find and compile the right version of the software and run it with RACEPRO. Some of the bugs in §3.2 are in programs that we cannot compile, so we excluded them from the experiments. For each known bug, we wrote a shell script to perform the operations described in the bug report, without applying any stress to make the bug easily occur. We ran this shell script without RACEPRO 50 times, and observed that the bug never occurred. We then ran RACEPRO with the script to detect the bug.

To find unknown bugs, we used four commonly used applications. We applied RACEPRO to the `locate` utility and `updatedb`, a utility to create a database for `locate`. These two utilities are commonly used and well tested, and they touch a shared database of file names, thus they are likely to race with each other. Inspired by the history file race in `bash`, we applied RACEPRO to `tcsh`. `tcsh` has a "savehist merge" option, which should supposedly merge history files from different windows and sessions. Because compilation of software packages often involves multiple concurrent and inter-dependent processes, we also applied RACEPRO to the `make -j` command.

Table 3.4 shows all the bugs RACEPRO found. RACEPRO found a total of 10 bugs, including all of the known bugs selected and 4 previously unknown bugs. We highlight a few interesting bugs. Of the known bugs, the debian-294579 bug is the most serious: it leads to corruption of `/etc/passwd` since `adduser` does not synchronize concurrent reads and writes of `/etc/passwd`. This bug was triggered when an administrator tried to import users from `OpenLDAP` to a local machine.

The redhat-54127 bug is due to the `ps | grep X` race. Instant messenger program `licq` uses `ps | grep` to detect whether KDE or Gnome is running. Due to the race in `ps | grep`, `licq` sometimes believes a windows manager is running when it in fact is not, thus loading the wrong version of `licq`.

The 4 previously unknown bugs were named `new-1`, `new-2`, `new-3`, and `new-4`. In the `new-1` bug, RACEPRO found that `tcsh` writes to its history file without proper synchronization, even when "savehist merge" is set. This option is supposed to merge history across windows and sessions, but unfortunately, it is not implemented correctly.

In the `new-2` bug, RACEPRO found that when `locate` and `updatedb` run concurrently, `locate` may observe an empty database and return zero results. The reason is that `updatedb` unlinks the old database, before calling `rename` to replace it with the new database. This unlink is unnecessary as `rename`

guarantees atomic replacement of the destination link.

In the `new-3` bug, RACEPRO found that when multiple instances of `updatedb` run concurrently, the resultant database may be corrupted. Multiple `updatedb` processes may exist, for example, when users manually run one instance while `cron` is running another. While `updatedb` carefully validates the size of the new database before using it to replace the old one, the validation and replacement are not atomic, and the database may still be corrupted.

In the `new-4` bug, RACEPRO found that in the compilation of `abr2gbr`, a package to convert between image formats, the build process may fail when using `make -j` for parallel compilation. The reason is that the dependencies defined in the Makefile are incomplete, which produces a race condition between the creation of an `$OBJDIR` directory and the use of that directory to store object files from the compilation.

3.7.2 Bug Statistics

Table 3.5 shows various statistics for each detected bug, including the number of processes involved (Processes), the number of system calls recorded (Syscalls), the number of unique shared resources tracked (Resources), the total number of races detected (Races), the number of races in which the replay diverged (Diverged), the number of benign races (Benign), and the number of harmful races (Harmful). The number of processes tends to be large because when running a shell script, the shell forks a new process for each external command. The number of system calls in the recorded executions ranges from 1,298 to 5,564. The number of distinct shared resources accessed by these system calls ranges from 201 to 2,981.

The number of races that RACEPRO detects varies across different bugs. For instance, RACEPRO detected only 17 races for debian-399930, but it detected over 4,000 races for debian-294579. Typically only a small number of races are harmful, while the majority are benign, as shown by the Benign column. In addition, RACEPRO effectively pruned many false positives as shown by the Diverged column. These two columns together illustrate the benefit of the replay and go-live approach.

The mapping between harmful races and bugs is generally many-to-one. There are multiple distinct races that produce the same or similar failures due to a common logical bug. There are two main reasons why a single programming error may result in multiple races. First, a bug may occur in a section of the code that is executed multiple times, for instance in a loop, or in a function called from multiple sites. Thus, there can be multiple races involving distinct instances of the same resource type; RACEPRO will detect and validate each independently. Second, a bug such as missing locks around critical sections may incorrectly

Name	Execution Times [seconds/race]			
	Record	Replay	Generate	Validate
debian-294579	2.47	2.43	3.42	2.92
debian-438076	3.76	0.75	0.84	2.87
debian-399930	0.59	0.57	0.75	0.84
redhat-54127	0.27	0.25	0.66	0.41
launchpad-596064	21.45	3.11	2.49	1.70
launchpad-10809	0.27	0.25	0.81	0.44
new-1	0.56	0.54	1.52	0.76
new-2	0.89	0.88	1.44	1.16
new-3	2.63	2.61	2.34	2.98
new-4	1.01	0.98	4.81	1.35

Table 3.6: **RACEPRO Execution Times.** *Record* and *Replay* are the times to record and replay the executions, respectively. *Generate* is the average time to generate an execution branch and *Validate* the average time to validate a race.

allow reordering of more than two system calls, and each pair of reordered system calls could produce a distinct race.

In most cases, we relied on built-in checkers in RACEPRO to detect the failures. For instance, RACEPRO caught bug launchpad-596064 by using `grep` to find error messages in standard daemon logs, and it caught bugs debian-438076, debian-399930, new-2, new-3, and new-4 by checking for the exit status of programs. Writing checkers to detect other cases was also easy, and required just one line in all cases. For example, for debian-294579, launchpad-10809, and new-1, we detected the failures simply using a `diff` of the old and new versions of the affected file.

3.7.3 Performance Overhead

Low recording overhead is crucial because RACEPRO runs with deployed systems. Low replay overhead is desirable because RACEPRO can check more execution branches within the same amount of time. Since RACEPRO shares its record-replay engine with SCRIBE, the same recording overhead is observed. Our

results show that RACEPRO's recording overhead is under 2.5% for server and under 15% for desktop applications 2.7. Replay speed was in all cases at least as fast as native execution and in some cases up to two orders of magnitude faster. This speedup is particularly useful for enabling rapid race validation. Replay speedup stems from omitted in-kernel work due to system calls partially or entirely skipped, and waiting time skipped at replay. Applications that do neither operations perform the same work whether recording or replaying, and sustain speedups close to 1.

We also measured various overhead statistics involved in finding the bugs listed in Table 3.6. These measurements were done on an HP DL360 G3 server with dual 3.06 GHz Intel Xeon CPUs, 4 GB RAM, and dual 18 GB local disks. For each bug, Table 3.6 shows the time to record the execution (Record) and to replay it (Replay), the average time to generate an execution branch for a race from a recorded execution (Generate), and the average time to validate an execution branch for a race (Validate).

In all cases, recording execution times were within 3% of the original execution times without recording, and replaying the execution took less time than the original recorded execution. Replay time for each recording ranged from 250 ms to 1.8 s, providing an upper limit on the time to replay execution branches. Replaying execution branches is generally faster because those branches are truncated versions of the original execution. Replay speedup was near 1 in most cases, but was as high as 7 times for launchpad-596064 due to very long idle times as part of starting up the workload. These results are in line with our other record-replay results for desktop and server applications. In particular, the results demonstrate that RACEPRO recording overhead is low enough to enable its use on deployed systems.

The time for our unoptimized prototype to detect all races was under 350 ms for most bugs, but in some cases as much as 3.8 s. This time correlates roughly with the number of unique shared kernel objects tracked and the number of processes involved. For example, detecting all races for launchpad-596064 took 2.5 s, or less than 0.5 ms per race. The average time to generate an execution branch for a race ranged from 0.66 s to 4.81 s. This time correlates roughly with the number of system calls. The average time to validate a race ranged from 0.44 s to 2.98 s. This time correlates roughly with the replay time.

In most cases, the average time to validate a race was somewhat larger than the time to replay the original execution by 0.3 s to 2 s. The time to validate a race is longer because, in addition to the time to replay the execution branch, it also includes the time to run the go-live execution, run the checker, and perform setup and cleanup work between races. Replaying an execution branch which ends at the anchor system calls is faster than replaying the whole original execution. However, during validation, the remainder of the recorded

execution now runs live, which is usually slower than replayed execution. In one case, launchpad-596064, validation was faster then original execution replay because nearly all of the execution branches resulted in replay divergence relatively early, eliminating the additional time it would take to replay the entire execution branches and have them go live.

The Generate and Validate times are averaged per race, so the total time to generate execution branches and validate races will grow with the number of races. However, races are independent of one another, so these operations can be easily done in parallel on multiple machines to speed them up significantly. Overall, the results show that RACEPRO can detect harmful process races not only automatically without human intervention, but efficiently.

3.8 Differences from SCRIBE

Our implementation of RACEPRO builds atop our deterministic SCRIBE record-replay engine introduced in the previous chapter. However, because SCRIBE only supports deterministic record-replay, enhancements had to be made to fulfill RACEPRO's requirements. These enhancements take place at different steps in RACEPRO's workflow. First, the recording step had to be modified to include additional deterministic information to permit RACEPRO to extract semantics allowing happens-before graph calculation and race detection. Second, RACEPRO's explorer must manipulates the recorded log file to finely control the ordering of system calls. For this, a library was built to manipulate the recording using a general purpose abstraction. Third, we added a barrier synchronization primitive in the replayer to allow RACEPRO to synchronize a group of processes.

SCRIBE makes significant efforts to record just enough information to perform deterministic replay as to minimize the recording overhead. SCRIBE focuses on capturing sources of nondeterminism and ignores deterministic data. Such recording is insufficient for RACEPRO. For example, when calculating the happens-before graph, or performing race analysis, RACEPRO needs access to system call semantics, unavailable from a SCRIBE recording. In fact, RACEPRO requires additional information in three areas. First, when the recorded application invokes a system call, SCRIBE does not record the system call number, nor its return value as these values are deterministic and thus provide no benefit for deterministic replay. RACEPRO records these values to enable system call semantics extraction. For example, when performing wait-wakeup race detection (§3.5.3), RACEPRO accesses theses recorded values to identify all `wait()` calls and their

returned PIDs. Second, when recording OS resource ordering information, SCRIBE only records the rendezvous point unique sequence number (§2.5). RACEPRO must identify each accessed resource to relate system calls accessing the same resource. For this, a resource type and a unique resource identifier are recorded along side to the sequence number of each rendezvous points. Complications arise with pipes and sockets as the kernel considers each endpoint to be a separate resource. Because RACEPRO needs to relate system calls accessing the same socket or pipe, extra metadata is recorded to uniquely identify the pipe or socket being accessed, regardless of the endpoint. Third, to handle signals, SCRIBE only records the value of signals to be delivered, to replay them verbatim while ignoring the originating cause of the signal (§2.6.1). To detect signal races, RACEPRO must tie the signal delivery to its corresponding source. For this, a per-signal identifier is recorded during signal emission, and recorded again during signal delivery. By matching these identifiers, RACEPRO can tie signal deliveries to their sources. These three areas of additional deterministic recorded information do not impede performance per our findings. If performance was impacted, RACEPRO could record a minimal log file containing only nondeterministic data akin to SCRIBE, and replay it once to recover the required deterministic data.

RACEPRO's explorer manipulates the recorded log file to finely control the ordering of system calls. To simplify the explorer implementation, we built a high level abstraction to parse and manipulate recorded log files. A log file can be seen as a list of serialized events. When processing a log file, loading all events in memory can be prohibitive due to the large amount of events. For this reason, we followed a streaming architecture to do our processing. The stream begins with a source, emitting each event from a given recorded log file. The event stream then goes through a series of functional blocks that may perform various actions. An event stream ends in a sink, which may write a new log to disk, or feed events directly to the replayer. The explorer instantiates and connects functional blocks to generate execution branches. Some of these blocks can add, remove, or modify certain events. Other blocks may do more complex tasks, for example consolidating all sequence numbers of resources due to the reordering of system calls. Another example is adding a barrier synchronization primitive at a consistent cut, to ensure that all processes reach a certain location in their event stream simultaneously.

This barrier synchronization primitive must be supported by the replayer. During the replay, when a barrier is encountered by a process in its event stream, the process waits for all other processes to also reach their corresponding barrier. The last process to reach its barrier resumes the execution of the other processes. This feature is useful in certain edge cases with wait-wakeup races, where multiple `exit()` calls sharing

no common resource must be synchronized. Barrier can also be useful in detecting TOCTOU races as the replayer can invoke a custom callback modifying the file system at a barrier right before resuming execution. Finally, barriers are used to ensure that all processes go-live simultaneously at a specific consistent cut. Aside from this additional barrier feature, the replayer has not gone through extensive changes compared to SCRIBE.

RACEPRO's record-replay engine differ from SCRIBE's mostly from its ability to record useful deterministic data, and modify recorded executions. RACEPRO and SCRIBE replayers mostly share the same feature set, and provide no tolerance to execution divergences, which is why RACEPRO must go-live as soon as the modified behavior has been executed.

To improve the quality of race triage, using a replayer tolerant to execution divergence may be useful to implement a powerful automatic checker. Instead of going live, the replayer would continue replaying past the introduced modifications and monitor the application behavior. In cases where the exhibited race is benign, a minimal divergence is expected, when the race is harmful, a substantial divergence is expected. Interestingly, RACEPRO could report the effect of a race by showing the difference from the original execution and the modified execution. We explore a record-replay engine tolerant to such divergences in the next chapter with DORA.

3.9 Related Work

Thread races. Enormous work has been devoted to detecting, diagnosing, avoiding, and repairing thread races (*e.g.*, [43; 74; 77; 97; 123; 128]). However, as discussed in §3.1, existing systems for detecting thread races do not directly address the challenges of detecting process races. For instance, existing static race detectors work with programs written in only one language [43; 74]; the dynamic ones detect races within only one process and often incur high overhead (*e.g.*, [73]). In addition, no previous detection algorithms as we know of explicitly detect wait-wakeup races, a common type of process races.

Nonetheless, many ideas in these systems apply to process races once RACEPRO models system call effects as *load* and *store* micro-operations. For instance, we may leverage the algorithm in AVIO [68] to detect atomicity violations involving multiple processes; the consequence-oriented method in ConSeq [130] to guide the detection of process races; and serializability or linearizability checking [46].

A recent system, 2ndStrike [49], detects races that violate complex access order constraints by tracking

the *typestate* of each shared object. For instance, after a thread calls `close(fd)`, 2ndStrike transits the file descriptor to a "closed" state; when another thread calls `read(fd)`, 2ndStrike flags an error because reads are allowed only on "open" file descriptors. RACEPRO may borrow this idea to model system calls with richer effects, but we have not found the need to do so for the bugs RACEPRO caught.

RACEPRO leverages the replay-classification idea [77] to distill harmful races from false or benign ones. The go-live mechanism in RACEPRO improves on existing work by turning a replayed execution into a real one, thus avoiding replay divergence when a race does occur and changes the execution to run code not recorded.

We anticipate that ideas in RACEPRO can help thread race detection, too. For instance, thread wait and wakeup operations may also pair up in different ways, such as a `sem_post` waking up multiple `sem_down` calls. Similarly, the go-live mechanism can enable other race classifiers to find "root races" instead of derived ones.

TOCTOU races. TOCTOU race detection [111; 112; 120] has been a hot topic in the security community. Similar to RACEPRO, these systems often perform OS-level detection because file accesses are sanitized by the kernel. However, TOCTOU races often refer to specific types of races that allow an attacker to access unauthorized files bypassing permission checks. In contrast, RACEPRO focuses on general process races and resources not only files. Nonetheless, RACEPRO can be used to detect TOCTOU races *in vivo*, which we leave for future work.

Checking deployed systems. Several tools can also check deployed systems. CrystalBall [125] detects and avoids errors in a deployed distributed system using an efficient global state collection and exploration technique. Porting CrystalBall to detect process races is difficult because it works only with programs written in a special language, and it does checking while the deployed system is running, relying on network delay to hide the checking overhead. in vivo testing [32] uses live program states, but it focuses on unit testing and lacks concurrency support.

To reduce the overhead on a deployed system, several systems decouple execution recording from dynamic analysis [30; 78]. RACEPRO leverages this approach to check process races. One difference is that RACEPRO uses OS-level record and replay, which has lower overhead than [30] and, unlike Speck [78], RACEPRO works with both multiprocess and multithreaded applications. In addition, a key mechanism required for validating races is that RACEPRO can faithfully replay an execution and make it go-live at any point, which neither previous system can do.

OS support for determinism and transaction. Our idea to pervasively detect process races is inspired by operating system transactions in TxOS [86] and pervasive determinism in Determinator [9] and dOS [16]. TxOS provides transaction support for heterogeneous OS resources, efficiently and consistently solving many concurrency problems at the OS-level. For instance, it can prevent file system TOCTOU attacks. However, as pointed out in [67], even with transaction support, execution order violations may still occur. Determinator advocates a new, radical programming model that converts all races, including thread and process races, into exceptions. A program conforming to this model runs deterministically in Determinator. dOS makes legacy multithreaded programs deterministic even in the presence of races on memory and other shared resources. None of these systems aim to detect process races.

3.10 Summary

We presented the first study of real process races, and the first system, RACEPRO, for effectively detecting process races beyond TOCTOU and signal races. Our study has shown that process races are numerous, elusive, and a real threat. To address this problem, RACEPRO automatically detects process races, checking deployed systems *in vivo* by recording live executions and then checking them later. It thus increases checking coverage beyond the configurations or executions covered by software vendors or beta testing sites. RACEPRO builds on SCRIBE, our transparent, low overhead record-replay engine described in the previous chapter, and extends it to allow system call reordering modifications at the end of a recorded log file. This ability goes beyond deterministic replay and is instrumental to RACEPRO's race validation feature. The race detection accuracy depends on the quality of the checker which runs modified executions. The checker would thus benefit from a replay engine tolerant to modifications in the execution. We introduce this concept and propose an implementation in the next chapter with DORA.

Chapter 4

DORA: Transparent Mutable Record-Replay

4.1 Introduction

As applications grow in complexity, software bugs have become increasingly common and more difficult to reproduce, diagnose, and fix. Aggressive release schedules exacerbate the problem, resulting in frail software that requires patches to fix problems that occur in the field. Resolving a bug typically starts with reproducing it in a controlled environment. Because the common approach of conveying a bug report is often inadequate for tricky, nondeterministic bugs, record-replay has been developed to capture application bugs as they occur and deterministically replay the bug at a later time, removing the burden of repeated testing to reproduce the bug.

Despite an abundance of research on using record-replay systems for debugging [18; 64; 52; 5; 77; 82; 96; 102; 106], these works have focused on bug reproducibility and have had limited or no support for diagnosing and fixing bugs. Debugging almost always requires modifying the program, whether by adding print statements, testing if a change fixes the problem, or some other method. However, most previous record-replay systems do not allow the recorded execution to be replayed with any modifications to the application. A handful of systems do allow some new code to be run in the middle of a replay, but they do not support changes to the application state [56; 30], which limits the utility of these systems for debugging and validating changes.

To address this problem, we introduce DORA, a mutable record-replay system which allows a recorded

execution of an application to be replayed with a modified version of the application. Mutable record-replay provides a number of benefits for reproducing, diagnosing, and fixing software bugs. For instance, mutable record-replay can replay a version of the recorded application that is recompiled with debugging information, reconfigured to produce verbose log output, or modified to include additional code instrumentation such as print statements. Further, in the previous chapter we introduced RACEPRO, a process races detection system, and noted that it would benefit from the ability to replay a modified recorded execution to improve the accuracy of its race checkers.

Mutable record-replay can also replay a recorded application execution of a production workload using a patched version of the application. This is useful for both application developers and system administrators. An application developer can use a recording of a bug when developing a fix. Replaying the recording on a modified application speeds up debugging and provides a novel way of validating bug fixes for nondeterministic bugs, which can otherwise be time consuming and difficult. For example, a developer who writes a patch can test it by taking a recorded execution of the exploit on the original application and replaying it using the patched application to quickly verify that the patch closes the vulnerability instead of painstakingly regenerating the exploit for each attempted fix of the problem.

System administrators often worry that applying patches will break their applications. Mutable replay allows administrators to independently test patches on production workloads. An administrator can record the unpatched application in production, apply the patch to an offline version of the application, and then replay the recorded execution using the patched application. If the replay succeeds, the administrator will be more confident that the changes will not introduce regressions.

Mutable replay complements traditional quality assurance testing. Quality assurance provides broad coverage but fails to handle many corner cases, which is why bugs arise in production in the first place. In contrast, mutable replay isolates actual bugs that occur in production. These bugs can be timing and configuration dependent, so some surface very rarely. This coverage is often not possible with traditional testing due to nondeterministic program behavior. Furthermore, mutable replay provides fast turnaround time, enabling a bug to be replayed quickly and directly tested against application changes that attempt to fix the problem. This not only speeds up debugging, but also provides a way to validate bug fixes for nondeterministic bugs, which can otherwise be much more time consuming and difficult.

We started with SCRIBE, our deterministic record-replay engine that aims to reenact exactly what has happened in the past. We evolved this engine with RACEPRO, to allow reordering of system calls at the

end of a recorded log file. This marks a departure from deterministic record-replay, but does not address divergence problematics. With DORA, we further evolves our record-replay engine to enable mutable replay capabilities, enabling compelling use cases.

DORA consists of three components: (1) a recorder that records application execution to a log, (2) a replayer that can replay a modified version of the application using the log, and (3) an explorer that uses the replayer to find the execution of the modified program that best corresponds to the log file. The recorder and replayer build upon our previously described record-replay engines.

The recorder operates primarily at the interface between applications and the operating system (OS) to transparently record an application's nondeterministic interactions. It avoids imposing unnecessary timing and ordering constraints that would hinder mutable replay with a modified application. To aid mutable replay, the recorder also logs deterministic interactions to detect and resolve any differences between the recorded application execution and the replay of a modified version of the application.

The replayer replays a previously recorded execution using a modified version of the application, matching events from the original log with the actions of the modified program. If the application used for replay is the same as the one recorded, the replayer provides deterministic replay of the unmodified application. However, if the replayed application's behavior diverges from the original's, the replayer gathers information for the explorer about the new code path the program was trying to execute and waits for instructions on how to proceed. Because it operates at the OS-level like the recorder, the replayer has access to sufficient OS semantics to understand why a replay diverges from the original execution and can leverage these semantics to help the explorer.

The explorer evaluates several possible execution paths to find a successful mutable replay. It performs a best-first search for an execution of the modified program that is as close to the original execution as possible according to some cost function d. It begins by replaying a recorded execution on a modified program. When the replay diverges from the original execution, the explorer tries to determine why. For example, suppose the modified program made an unexpected `printf()` call. This could be a new call to produce debugging information, or it could simply occur earlier than expected because code was deleted. The explorer chooses the most promising possibility and communicates its decision to the replayer. This process repeats until a successful execution is found.

DORA is designed to handle an wide range of real-world programs, including multi-threaded applications. It can support a broad range of useful application changes, but cannot support arbitrary changes;

major changes to the process layout or shared memory layout are not supported. Despite this limitation, DORA is useful in a wide range of real-world use cases for testing, debugging, and validating application changes. In fact, we even found a previously unknown bug in Apache using DORA [6]. DORA's usefulness in practice makes sense given that bug fixes tend to be relatively small and rarely change core application semantics [113; 57].

We have implemented a DORA Linux prototype that runs on commodity multicore hardware without changing, relinking, or recompiling applications or libraries. Our experimental results with over thirty different application changes show that DORA can (1) record unmodified real-world multi-threaded applications with less than 10% overhead on multicore hardware, (2) replay applications that have been reconfigured to produce verbose debugging output or modified with added debugging instrumentation, (3) replay real exploits on patched applications to verify that the patches close these vulnerabilities, and (4) replay benchmark workloads to validate application patches and version upgrades despite changes in thousands of lines of code.

We present the design, implementation, and evaluation of the DORA mutable replay system. §4.2 provides a definition of mutable replay. §4.3 describes the DORA recorder. §4.4 describes the DORA replayer. §4.5 describes the DORA explorer and presents an example illustrating the use of the system. §4.6 presents some key properties that can be guaranteed regarding DORA's mutable replay behavior. §4.7 discusses limitations of the current system. §4.8 presents experimental results. §4.9 discusses related work. Finally, §4.11 presents a summary and concluding remarks of this chapter.

4.2 Mutable Replay Concept

Since mutable replay is a previously undefined concept, we begin by presenting a definition. Let e be the recorded execution of some program P. Let E' be the set of possible executions of P', a modified version of P. A mutable replay of e on P' will then be an execution e' in E' such that the differences between e and e' are a result of the differences between P and P'.

The difference between two programs includes not only differences in their executables, but can also include changes in input and environment, such as environment variables, configuration files, the file system, and host-related information. Note that there is not always a clear mapping from input in the original program to input in the modified program. For example, the original program could read more bytes from

`stdin` than the modified program.

Since some executions in E' are intuitively preferable to others, we introduce the concept of a d-*optimal mutable replay*, an execution in E' that is optimal according to a cost function d. The cost function measures the difference between the original execution and the mutable replay. The value returned by d reflects the minimal cost of transforming the execution e_1 into a candidate execution e_2. A lower score is better, scores can be negative, and the score must be the lowest when e_1 is identical to e_2. A d-optimal mutable replay e_d of an execution e on P' satisfies $d(e, e_d) = min_{e' \in E'} d(e, e')$. There is always at least one d-optimal mutable replay for a given execution e and a program P'.

Finding the d-optimal mutable replay is undecidable in the general case. To show this, we first observe that finding the d-optimal mutable replay requires running P' because predicting the executions of a program is undecidable. Suppose P' has an added infinite loop at its beginning. Then, when running P', the replayer will loop infinitely since detecting an infinite loop is undecidable. Thus, finding the d-optimal replay is undecidable.

Even in the subset of cases in which finding a d-optimal mutable replay is decidable, it is still NP-hard with respect to the number of events in the log. Consider a program P' which only adds a `read()` system call of `n` bytes. There are $O(2^n)$ possible results of this call. In addition, arbitrary signals could be delivered between any two instructions. If the program is threaded, they are many possible thread interleavings. Since differences in signal delivery and thread interleaving could theoretically cause radically different behavior, and since determining the future execution of a program is undecidable, a mutable replayer must consider every possibility, which is infeasible. Thus, no mutable replay system can efficiently find a d-optimal mutable replay in all cases.

Fortunately, however, many useful changes to programs are modest in size and scope. In particular, bug fixes tend to be relatively small and rarely change core application semantics [113; 57]. The same is typically true of code instrumentation added to a program for debugging. Based on this observation, we designed DORA with a d function that has useful properties for testing and debugging. In this context, §4.8 shows that DORA is able to find d-optimal mutable replays in practice using real-world applications. Furthermore, §4.6 presents guarantees that can be made about the optimality of DORA's approximation algorithm.

A simple example may help further clarify the concept of mutable replay. Figure 4.1 shows a program on the left that prints the current time in seconds to `stdout`. On the right, it shows a modified version of

```
int main() {
  printf("d\n", time(NULL));
  return 0;
}
```

```
int main() {
  FILE *out = fopen("output", "w");
  fprintf(out, "d\n", time(NULL));
  return 0;
}
```

Figure 4.1: **Program Modification Example.** Original program (left), modified program (right).

the original program that instead writes the output to a file. Intuitively, we want the `gettimeofday()` call in the replay of the modified program to return the same time returned in the recorded execution of the original program. We will show that DORA does this, producing a d-optimal replay with the cost function described in §4.5.

4.3 Recorder

DORA's recorder builds upon RACEPRO (§3) The RACEPRO record-replay engine provides five key benefits for mutable replay. First, operating at the OS-level avoids tracking low-level hardware nondeterminism that is unnecessary for application replay and would significantly complicate mutable replay. Second, DORA records the execution of system calls in a manner that enables system calls and their effects on the kernel to be fully executed during replay. As discussed in §4.4, this is essential for mutable replay because there are times when DORA must transition processes from controlled replay to live execution to enable mutable replay. Third, DORA's recorder can record the execution of multiple processes and threads with low overhead on production systems. Fourth, DORA's recording is transparent to applications. It does not require changing, relinking, or recompiling applications or libraries, and it supports programs written in any programming language. Finally, DORA's recorder can record additional deterministic information useful to perform mutable replay, and allow the modification of a recorded execution log file.

The recorder operates on a group of tasks (threads and processes), which we refer to as a *session*. DORA records interactions between the session and its external environment, such as incoming network packets and nondeterministic interactions between tasks, in a manner which accommodates application changes during replay. DORA also records deterministic information to help detect changes in an application's execution path during replay.

```
// 83 initialization events
--- cut ---
munmap(0xb76e1000, 968e) = 0
rdtsc = 000057ed322904cf
time(NULL) = 0x4f9bd2e7
fstat(1, 0xbff8d684) = 0
mmap(0, 1000, 3, 34, -1, 0) = 0xb76ea000
write(1, 0xb76ea000, 11) = 11
exit(0) = 0
```

Figure 4.2: **Recorded Log File of Original Execution.** The first 83 events are omitted.

The recorder saves the recorded execution to a log file. Figure 4.2 shows the tail of the log file generated by running the simple program in Figure 4.1 on the left. We excluded 83 events related to program initialization, including `execve()` and C library bootstrapping events. The last two initialization events are shown. The `rdtsc` event corresponds to seeding a random generator from the C library by reading the time stamp counter of the CPU. The log also includes several system calls and information about their arguments.

In deterministic record-replay, all deterministic information need not be recorded because it will be regenerated during replay, but in mutable replay, the modified application may behave differently even in deterministic sections of code. DORA's recorder stores additional deterministic information in the log to help the replayer detect differences between the original and replayed executions as early as possible. Since DORA provides deterministic replay for unmodified applications, it does not need to record the additional deterministic information in production, but instead records such information afterwards by replaying the original execution and recording additional deterministic information as needed.

Similarly to RACEPRO that records additional deterministic information (§3.8) to extract execution semantics, DORA records all system calls executed, not just those involved in nondeterministic interactions. DORA records the system call number, and return value for each system call. Unlike RACEPRO, DORA also records all system call arguments. For arguments that are pointers, DORA follows the pointer chain to record the actual memory contents, which are used during replay to see if two system calls are equivalent. By recording memory contents instead of the pointer values, DORA is more tolerant of memory layout changes caused by application modifications.

DORA also records the virtual addresses of shared memory accesses, allowing the replayer to match

shared memory access to detect divergence. However, this mechanism means that changes to the memory layout of writable shared memory affecting page boundaries can cause DORA to incorrectly replay data races. Fortunately, a recent study of common security patches indicates that a vast majority of application patches [57] do not make such changes.

Finally, when performing mutable replay, the modified application may introduce system calls that are nondeterministic and environment dependent. DORA addresses this issue by recording two additional types of information during the original recorded execution. First, DORA stores additional information for mutable replay to ensure that nondeterministic actions that occur during the modified application replay but not during the original recorded execution are consistent with the recorded execution. For example, DORA periodically records timing information to ensure that any new calls to time-related functions are consistent both with each other and with any calls in the recorded execution. Second, DORA records other information about the execution environment so that new system calls in the modified application behave as they would have in the environment in which the program was recorded. For example, the recorder stores host information in case the modified application requests it with a new `uname()` or `gethostname()` call.

4.4 Replayer

DORA's replayer replays the originally recorded execution using either the original program or a modified program. It requires that the execution is replayed on a machine which supports all the recorded instructions. For example, a program that uses SSE instructions when it is recorded cannot be replayed on a machine without SSE instructions unless it is recompiled to use a different ISA. The replayer uses the recorded log file to generate a separate log file per task. Each task is replayed independently, but the replayer enforces the recorded order of access to shared resources.

A key aspect of the replayer is that it can transition a task or a group of tasks from controlled replay to normal execution. This feature is essential for mutable replay because a modified program may have new code to execute that is not part of the recorded execution. DORA can run such code at any time because it fully executes system calls and their effects on the kernel during replay. Many other replay systems only emulate the effects they have on userspace [96; 52], but this would prevent normal execution of the application from being enabled in the middle of replay.

As a task executes kernel code, the replayer compares the execution with what is expected in the log

file. When the execution *matches* expected events in the log, the replayer ensures it behaves as it did in the original execution. System calls match if they have the same system call numbers and arguments. When an argument is a pointer to a buffer, DORA compares the contents of the buffers instead of the pointer addresses. Shared memory access events match if the access types and page addresses are the same. The replay ends successfully if all tasks terminate after consuming every recorded event. A replay that uses an unmodified application will always end successfully in this manner.

However, if a replaying task is about to execute a code path that does not correspond to the expected event, the replay has *diverged* from the log. The replayer conveys this to the explorer, which determines how the replayer should resolve the divergence. If the explorer determines that the replayer should continue replaying the current log, the unexpected event can be treated in number of different ways to try to resolve the divergence so that later events will match events in the log. This ability to act on a divergence is an improvement compared to SCRIBE and RACEPRO where a divergence always resulted in a unrecoverable failure. For simplicity, DORA treats a divergence as one of two possible types of *mutations*, an *addition* or a *deletion*.

4.4.1 Additions

An addition is an event added to the program. For example, a program could be modified by adding code that includes a new system call. A new event is most often a system call, but it can also be a new signal or shared memory access. When executing an event not in the log file, DORA has three main responsibilities. First, it must decide when to execute the new event relative to events already in the log file. System calls, signals, and shared memory events are often racy with respect to other processes or threads, so there can be many possible orderings. Second, it must ensure that the semantics of the event are consistent with the semantics of the recorded execution. Finally, it is useful to be able to deterministically reproduce these decisions in subsequent replays, as explained in §4.5.

To handle these responsibilities, the replayer switches the process that encountered the addition from controlled replay into *direct mode*. Direct mode switches the respective process to normal execution and enables DORA's recorder to record the execution. This adds a new event to the log and orders it with respect to other events in the log. The resulting log can then be later deterministically replayed with the same modified program. Once the process completes the additional operation, it returns to regular replay mode; no other process has left regular replay mode. This capability previously not available in SCRIBE

and RACEPRO is instrumental in the way DORA performs mutable replay. We discuss how switching from controlled replay to direct mode is done in further detail for new system calls, signals, and shared memory accesses.

System calls. When encountering a new system call, DORA executes the call and records the execution as discussed in §4.3. This is possible because the replayer fully executes system calls and their effects instead of just emulating them, ensuring that it is possible to switch a process to normal execution at any time. DORA further instruments various system calls to ensure that the new system call behaves consistently with the recorded execution. The specifics of this depend upon the semantics of the added system call. We highlight system calls that deal with three important types of issues: environmental or timing information, resource allocation, and sockets.

For system calls that request environmental information or timing information, DORA ensures that the return values are made consistent with the information in the original log. This includes `gettimeofday()` and `gethostname()`. For example, if a new `gethostname()` call is executed, DORA already recorded such environmental information and ensures that the name reported is the same as what was already recorded.

For system calls that request new resources, DORA ensures that assigned resources do not conflict with those used by the replayed execution. For example, if a new page in memory is allocated, DORA ensures that its address will not conflict with those used in the original program. If the system call manipulates an existing shared resource, the respective resource serial numbers are renumbered to account for the new event.

DORA simply executes new socket-related system calls during replay except when dealing with data streams originating from outside the session. To deal with data streams, such as external sockets, DORA registers fake backends to the corresponding file descriptor. Socket system calls related to those data streams will simply manipulate the recorded network data. For example, if a `read()` on a network socket is changed to a `recvmsg()` on the same socket, DORA provides the appropriate data. If a new `read()` from a socket tries to access more data than recorded, 0 is returned to indicate end of file.

Signals. When encountering a new signal, such as from a modified application with a new `kill()` call, DORA needs to determine when to deliver the signal to the target process. The replayer does this much as the recorder does. Like the recorder, the replayer defers signal delivery until the target process encounters a sync point. This ensures that signal delivery can be replayed deterministically during subsequent replays.

Shared memory. When encountering a new shared memory access, DORA needs to determine how to interleave the access with other accesses. As in recording, a page fault occurs when a replayed process tries to access a shared page that it does not own, and the process must acquire ownership of the page. Once the process obtains ownership and completes the memory access, it releases ownership to the previous owner at the next sync point to ensure that the access order in the original execution is respected. The new memory events are added to the log file and the original serial numbers are reordered as necessary to ensure that subsequent replays based on this log deterministically perform the memory access in the same way.

4.4.2 Deletions

A deletion corresponds to the removal of events from the original log file and implies that the unexpected event matches a later event in the log. The replayer deletes the intermediate events. There can be several possible matches for an event if the event occurs several times later in the original log. DORA identifies possible matches and reports each possible match to the explorer. Because it is expensive to process many events and it becomes increasingly unlikely to find a match that will result in a successful mutable replay if an extremely large number of events need to be deleted, DORA imposes a cap on the number of events it can remove for a deletion. The cap is 10,000 events in our implementation. We present further detail regarding deletions that involve system calls, signals, and shared memory accesses.

System calls. Most system calls do not have any side effects in the log file, so not executing a call itself is all that is necessary to delete it. If the system call involves a shared resource, DORA also renumbers the serial numbers for the resource so that its serial number sequence does not contain any gaps. Deleting system calls related to external sockets does not remove the incoming data, since it is preserved as part of the stream of data associated with the resource. Data that is not consumed from a deleted socket system call will eventually be consumed by other remaining or new system calls.

When a deleted system call was originally associated with the delivery of asynchronous events, the events must be relocated to other sync points. For example, consider a program that has a `SIGALRM` delivery scheduled on a `getpid()` sync point. If the new execution no longer calls `getpid()`, DORA must choose a new sync point at which to deliver the `SIGALRM` signal. DORA postpones the delivery of asynchronous events until the next sync point the application encounters. Choosing the next sync point is better than choosing the previous one because releasing page ownership prematurely would introduce spurious

page faults in the application and could prevent DORA from respecting the original page access order. For example, suppose a thread writes to a page and then releases ownership of the page at the following sync point, which is on a system call. If the system call is removed and DORA moved the ownership release event to a previous sync point, it would occur before the access to the page. This access would then trigger a page fault and generate a new ownership acquisition event that may not respect the original ordering. Moving the release of ownership to the following sync point avoids this issue.

Signals. Deleting a signal involves deleting its source, which is typically a system call. System calls that deliver signals require additional consideration because their effects create multiple events in the log file. For example, removing a `kill()` system call must also remove the delivery of the corresponding signal. During recording, DORA associates delivered signals with their sources by using an incrementing global token used across the entire session. When a signal is about to be delivered, DORA waits for the source to be triggered or deleted, which respectively delivers the signal or omits the signal from being delivered.

Shared Memory. When deleting a shared memory access, the corresponding ownership acquisition event should not be executed. However, the previous owner still releases its page ownership so that its behavior is consistent with the original recorded execution. Asynchronous events associated with a deleted shared memory event are relocated to other sync points in the same manner as they are for system calls.

4.4.3 Going Live

In rare cases, the entire log file is consumed before the modified application terminates. This can occur when the original application crashes, but the patched application avoids crashing. The replayer allows the session to *go live* and entirely transition from controlled replay to live execution. This enables the user to validate the correctness of the patch. Since DORA faithfully replays kernel actions, the system is always in a state that allows it to transition to live execution. Linux namespaces [19] create a consistent environment for processes before and after they go live. Examples of this are demonstrated in §4.8.

4.5 Explorer

The explorer uses the replayer to search for a d-optimal replay. When the replay diverges, the explorer must determine how to proceed. If this was the first divergence, the explorer decides whether the replayer should

consider the mutation as an addition or a deletion. If there were previous divergences, the explorer might also tell the replayer to reconsider a previously detected divergence and explore a different path. In this case, the explorer provides the replayer with a new log file. Thus, the replayer needs no knowledge of the exploration algorithm.

The explorer treats the problem of finding the best mutable replay as a search through a tree T of possible candidate executions from a start node e, the execution of the original program, to one of many goal nodes, which represent executions in E'. This problem is different from most other search problems because (1) expanding a node can result in an infinite loop, (2) there can be virtually infinite goal nodes, and (3) the paths to the goal nodes are not known beforehand because determining the possible executions of a program in advance is undecidable.

Since an exact search is undecidable in the general case and NP-hard even when it is decidable, DORA performs an inexact search using a modified uniform-cost search. For simplicity, DORA only considers additions and deletions. It does not consider, for example, input fuzzing or trying all possible racy paths. The algorithm performs the following steps:

1. Initialize T to contain the root node e.
2. Pick an unexplored execution in the tree with the lowest cost according to the cost function d.
3. Attempt to replay this execution on P'.
   (a) If this replay succeeds, this execution is selected and the exploration concludes.
   (b) Otherwise, the replay diverges on an unexpected event, and new nodes are added to the graph. One node represents an addition and the others correspond to each possible deletion. Each node has an associated log file so that nondeterminism due to mutations is reproduced exactly across replays. Go to step 2.

A useful feature of the explorer is that the end result of a replay up to a given node is recorded. Since this recording can be deterministically replayed, a mutable replay is easily reproducible. Furthermore, it is easy to compare two logs to see the differences between two executions. For example, a developer can compare the log of the originally recorded execution with the log of a mutable replay to understand how application modifications affected the replay.

For simplicity, we have implemented the explorer algorithm by replaying a new execution from the beginning of the log upon divergence. In reality, there is no inherent reason for executions to be replayed from the beginning since each child node's log only differs from its parent node's log after the point of divergence. For example, a checkpoint could be taken just before divergence occurs. Nodes created because of this divergence could replay from the checkpoint instead of from the beginning of execution [59; 60; 81].

As mentioned in 4.2, application modifications include not only differences in their source code or executables, but also changes in input and environment, such as environment variables, configuration files, the file system, and host-related information. DORA does not handle any of these modifications differently as the explorer makes decisions solely based on application behavior changes, not based on the root cause of the change.

While any function satisfying the properties specified in §4.2 can be used for d, we present a simple function that has useful properties for debugging purposes. Since matches are desirable and additions and deletions are undesirable, each match has a cost of $-M$ and each addition or deletion has a cost of $+1$, where $M > 1$. We use a value of 3 for M in our prototype, but the process of selecting a mutable replay was relatively insensitive to the specific value. Using a negative cost for matches means that the explorer is unlikely to backtrack after making many contiguous matches. This also means that the uniform-cost search is not guaranteed to find the optimal replay even amongst the nodes it considers (additions and deletions). We made this decision because the number of nodes it would need to consider to guarantee correctness is exponential. Since the future execution of a program is unknown, even potential executions which seem very unpromising could theoretically match many later events in the log file and obtain a very good score. Thus, the search would effectively become a breadth-first search if d could only return non-negative numbers. Event logs may have billions of events long, so this would not be feasible.

Example. To make this process more clear, we return to the example introduced in §4.2, in which a program that prints the time to `stdout` is modified to write the time to a file. Figure 4.3 illustrates how the explorer replays the modified program in Figure 4.1 using the recorded execution shown in Figure 4.2 of the original program in Figure 4.1. The explorer starts by replaying the original log file on the modified program. The complete original log file contains 90 events, but Figure 4.3 omits initialization events, such as `execve()` and bootstrapping code from the C library.

```
   munmap(0xb76e1000, 968e) = 0
   rdtsc = 000057ed322904cf
+ brk(NULL) = 0x9870000
+ brk(0x9891000) = 0x9891000
+ open("output", 577, 438) = 500
   time(NULL) = 0x4f9bd2e7
+ fstat(500, 0xbff8d674) = 0
- fstat(1, 0xbff8d684) = 0
   mmap(0, 1000, 3, 34, -1, 0) = 0xb76ea000
+ write(500, 0xb76ea000, 11) = 11
- write(1, 0xb76ea000, 11) = 11
   exit(0) = 0
```

Figure 4.3: **Mutable Replay of Modified Program.**

The replayer matches the first 85 events successfully, which are all system call events, resulting in a cost of -255. At this point, the replayer encounters a `brk()` that does not match the `time()` call in the original log file and diverges. Upon divergence, DORA can treat `brk()` as an added system call or search the original log for a `brk()` call and delete intermediate events. Since no other call to `brk()` occurs in the log file, only the addition path is considered, resulting in a cost of -254 and an additional node in the tree of candidate executions.

Following this algorithm, replayer adds two more system calls: another `brk()` and `open()`, resulting in a cost of -252. A `time()` call is executed and successfully matches the expected call in the original log, ensuring that the time returned in the replayed execution is the same as the time in the original log. The match lowers the cost of the execution path to -255.

The changed program then executes an `fstat(500, ...)` in the replaying execution. Although the system call number is the same as the `fstat(1, ...)` in the log, the file descriptors passed as the first argument are different. This is treated as a mismatch. No subsequent matching `fstat()` calls are in the log, so this is treated as an another addition, increasing the cost of the execution path to -254.

Next, the application diverges on the call to `mmap()` since it does not match `fstat(1, ...)`. For the first time in this example, the divergence can lead to a deletion or addition since there is a matching `mmap()` in the original log. The explorer creates two nodes, and explores the unexplored node with the

lowest cost. In this case, these two nodes are the only unexplored nodes. The addition node costs -253 since there is a one point addition penalty. The deletion node costs -256 because there is a one point deletion penalty for removing one node and a three point bonus for matching `mmap()`. Therefore, the deletion node is selected. The addition and deletion of `fstat()` is effectively a *replacement* of the system call.

The modified program then runs a `write(500, ...)` in the replaying execution which is different from the `write(1, ...)` in the log. Although the function calls are the same, the file descriptors are different, so this is treated as a mismatch. Since no subsequent matching `write()` calls are in the file, this must be treated as an addition, increasing the cost of the execution path to -255.

Finally, the program calls `exit()`, which diverges from the `write(1, ...)` in the original log. The divergence can lead to an addition of `exit_group()` or a deletion of `write(1, ...)`, matching the `exit_group()` in the original log. The addition node costs -254, the deletion node costs -257, and the unexplored node which added `mmap()` costs -253. The deletion node is selected.

Since the end of the log file has been reached, the explorer has found a successful replay with a cost of -257 and terminates. In this case, the explorer has successfully found a d-optimal replay. Although the explorer cannot prove this, the optimality of this replay is evident given the nature of the application modification.

From this simple example, we can observe that small code changes may significantly impact the behavior interactions of the application with the kernel API. The `fopen()` library call internally calls `malloc()`, resulting in two new invocations to `brk()`. Thus, even minor changes to high-level source code can result in relatively large changes to the low-level executable code.

4.6 Properties

We can make several useful guarantees about the behavior of DORA for certain classes of application changes.

Property 1 DORA *deterministically replays the original execution if the program is unmodified.*

In other words, DORA performs traditional deterministic record-replay when the program is unchanged. This property also implies that DORA provides d-optimal mutable replay for all d for unmodified programs.

Property 2 *If all explored mutations are* safe*,* DORA *deterministically replays all events in the original execution with the modified program.*

For our d, a safe mutation is an addition that does not change any state which is read by the original execution. These additions may store state which is later read, but the original execution must not access this new data. This guarantee is quite useful for debugging because it implies that all behavior in the original program, including race conditions, will be preserved deterministically.

For example, a `printf()` changes the internal state of the program by modifying an internal buffer, but returns the program to its original state, assuming proper newlines. Therefore, adding a `printf()` to debug a race condition always preserves recorded races because it will not change the relative ordering of events in the log. DORA can also handle the creation of new files. Any new calls to `open()` will receive a file descriptor not used by the original execution, so DORA guarantees that the original behavior of the program will be unaffected.

As another example, consider memory changes, for both shared and private regions of memory. First, reading the value of a variable in memory is safe. This is true even if the read is from shared memory and triggers a page fault leading to a temporary ownership transition. Additionally, the program can allocate and write to pages that are unused by the original execution. To avoid conflicts, DORA always assigns new memory allocations to a reserved area that is isolated from the original memory mappings.

Property 3 DORA *does not guarantee deterministic replay when given a modified application with arbitrary modifications.*

As explained in Section 4.4, DORA does not evaluate all possible interleaving of additions. For example, when a `printf()` is added in two different threads without locking, the order in which the calls are executed is variable. DORA picks the first possibility it encounters during replay and enforces this ordering for subsequent replays. However, this ordering is not enforced across separate invocations of the explorer.

Property 4 DORA *can deterministically replay a mutable replay of a modified application.*

DORA's explorer outputs the replay it selects to a log file. Thus, DORA can deterministically replay the original execution of the explorer using the modified program. This enables exact reproduction of a previously found replay, allowing DORA to be used iteratively.

4.7 Limitations

DORA has several limitations as it has no knowledge of application semantics. As a result, it only supports

replay across application changes that do not alter core application or execution semantics.

For example, an exploit might add a new entry to a MySQL database. This entry would be assigned a particular id by MySQL, and would affect the ids of all later entries. A patch that removes the exploit would also remove the created entry, resulting in a mismatch between the ids assigned during replay and the recorded assignments. DORA would not find a d-optimal mutable replay because the core semantics of the execution have changed.

Additionally, DORA currently does not effectively support major changes in the layout of shared memory. If objects are relocated from a page to another, DORA cannot always preserve the original access ordering since DORA manages shared memory at the page level. DORA could be modified to track objects instead of pages by instrumenting the application. Even without this functionality, however, DORA is able to handle some changes to MySQL, which heavily uses shared memory.

Finally, DORA currently does not support process/thread layout changes. For example, if an application was originally recorded with 10 threads running, and is now reconfigured to run with 5 threads as part of the application modification, DORA does not provide a way to find a good mutable replay. Similarly, DORA has difficulty with applications that use green threads as small code changes may result in radically different schedules.

Thus, there are some types of changes for which DORA will not find the d-optimal replay. Fortunately, DORA produces enough information for the user to identify when these conditions occur. This allows the user to distinguish behavior caused by an application change from behavior due to DORA's limitations and makes DORA a useful tool for debugging and validation.

These conditions may seem restrictive, but they are often not an issue in practice because patches rarely change core application semantics. In a study of 60 patches each for MySQL, Apache, OpenSSL, and Squid, 83% resulted in only minor changes to the application behavior. Analysis of various security patches showed that over 75% only changed applications in minor ways [57]. This makes sense, as patches often attempt to fix an edge case in an application.

4.8 Evaluation

We have implemented a prototype of DORA in Linux. The recorder and replayer run in kernel space while the explorer runs in user space. Although the prototype only instruments a subset of the Linux kernel

Name	Description
apache-log	Apache 2.4.2 web server
apache-sec	Apache 2.2.19 web server
exim	Exim 4.69 mail server
mysql	MySQL 5.0.67 database server
nginx	Nginx 0.8.14 web server
proftpd	ProFTPD 1.3.0 ftp server
redis	Redis 2.4.11 key-value store
squid	Squid 3.1.7 http proxy server
wget	wget 1.11.4 http client

Table 4.1: **Application Descriptions.**

API, we demonstrated the functionality of this prototype in diagnosing and fixing bugs and measured its performance overhead with nine widely used real-world, multi-process, and multi-threaded applications and 32 different application changes involving thousands of lines of code. For our experiments, we used version 2.6.35 of the Linux kernel, Python 2.6.6, Cython 0.14, and UnionFs-Fuse 0.23. Measurements were done on a set of HP DL360 G3 servers, each with dual 3.06 GHz Intel Xeon CPUs, 4 GB RAM, and dual 18 GB local disks.

We recorded a wide range of applications as listed on Table 4.1. We ran these applications with various workloads that exhibited bugs, as listed in the second column of Table 4.2. We verified that DORA can deterministically replay the original recorded applications and then replayed the executions with modified applications. §4.8.1 shows that DORA can replay these workloads using reconfigured or modified applications with additional debugging or other instrumentation. §4.8.2 shows how DORA can replay the exploits in Table 4.2 using patched versions of the applications to help developers verify that the bug patches successfully resolve the problems. It also shows how DORA can replay the workloads listed in the second column of Table 4.2 using the patched versions of the applications to help system administrators verify that the patches do not introduce errors in production workloads. §4.8.3 shows that DORA can verify production workloads on a series of release upgrades for the applications listed in Table 4.6. Finally, §4.8.4 presents record-replay overhead for the production workloads.

Name	Problem/Exploit Workload	Production Workload
apache-log	Log format change (Apache Bug 53131)	`httperf 0.8` with 100KB web page
apache-sec	DoS attack (CVE 2011-3192)	`httperf 0.8` with 100KB web page
exim	Privilege escalation (CVE 2010-4344)	Send 1000 1KB e-mail messages
mysql	Unauthorized access (CVE 2008-2079)	`sql-bench`
nginx	Crash server (CVE 2009-2629)	`httperf 0.8` with 100KB web page
proftpd	Crash server (CVE 2006-5815)	100 clients fetch 10MB file
redis	Request with insufficient logging	`redis-benchmark` with 50 clients
squid	DoS attack (CVE 2010-3072)	`ab 2.3` with cached facebook.com
wget	Create arbitrary file (CVE 2010-2252)	100 requests to http://www.cnn.com

Table 4.2: **Application Workloads.**

4.8.1 Debugging and Diagnosis Techniques

We used a wide range of debugging and diagnosis techniques with the workloads listed in the second column of Table 4.2. Since our work focuses on diagnosing and fixing bugs, most of the problems involve known security vulnerabilities, as indicated by their Common Vulnerabilities and Exposures (CVE) identifiers. However, the apache-log scenario shows a previously unknown bug in Apache that we found with DORA, and the Redis scenario shows how to add retroactive logging without discussing a specific bug.

For each of these exploits, we show how DORA can be used to diagnose the cause of a bug. We consider debugging techniques an experienced developer might apply to identify the root cause of each problem. Table 4.3 lists the application changes needed to use various debugging and diagnosis techniques for each scenario. DORA successfully found the d-optimal replay for all of these application changes. Table 4.4 shows the needed replay mutations.

apache-log was originally intended to show how DORA could be used for retroactive logging, but ended up showing DORA finding a previously unknown Apache bug [6]. We wanted to use DORA to add user agent and referrer information to Apache log files, since these could provide useful usage statistics for a website administrator. Although the default Apache logging configuration will not log this information, DORA records all HTTP header information that the server receives. Thus, an administrator using DORA could modify a configuration file to include this information and replay the recorded execution with the

Name	Debugging Change
apache-log	Modify log format in configuration file
apache-sec	Add print statements for debugging
exim	Recompile with debugging options enabled
mysql	Add conditional print statements for debugging
nginx	Change the config file to enable debug messages
proftpd	Recompile with debugging options enabled
redis	Log erroneous client requests
squid	Save parsed requests to file
wget	Change language from Italian to Japanese

Table 4.3: **Debugging Scenarios.**

modified configuration to generate the desired web server log.

However, doing this yields a log file with incorrectly truncated entries. This behavior was due to a previously unknown bug in Apache. In several places in code, Apache mistakenly assumes that a call to `write()` will either write the desired amount of bytes or fail, instead of checking the return value and calling `write()` until all the required bytes are written. We submitted a bug report and patch to Apache [6] which was accepted into the codebase.

apache-sec records an exploit of a heap overflow vulnerability that launches a denial of service attack against an Apache web server using only a handful of requests. By examining Apache's log, an experienced Apache developer will notice oddities in some of the requests, but will not have enough information to identify the bug with the default logging settings. In particular, it would be helpful to have more information about the range headers of the requests. Using DORA, a developer can add print statements to Apache, replay, and recognize that the problem was due to incorrect handling of overlapping range headers. Five system calls were added for each request as a result of the additional print statements.

exim involves an exploit that crashes the mail server using a heap overflow vulnerability in a buggy string formatting function that allows attackers to execute arbitrary code. If this crash was recorded in the wild, it would be helpful to use GDB to analyze the program at the time of the crash. However, production servers are almost always optimized and compiled without debugging symbols. Thus, a traditional record-replay

Name	Replay Mutations
apache-log	Add 1 `fstat()`, 1 `mmap()`, and 3 `write()` per request
apache-sec	Add 1 `fstat()`, 1 `mmap()` and then 2 `write()` per request
exim	None
mysql	Add 1 `fstat()`, 1 `mmap()`, 1 memory event, 12 `write()`
nginx	Add 508 `write()`, delete 1064 syscalls
proftpd	Add 1 `close()`
redis	Add 1 `open()`, at least 3 `write()` and 1 `close()` per request
squid	Add 1 `open()`, at least 10 `write()` and 1 `close()` per request
wget	Replace 17 `write()`, 24 `mmap()`, 2 `open()`, and delete 5 syscalls

Table 4.4: **Application Modifications for Debugging.**

system would be unable to help. Using DORA, a developer can recompile the program, replay the exploit using the recompiled program, and hook GDB to the replayed program before it crashes. When investigating the stack trace, the nature of this attack becomes clear. No mutations were needed in the d-optimal replay, despite various memory layout changes to the program as a result of recompilation.

mysql involves an exploit which maliciously uses symlinks to elevate permissions to a database. By default, MySQL disables logging. Thus, a developer trying to discover how a malicious user gained access to a database will have no information about which commands were executed. Using DORA, the developer can modify the program to log executed commands, then replay the exploit using the modified program. This process can be repeated, allowing the developer to iteratively add print statements to different sections of the code and pinpoint the bug. To demonstrate this, we added enough print statements to identify the bug. DORA found the d-optimal replay, which had mutations of fourteen added system calls and a shared memory event.

nginx involves running Nginx, a high-performance HTTP server, and crashing one of its worker processes with a malicious HTTP request that uses a buffer underflow attack to execute arbitrary code. The default log does not show what actions were taken on each request, which makes debugging difficult. Using DORA, the developer can modify the configuration file to enable verbose logging and replay the exploit with the modified configuration. To generate verbose logs on a workload exhibiting the exploit, 508 `write()`

calls were added and 1064 system calls were deleted.

proftpd records an exploit that crashes the FTP server by taking advantage of an off-by-one error to execute arbitrary code. As with the exim use case, GDB would be a helpful debugging tool, but a production server is unlikely to be compiled with debugging symbols. With DORA, a developer can recompile with the Makefile configuration for debugging, replay the exploit using the recompiled program, and hook GDB to the replayed program before it crashes. The resulting stack trace makes it easy to diagnose the problem. A replay mutation of adding 1 `close()` was needed to use the debugging configuration.

redis involves recording Redis, an in-memory key-value store often used in production applications as a caching layer on top of a general purpose database. This use case does not involve a specific bug but instead shows how a developer can add logging to Redis and replay this modification on the original recording, effectively turning on retroactive logging. Replay mutations of adding at least five system calls per request was needed. The number of additions varied based on the nature of the request. For instance, a malformed request triggered more logging than a proper request.

squid involves an exploit that crashes the Squid daemon by sending an empty Expect HTTP header parameter. The default request logging does not provide enough information about the header to determine the cause of the bug. Using DORA, a developer can modify the program to log each request to a file with the complete header information of the request, then replay a recording of the exploit with the modified program. This allows the developer to see that the requests which crash the server have empty header parameters and narrow down the bug to a very specific section of code. At least 12 system calls were added per request. The exact number varied depending on the type of request.

wget involves downloading a file from a malicious server that does a 301 redirect. A vulnerability in wget allows the server to choose the destination filename. Remote servers can create or overwrite arbitrary files and even execute arbitrary code by writing dotfile in a home directory. Because this behavior depends on a live server behaving in a particular way, this issue may be difficult to reproduce and debug if it is not noticed immediately. Additionally, to demonstrate the robustness of DORA, we suppose that an Italian developer observes this behavior and wants to show it to a Japanese developer who cannot reproduce the results because the server is no longer available. Using DORA, the second developer can replay wget in a different language, enabling collaborative debugging across international borders and language barriers. While this scenario is tongue-in-cheek, the translation use case is novel and has interesting applications. The replay involved replacing 43 system calls and deleting 5 system calls.

Name	Patch LOC +/-	Replay Mutations
apache-log	39+, 39-	Add 1 `write()` per truncated log entry
apache-sec	292+, 154-	Add 1 `write()` and replace 1 `writev()` per request
exim	7+, 0-	Delete 18 syscalls, add 29 syscalls
mysql	170+, 60-	Add 29 `lstat()`, delete 79 syscalls, add 1 `write()`, delete 15 and add 8 memory events
nginx	9+, 5-	Delete 1 `write()` and replace 1 `writev()` per request, then go live on crash
proftpd	17+, 3-	Delete 694 syscalls, add 38 syscalls, delete a `SIGSEGV`
squid	38+, 33-	Delete a `SIGSEGV`, 1 `close()`, 1 `stat()`, 1 `write()`, then go live
wget	43+, 12-	Replace 9 syscalls among `stat()`, `write()`, `open()`, `utime()`

Table 4.5: **Application Patches Tested Against Exploits.**

4.8.2 Patch Validation

For each bug exhibited by the workloads listed in the second column of Table 4.2, we replayed the bug-inducing workload on the patched application to verify that the patch successfully fixed the bug. Table 4.5 lists the number of lines of code added and deleted for each patch and the mutations needed for each replay. Redis is not included; since it did not involve an application bug, no patch was necessary. Table 4.5 shows three interesting points.

First, DORA found the d-optimal replay even with substantial patches of over 400 lines of code changed. The replay mutations that were needed to find the d-optimal replay varied. Many involved executing different system calls, but others involved changes in signal delivery and shared memory accesses. This demonstrates DORA's ability to replay despite a broad range of application modifications so long as the core application semantics remain the same.

Second, we show that production workloads can be replayed using patched applications to verify that the patch does not introduce errors into those workloads. For each workload listed in the second column of Table 4.2, DORA recorded the workload using the unpatched application, then found a d-optimal mutable

Name	apache-upgrade	redis-upgrade
Start Version	Apache 2.2.19	Redis 2.4.1
Upgrades	3 (2.2.20 - 2.2.22)	12 (2.4.2 - 2.4.13)
Commits	277	137
LOC	5179+, 388-	2942+, 1154-
Workload	`httperf 0.8`	`redis-benchmark`

Table 4.6: **Application Upgrades**

replay using patched versions of each application. We examined the output of each mutable replay and verified that the patches did not change application behavior when running the workload. We also compared each original recorded log with the log of the corresponding mutable replay to verify that the patches did not change the application execution in unexpected ways. System administrators could use this technique to test patches before deploying them to have more confidence that they will not break their production systems.

Third, Table 4.5 shows that the go live feature of the replayer can be used to validate patches even when a recorded exploit crashes a process. The exploit for Nginx caused a worker to crash, and the Squid exploit crashed the entire application. In both cases, DORA does not replay the original `SIGSEGV` and allows the applications to go live and handle new requests. Although a worker process crashed in the proftpd scenario, DORA did not go live because the proftpd master forks a new worker per connection and is resilient to worker crashes, allowing subsequent requests to be replayed without going live.

4.8.3 Release Upgrades

To demonstrate another use case of DORA, we took two server applications and recorded them running the benchmarks we used as production workloads as listed in Table 4.6. We then replayed those executions over a series of 15 release upgrades to verify that the workloads continued to function correctly across upgrades. DORA found the d-optimal replay in all of these cases.

apache-upgrade consists of a series of upgrades of Apache over an 8 month timeframe from 2.2.19 (May 21, 2011) to 2.2.22 (Jan 30, 2012). The upgrades involved changes of more than 5000 lines of code and 277 separate commits. Using DORA, we recorded version 2.2.19 running httperf, then replayed the recording with each subsequent version. We repeated this process for versions 2.2.20 and 2.2.21. DORA

Name	Recording Overhead	Storage Growth	Replay Speedup
apache-log	9.3%	31 KB/s	3.8x
apache-sec	4.8%	18 KB/s	1.9x
exim	4.3%	30 KB/s	7.2x
mysql	4.7%	9.6 KB/s	1.1x
nginx	9.7%	15 KB/s	2.2x
redis	2.6%	91 KB/s	1.3x
proftpd	4.1%	22 KB/s	2.6x
squid	8.2%	124 KB/s	1.2x
wget	2.2%	19 KB/s	11x

Table 4.7: **Mutable Replay Performance**

successfully replayed all of these application changes. This required various add and delete mutations of `read()` and `brk()` calls. Note that we also tried this experiment starting with Apache 2.2.18, but DORA was unable to replay from that version due to core library modifications that caused large shared memory layout changes between Apache 2.2.18 and 2.2.19.

redis-upgrade consists of a series of upgrades of Redis over a 7 month timeframe from 2.4.1 (October 17, 2011) to 2.4.13 (May 2, 2012). The upgrades involved changes of more than 4000 lines of code in 137 separate commits. Using DORA, we recorded version 2.4.1 running redis-benchmark, then replayed the recording with each later version. We also repeated this experiment for version 2.4.2 and upgrades 2.4.3 to 2.4.13, version 2.4.3 and upgrades 2.4.13 to 2.4.4, and so on. DORA successfully replayed all of these application changes. They required add and delete mutations of 12 different system calls, including `open()`, `close()`, `read()`, `write()`, `mmap()`, `munmap()` and `time()` system calls.

4.8.4 Performance

To quantify the performance costs of using DORA, we measured the runtime overhead of recording and replaying the production workloads listed in the second column of Table 4.2. Table 4.7 shows the overhead of recording the production workload with the unpatched application, the storage growth rate of recording,

and the speedup when replaying the recording with the patched application. Unless otherwise noted, default configuration options were used for all applications. The standard deviations for all measurements were negligible.

The recording overhead in all cases was less than 10% even for CPU-bound workloads designed to stress application performance. For example, Squid performance was measured with a fully cached web page, resulting in a CPU intensive workload. Even with these unfavorable workloads, the results indicate that DORA can be used in production systems with modest overhead.

Similarly, the time to replay the original recording on the original application was in all cases faster than the original execution; in one case, it was over an order of magnitude faster. This is because DORA can bypass blocking system calls that sleep. Since production servers are likely to sleep more and service requests less frequently than in our benchmarks, replay speedup will be much higher in practice.

While recording, the log was streamed through `gzip` before being persisted to disk. The storage growth rates ranged from 10 KB/s to 130 KB/s. These storage requirements are modest considering our workloads. DORA would take almost three months to fill a 1 TB drive at the worst of these rates, which makes it an affordable and practical solution.

4.9 Related Work

Many record-replay approaches have been proposed to improve bug reproducibility debugging [18; 30; 64; 52; 5; 77; 82; 96; 102; 106], but none allows for mutable replay. Some approaches propose relaxing the requirement of deterministic replay for performance reasons. For example, ODR [5] proposes only ensuring that the output is deterministically replayed for replay debugging. This is quite different from mutable replay, in which the output may change due to application changes.

Some record-replay systems can support a form of replay that may differ in limited ways from the original recorded execution. Crosscut [31] can reduce the information recorded in a log so that, for example, sensitive information can be purged before replay. Our previous work on RACEPRO detects process races due to dependencies in the ordering of system calls by recording an application execution to a log, identifying a pair of system calls that may be racy, truncating the log at the occurrence of the pair of system calls, inverting their order, and then replaying the truncated log with the reordered system calls to detect process races. However, RACEPRO only supports changes that reorder system calls and does not support changes in

the middle of replay. None of these approaches supports mutable replay, but mutable replay could be useful for some of these systems. For example, RACEPRO could use mutable replay to avoid replay divergence and more effectively detect process races. Another race detection tool [77] uses the iDNA [18] record-replay framework and would also benefit from mutable replay.

A few record-replay systems allow new code to be run while replaying a recorded execution [56; 30]. However, this new code cannot have any side effects on the program. If a replay diverges due to new code, these systems must rollback to a point prior to the divergence for the replay to continue. In contrast, DORA allows replay to continue even after divergence; side effects due to new code are preserved. Moreover, unlike DORA, these other approaches prevent application developers from leveraging existing configurable application functionality and instead require that developers learn a new complex system. Because these other approaches work at a VM level, they are fundamentally limited in their abilities to perform mutable replay and support the kind of application changes supported by DORA. Finally, none of these other approaches work on multicore or multiprocessor systems.

A concept of mutable replay was mentioned as a part of DSF [107], a Java-only framework for implementing distributed algorithms. DSF recognized that existing replay approaches did not allow adding print statements for debugging. DSF requires that all applications to be written using its framework, requires modification to the applications, and is primarily simulation-based. Furthermore, DSF presents no algorithms or mechanisms for actually doing mutable replay, and presents no experimental results demonstrating the ability to do mutable replay. More recently, a study has assessed the potential utility of mutable replay on real patches [57], though no mutable replay system or results are presented. DORA presents the first system that achieves transparent mutable replay, requires no application modifications, and demonstrates experimentally that mutable replay can be used with real applications.

Alternative techniques have been proposed to help with patch validation, one use case of mutable replay. Band-aid patching [98] and delta execution [113] instrument patches to identify portions of an application that have changed, execute both the unpatched and patched code paths either serially or in parallel, and select the results from one path or merge the results from both paths. However, these approaches incur substantial performance overhead. Many simple patches cannot be handled by these approaches, such as simple changes to data structures. Unlike DORA, these approaches do not allow patch validation on a recorded bug or offline patch validation on a production workload. Furthermore, they are designed only for patch validation and are not effective for debugging.

Self-healing systems have been proposed which record the occurrence of a bug, then automatically generate and apply a patch as a temporary fix to the problem [99]. DORA is complementary to these systems and can be used to verify that a generated patch successfully fixes problems that occurred in the original workload.

Finding a mutable replay has some similarities to the edit distance and longest common subsequence problems, which have applications to approximate string matching and bioinformatics. In those problems however, both sequences being used for matching are known in advance. In contrast, mutable replay must match a known execution log with an execution sequence that is not known in advance, so these algorithms cannot be directly applied.

4.10 Comparison with SCRIBE and RACEPRO

DORA builds atop RACEPRO's record-replay engine and shares some similarities and differences with its recorder, explorer, and replayer. RACEPRO evolves SCRIBE's recorder to include deterministic information in the recorded log file such as system call numbers and return values, unique resource identifiers along with sequence numbers in rendezvous points, and signal identifiers to tie signal deliveries to their sources (§3.8). DORA goes further in recording additional deterministic information including system calls arguments and their associated buffers. This allows DORA to detect a divergence early. Further, DORA records additional timing and environmental related information allowing new system calls in the modified application to behave as they would have in the environment in which the program was recorded (§4.3).

To perform additions and deletions of events in recorded log files, DORA reuses the high level abstraction introduced in RACEPRO for generating execution branches(§3.8). The implementation effort of the explorer is reduced, the memory requirements are lowered, and performance is improved due to the ability to feed the modified event stream directly to the replayer. Further, when deleting events, resource sequence numbers must be consolidated, which is a feature RACEPRO needs and is reused with DORA.

DORA's replayer has been improved from RACEPRO's in its ability to act upon divergence, instead of failing the replay or going live immediately. DORA's replayer has the ability to switch a process thread from controlled replay mode into record mode (i.e. direct mode) as discussed in §4.4.1. The ability to allow threads to be replayed while other are being recorded simultaneously is a significant departure from the replayer used in SCRIBE and RACEPRO.

When comparing DORA and SCRIBE, the two engines perform deterministic record-replay when the replayed application is identical to the original application. However, when the replayed application has been modified SCRIBE fails to replay due to divergence, while DORA performs mutable replay. DORA's feature set is a superset of SCRIBE's feature set.

4.11 Summary

DORA introduces the concept of mutable record-replay and is the first transparent mutable record-replay system. It enables, for the first time, a recording of an application execution to be replayed using a modified version of the application for a large class of application changes. Mutable record-replay is a superset of deterministic record-replay. DORA introduces an explorer that directs the replay mechanism to identify a mutable replay of the modified application that minimizes differences with the original unmodified application execution. We implemented a DORA prototype based on our previously introduced RACEPRO engine. Our experimental results demonstrate that mutable replay is feasible across a wide range of real-world applications and application changes which can reach thousands of lines of code, even without support for major changes to core application semantics. We show that mutable replay is useful for enabling common debugging techniques not possible with previous record-replay systems. Mutable replay can also be useful for a vast number of use cases, including race detection systems such as RACEPRO. While we explored mutable replay on applications implemented on a POSIX interface, our DORA prototype cannot be directly applied to all types of applications, in particular distributed applications. In the next chapter, we introduce SYNAPSE, a record-replay engine for distributed database systems.

Chapter 5

SYNAPSE: Distributed Record-Replay for Database Systems

5.1 Introduction

So far, we have explored the usefulness of execution record-replay mechanisms with the introduction the SCRIBE, RACEPRO, and DORA systems. These systems prime use case is to improve software robustness by facilitating debugging and detecting previously unknown bugs. Systems that provides fault-tolerance are typically built on top of record-replay mechanisms. Resilient distributed databases replicate data on different machines. Aside from fault-tolerance, running replicas provide other benefits such as horizontal scalability for load balancing. Databases (DBs) that implement data replication protocols often rely on an underlying record-replay mechanism. However, using deterministic record-replay is not desirable as these replicas would not be able to serve any traffic, as any new incoming queries would be a divergence from the primary execution. Data replication can be seen as replaying a recorded execution from a primary DB on a replica DB. For example, many distributed DBs such as MongoDB, RethinkDB or Elasticsearch rely on log based replication mechanisms, which is a form of record-replay. These distributed DBs are mostly used to power Web applications.

Web applications are increasingly built using a service-oriented architecture that integrates composable services using a variety of DBs. For example, graph-oriented DBs, such as Neo4j and Titan, optimize for traversal of graph data and are often used to implement recommendation systems [21]; search-oriented DBs, such as Elasticsearch and Solr, offer great performance for textual searches; and column-oriented DBs, such

as Cassandra and HBase,

Often, the same data, needed by multiple services, must be replicated across different DBs and kept in sync. Unfortunately, the data replication engines of these DBs are vendor specific and not compatible with each other. To complicate matters further, these replication engines offer very different consistency semantics which makes web development difficult. It is thus desirable to provide a generic data replication system compatible with most databases and offer an easy to understand consistency model in the context of Web applications.

We present SYNAPSE, the first easy-to-use and scalable cross-DB replication system for simplifying the development and evolution of data-driven Web applications. With SYNAPSE, different services that operate on the same data but demand different structures can be developed independently and with their own DBs. These DBs may differ in schema, indexes, layouts, and engines, but each can seamlessly integrate subsets of their data from the others. SYNAPSE transparently synchronizes these data subsets in real-time with little to no programmer effort. To use SYNAPSE, developers generally need only specify declaratively what data to share with or incorporate from other services in a simple publish/subscribe model. The data is then delivered to the DBs in real-time, at scale, and with delivery semantic guarantees.

SYNAPSE makes this possible by leveraging the same abstractions that Web programmers already use in widely-used Model-View-Controller (MVC) Web frameworks, such as Ruby-on-Rails, Python Django, or PHP Symfony. Using an MVC paradigm, programmers logically separate an application into *models*, which describe the data persisted and manipulated, and *controllers*, which are units of work that implement business logic and act on the models. Developers specify what data to share among services within model declarations through SYNAPSE's intuitive API. Models are expressed in terms of high-level objects that are defined and automatically mapped to a DB via *Object/Relational Mappers* (ORMs) [15]. Although different DBs may need different ORMs, most ORMs expose a common high-level object API to developers that includes create, read, update, and delete operations. SYNAPSE leverages this common object API and lets ORMs do the heavy lifting to provide a cross-DB translation layer among Web services.

We have built SYNAPSE on Ruby-on-Rails and released it as open source software on GitHub [116]. We demonstrate three key benefits. First, SYNAPSE supports the needs of modern Web applications by enabling them to use many combinations of *heterogeneous DBs*, both SQL and NoSQL. Table 5.1 shows the DBs we support, many of which are very popular. We show that adding support for new DBs incurs limited effort. The translation between different DBs is often automatic through SYNAPSE and its use of ORMs.

Type	Supported Vendors	Example use cases
Relational	PostgreSQL, MySQL, Oracle	Highly structured content
Document	MongoDB,TokuMX, RethinkDB	General purpose
Columnar	Cassandra	Write-intensive workloads
Search	Elasticsearch	Aggregations and analytics
Graph	Neo4j	Social network modeling

Table 5.1: **DB Types and Vendors Supported by SYNAPSE.**

Second, SYNAPSE provides *programmer simplicity* through a simple programming abstraction based on a publish/subscribe data sharing model that allows programmers to choose their own data update semantics to match the needs of their MVC Web applications. We have demonstrated SYNAPSE's ease of use in a case study that integrates several large and widely-used Web components such as the e-commerce platform, Spree. SYNAPSE is already field-tested in production: a startup, Crowdtap, has been using it to support its microservices architecture serving over 450,000 users for the past two years. Our integration and operation experience indicates that SYNAPSE vastly simplifies the construction and evolution of complex data-driven Web applications, providing a level of agility that is crucial in this burgeoning big-data world.

Finally, SYNAPSE can provide excellent *scalability* with low publisher overheads and modest update propagation delays; we present some experimental data showing that SYNAPSE scales well up to 60,000 updates/second for various workloads. To achieve these goals, it lets subscribers parallelize their processing of updates as much as the workload and their semantic needs permit.

This chapter is organized as follows. §5.2 introduces the MVC abstraction for Web applications. §5.3 describes the SYNAPSE API. §5.4 covers the SYNAPSE architecture. §5.5 demonstrate the usefulness of SYNAPSE in three real world applications. §5.6 presents experimental results. §5.7 discusses related work. Finally, §5.8 presents a summary and concluding remarks of this chapter.

5.2 Background

MVC (Model View Controller) is a widely-used Web application architecture supported by many frameworks including Struts (Java), Django (Python), Rails (Ruby), Symfony (PHP), Enterprise Java Beans (Java), and ASP.NET MVC (.NET). For example, GitHub, Twitter, and YellowPages are built with Rails, DailyMo-

Abstraction	Description
Publisher	Service publishing attributes of a model.
Subscriber	Service subscribing to attributes of a model.
Decorator	Service subscribing and publishing a model.
Ephemeral	DB-less publisher.
Observer	DB-less subscriber.
Virtual attribute	Deterministic functions (can be published).

Table 5.2: **SYNAPSE Abstractions.**

tion and Yahoo! Answers are built with Symfony, and Pinterest and Instragram are built with Django. In the MVC pattern, applications define data models that describe the data, which are typically persisted to DBs. Because the data is persisted to a DB, the model is expressed in terms of constructs that can be manipulated by the DBs. Since MVC applications interact with DBs via ORMs (Object Relational Mappers), ORMs provide the model definition constructs [15].

ORMs abstract many DB-related details and let programmers code in terms of high-level objects, which generally correspond one-to-one to individual rows, or documents in the underlying DB. Although ORMs were initially developed for relational DBs, the concept has recently been applied to many other types of NoSQL DBs. Although different ORMs may offer different APIs, at a minimum they must provide a way to *create*, *update*, and *delete* the objects in the DB. For example, an application would typically instantiate an object, set its attributes in memory, and then invoke the ORM's save function to persist it. Many ORMs and MVC frameworks also support a notion of *active models* [48], which allow developers to specify *callbacks* that are invoked before or after any ORM-based update operation.

MVC applications define controllers to implement business logic and act on the data. Controllers define basic units of work in which data is read, manipulated, and then written back to DBs. Applications are otherwise stateless outside of controllers. Since Web applications are typically designed to respond to and interact with users, controllers typically operate within the context of a *user session*, which means their logic is applied on a per user basis.

API	Description
`publish`, `subscribe`	Annotations to denote which attributes to publish or subscribe.
`before_create`, `before_update`, `before_destroy`	Re-purposed active model callbacks for subscriber update notification. Similar callbacks for after_create/update/destroy.
`add_read_deps`, `add_write_deps`	Specify explicit dependencies for read and write DB queries.
`delivery_mode`	Parameter for selecting delivery semantic.
`bootstrap?`	Predicate method denoting bootstrap mode.

Table 5.3: **SYNAPSE API.**

5.3 SYNAPSE API

SYNAPSE extends the MVC pattern to create an easy-to-use platform for integrating Web services that use heterogeneous DBs. Because MVC frameworks and ORMs provide common abstractions that are often used in practice for Web development, SYNAPSE leverages them to provide a transparent and mostly automatic data propagation layer.

Using the SYNAPSE API shown in Table 5.3, developers make simple modifications to their existing model definitions to share their data across services using SYNAPSE Abstractions shown in Table 5.2. At a high level, an application that uses SYNAPSE consists of one or more publishers, and one or more subscribers. Publishers are services that make attributes of their data models available to subscribers, which maintain their own local, read-only copies of these attributes. SYNAPSE transparently synchronizes changes to published models (creations, updates, or deletions of model instances) from the publisher to the subscriber. It serializes updated objects on the publisher, transmits them to the subscriber, deserializes them, and persists them through the ORM.

A service can subscribe to a model, *decorate* that model by adding new attributes to it, and publish these attributes. By cascading subscribers into publishers developers can create complex ecosystems of Web services that subscribe to data from each other, enhance it with new attributes, and publish it further.

```
# Publisher side (Pub1).
class User
  publish do
    field :name
  end
end
```

```
# Subscriber side (Sub1).
class User
  subscribe from: :Pub1 do
    field :name
  end
end
```

Figure 5.1: **API Example.** Publisher (left), subscriber (right).

This programming model is easy to use and supports powerful use cases as shown in §5.5. We discuss the SYNAPSE API using Ruby-on-Rails, but similar APIs can be built for other frameworks.

5.3.1 SYNAPSE Abstractions

Publishers. To publish a model, the developer simply specifies which attributes within that model should be shared. The code at the top of Figure 5.1 shows how to publish in Ruby using the `publish` keyword, with Synapse-specific code underlined. Each published model has a globally unique URI, given by `app_name/model_name`. SYNAPSE generates a publisher file for each publisher listing the various objects and fields being published and is made available to developers who want to create subscribers for the published data. A factory file is also made available for each publisher that provides sample data for writing integration tests (§5.4.5). Other API calls that can be used by publishers are `add_read_deps`, `add_write_dep`, and `delivery_mode`, discussed in §5.3.3 and §5.4.2.

Subscribers. To subscribe to a published model, the developer simply marks the attributes of interest accordingly. In Figure 5.1, the code at the bottom shows how to subscribe in Ruby to the publisher at the top. Since the model name is the same in the subscriber as the publisher, it does not need to be explicitly identified in conjunction with the `subscribe` keyword. A subscriber application can subscribe to some or all of a publisher's models, and can subscribe to models from multiple publishers. While there may be many subscribers for a given model, there can only be one publisher (the *owner* of that model). The owner is the only service who can create or delete new instances of the model (i.e., *objects*). Moreover, subscribers cannot update attributes that they import from other services, although they can update their own decoration attributes on these models. We enforce this *read-only* subscription model to avoid difficult issues related to concurrent update conflicts from distinct services. That said, SYNAPSE handles concurrent

```
# Notification Subscriber
class User
  subscribe from: :PubApp do
    field :name
    field :email
  end
  after_create do
    unless Synapse.bootstrap?
      self.send_welcome_email
    end
  end
end
```

Figure 5.2: **Callback Example.**

updates made from different servers from the same service. Subscribers often need to perform application-specific processing of updates before applying them to their DBs. For example, a subscriber may need to compute new fields, denormalize data, or send a notification. SYNAPSE supports this by piggybacking upon active model callbacks often supported by MVC frameworks, including `before/after_create`, `before/after_update`, or `before/after_destroy`. The code on the right shows an example of an *after* subscriber callback that sends a welcome email for each newly created User. These callbacks are particularly useful to adapt schemas between publishers and subscribers, as discussed in §5.3.4. Other API calls that can be used by subscribers are `bootstrap?` and `delivery_mode`, discussed in §5.3.3.

Decorators. Decorators are services that subscribe to a model and publish new attributes for it. Conceptually, decorators mix the publisher and subscriber abstractions, although several subtle restrictions apply to them. First, decorators cannot create or delete instances of a model, because they are not its originators. Second, decorators cannot update the attributes of the model that they subscribe to. Third, decorators cannot publish attributes that they subscribe to. Our decorator abstraction encapsulates and enforces these restrictions. As an example, the code at the top of Figure 5.3 shows a decorator service, which decorates the User model from Pub1 with the user's interests. The data used to compute those interests comes from other sources, such as social activity, but is omitted here. Other services can then subscribe to any subset of the

```
# Decorator side (Dec2).
class User
  subscribe from: :Pub1 do
    field :name
  end
  publish do
    field :interests
  end
end
```

```
# Subscriber side (Sub2).
class User
  subscribe from: :Pub1 do
    field :name
  end
  subscribe from: :Dec2 do
    field :interests
  end
end
```

Figure 5.3: **Decorator Example.**

model's attributes or decorations by specifying the originators of those attributes, as shown in the code at the bottom of Figure 5.3. Using decorators, one can construct complex ecosystems of services that enhance the data in various ways, as shown in the examples in §5.5.

Ephemerals and Observers. SYNAPSE aims to support as many use cases for data-driven integration as possible. Often times we find it useful to also support integration of non-persisted models. For example, one could define a mailer application that observes user registrations and sends a welcome message, but does not need to store the data. Similarly, although user-facing services may receive user actions (such as clicks, searches, mouse hovering, etc.), it is backend analytics services that truly use that information. Having the front-end application just pass on (publish) the data onto persisting subscribers is useful in such cases. SYNAPSE hence lets programmers mix persisted models with *ephemerals* (non-persisted published models) and/or *observers* (non-persisted subscribed models). Aside from supporting application-driven needs, non-persisted models are often used to adapt mismatching data models across heterogeneous DBs, as shown in §5.3.4.

Virtual Attributes. To perform data translation between ORMs, SYNAPSE simply calls field getter methods on the publisher side, and then calls the corresponding field setters on the subscriber side. SYNAPSE additionally lets programmers introduce getters and setters for attributes that are not in the DB schema. We call these programmer-provided attributes *virtual attributes*. Virtual attributes are valuable for schema mappings, as shown in §5.3.4.

To support different team workflows, publisher and subscriber declarations can be expressed with either

```
# models/user.rb
class User < ActiveRecord::Base
  publish do
    field :name
  end
end

# models/comment.rb
class Comment < ActiveRecord::Base
  publish do
    :belongs_to :user
  end
end

# config/synapse.rb
# EOF
```

```
# models/user.rb
class User < ActiveRecord::Base
end

# models/comment.rb
class Comment < ActiveRecord::Base
end

# config/synapse.rb
Synapse.define do
  publish :users do
    field :name
  end
  publish :comments do
    belongs_to :user
  end
end
```

Figure 5.4: **Writing Definitions.** SYNAPSE offers an inline model syntax (left) and a configuration file syntax (right) to declare definitions.

the inline model syntax or the configuration file syntax. The inline model syntax allows developers to declare definitions in existing model class scopes. It provides a model centric view to developers. This syntax has been used in the previously shown API examples. The configuration file syntax allows developers to consolidate SYNAPSE declarations in a single file. It provides a SYNAPSE centric view to developers. Both syntaxes offer different benefits, especially when used in conjunction with a source control system. Figure 5.4 showcases the two syntaxes in declaring a User publisher and a Comment publisher.

5.3.2 Comparison with SCRIBE

SYNAPSE is a distributed record-replay system and share many similarities with traditional OS-level record-replay systems. Under SYNAPSE architecture, the publisher application *records* operations that change its

Property	SCRIBE	SYNAPSE
Replicated state	OS state	DB state
Actors	Userspace processes	Web app processes (distributed)
State mutation API	System calls	DB operations
State mutation library	libc	DB driver
Parallelism	Multi-Core	Distributed
Synchronization engine	Causal	Causal
Synchronization protocol	CREW	CREW
Synchronization primitives	Rendezvous points	Dependency tracking
Recorded events	API calls	DB data
Log format	Binary	JSON

Table 5.4: **Similarities Between SCRIBE and SYNAPSE.** Each row depicts the equivalent entity with SCRIBE and SYNAPSE.

state, which is entirely contained in its DB. At a later point in time, the subscriber *replays* these recorded operations mutating its state. Considering a publisher app that publishes all of its models, and a subscriber app running the same code as the publisher app that subscribes to all models, the subscriber is a perfect replica of the publisher. This replica can be used for fault-tolerance purposes, a typical use case of record-replay systems.

SYNAPSE shares many similarities with the deterministic record-replay system SCRIBE. Due to its go live feature, SCRIBE must maintain a consistent and valid OS kernel state at any point in time during replay because the application execution may continue uncontrolled at any point in time. From this perspective, SCRIBE (resp. SYNAPSE) records and replays the state of the OS (resp. DB). Table 5.4 summarizes the similarities between the two record-replay systems. The takeaway is that both systems use a similar causal engine to record and replay state mutation operations interleavings. Due to different semantics needs, SYNAPSE causal replay engine has the ability to replicate the recorded ordering with different levels of faithfulness as explained in §5.3.3. Despite their similarities, the two systems do not share common features.

Could DORA, which provides a superset of SCRIBE features, be sufficient to implement a heterogeneous DB replication engine? The answer is no for two mains reasons. Replaying a recorded execution at the OS-level of a DB on a different DB would be impractical, even with advanced mutable replay techniques. Given

enough differences between the two DBs, resolving replay divergences is impractical. Additionally, one of SYNAPSE's key insight is the use of ORM as data translation layer. Generic mutable replay offers no solution to address this data translation problem. Further, our OS-level record-replay engine do not provide any support for distributed applications, a requirement for multi-node DB support.

5.3.3 SYNAPSE Delivery Semantics

Update delivery semantics define the ordering of updates as viewed by subscribers and are an important part of SYNAPSE's programming model. Different applications may require different levels of semantics: while some may be able to handle overwritten histories, others may prefer to see every single update. Similarly, while some applications may be able to handle updates in any order, others may expect them in an order that respects application logic. In support of applications with different needs, and inspired by well-established prior art [20], SYNAPSE allows publishers and subscribers to use the `delivery_mode` configuration directive to select among three delivery semantics: *global*, *causal*, and *weak*.

Global Ordering. On the publisher side, global order delivery mode means that all object updates will be sequentially ordered by the publisher. On subscribers, it means that the sequential order from a global order publisher will be provided to subscribers. This provides the strongest semantics, but in practice limits horizontal scaling and is rarely if ever used in production systems.

Causal Ordering. Causal ordering identifies for each update U the prior update that must be applied before U to avoid negative effects, such as sending a notification for a new post to an out-of-date friends set. On the publisher, causal order delivery mode means that (1) all updates to the same object are serialized, (2) all updates performed within a controller are serialized to match developer expectations of sequential controller code, and (3) controllers within the same user session are serialized so that all updates performed within the same user session are serialized to match user expectations of Web applications. On the subscriber, causal ordering provides the same three semantics as on the publisher, but also ensures causality between reads and writes across controllers. Specifically, when a subscriber processes an update U, SYNAPSE guarantees that if the subscriber reads objects from its DB that were specified as read dependencies during the publishing, the values of these objects are equal to the ones on the publisher's DB when it performed U. In other words, it's as if the subscriber was given a snapshot of the objects specified as read dependencies along with the update U. These semantics are useful because publisher controllers are stateless, so updates are performed after reading and validating dependent objects (e.g., access control, foreign keys) without relying on caches

(otherwise the publisher would be racy). This mode provides sufficient semantics for many Web applications without the performance limitations of global order delivery mode, as shown by Figure 5.10 in §5.4 and our evaluation in §5.6.4.

Weak Ordering. On the publisher, weak order delivery mode means that all updates to the same object will be sequentially ordered by the publisher, but there is no ordering guarantee regarding updates to different objects. On subscribers, it means that the sequential order of updates for each object is provided, but intermediate updates may be missed or ignored if they are delivered out-of-order. Essentially, weak delivery subscribers always update objects to their latest version. This mode is suitable for applications that have low semantic requirements and provides good scaling properties, but its most important benefit is high availability due to its tolerance of message loss. For example, causal order delivery mode requires delivery of every single update for all objects, so loss of an update would result in failure. In production, unfortunately, situations occur where messages may get lost despite the use of reliable components (see §5.6.5). Weak order delivery mode can ignore causal dependencies and only update to the latest version.

Selecting Delivery Modes. Publishers select the modes that deliver the strongest semantics that they wish to support for their subscribers, subject to the performance overheads they can afford. The more flexible the delivery order semantics, the more SYNAPSE can enable subscribers to process updates with as much parallelism as possible to keep up with a high throughput publisher. Subscribers can only select delivery semantics that are at most as strong as the publishers support. Subscribers can select different delivery modes for data coming from different publishers. In the common case, a publisher would select to support causal delivery, while the subscriber may configure either causal or weak delivery. For example, given a causal mode publisher, a mailer subscriber that sends emails on the state transitions of a shopping cart would not tolerate overwritten histories without additional code, although it is generally tolerable for the mailer service to be unavailable for short periods of time. The causal semantic would be well fit for such a subscriber. In contrast, a real-time analytics service that aggregates million of rows at once may not care about orders, while being unavailable, even for short period of time, may damage the business. The weak semantic would be sufficient for this subscriber.

There is only one constraint on delivery mode choice in SYNAPSE. During the bootstrapping period, which occurs when a subscriber must catch up after a period of unavailability, SYNAPSE forces the weak semantic (i.e., the subscriber may witness overwritten histories and out-of-order deliveries). We signal such periods clearly to programmers in our API using the `bootstrap?` predicate. Figure 5.2 shows a usage

```
# Publisher 1 (Pub1).
# Runs on MongoDB.
class User
  include Mongoid::Document
  publish do
    field :name
  end
end
```

```
# Subscriber 1a (Sub1a).
# Runs on any SQL DB.
class User < ActiveRecord::Base
  subscribe from: :Pub1 do
    field :name
  end
end
```

```
# Subscriber 1b (Sub1b).
# Runs on Elasticsearch.
class User < Stretcher::Model
  subscribe from: :Pub1 do
    property :name, analyzer: :simple
  end
end
```

```
# Subscriber 1c (Sub1c).
# Runs on MongoDB.
class User
  include Mongoid::Document
  subscribe from: :Pub1 do
    field :name
  end
end
```

Figure 5.5: **Example 1: Basic Integration.** Shows publishing/subscribing examples with actual ORMs. SYNAPSE code is trivial. This is the common case in practice.

example and §5.4.4 describes this situation and explains how subscribers demanding higher semantics can deal with semantics degradation.

5.3.4 SYNAPSE Programming by Example

SYNAPSE addresses many of the challenges of heterogeneous-DB applications automatically, often in a completely plug-and-play manner thanks to its use of ORM abstractions. In other cases, the programmer may need to perform explicit translations on the subscriber to align the data models. Our experience suggests that SYNAPSE's abstractions facilitate these translations, and we illustrate our experience using examples showcasing SYNAPSE's usability with each major class of DB: SQL, document, analytic, and graph.

Example 1: Basic Integrations. Our experience suggests that most integrations with SYNAPSE are en-

tirely automatic and require only simple annotations of what should be published or subscribed to, similar to the ones shown in Figure 5.1. For example, Figure 5.5 shows the integration of a MongoDB publisher (Pub1) with three subscribers: SQL (Sub1a), Elasticsearch (Sub1b), and MongoDB (Sub1c). The programmers write their models using the specific syntax that the underlying ORM provides. Barring the `publish/subscribe` keywords, the models are exactly how each programmer would write them if they were not using SYNAPSE (i.e., the data were local to their service). In our experience deploying SYNAPSE, this is by far the most frequent case of integration.

That said, there are at times more complex situations, where programmers must intervene to address mismatches between schemas, supported data types, or optimal layouts. We find that even in these cases, SYNAPSE provides just the right abstractions to help the programmer address them easily and elegantly. We describe next complex examples, which illustrate SYNAPSE's flexibility and great added value. We stress that not all integrations between a given DB pair will face such difficulties, and vice versa, the same difficulty might be faced between other pairs than those we illustrate.

Example 2: Mapping Data Models with Observers. Different DBs model data in different ways so as to optimize different modes of accessing it. This example shows how to map the data models between a SQL and Neo4j DB to best leverage the DBs' functions. Neo4j, a graph-oriented DB, is optimized for graph-structured data and queries. It stores relationships between data items – such as users in a social network or products in an e-commerce app – as edges in a graph and is optimized for queries that must traverse the graph such as those of recommendation engines. In contrast, SQL stores relationships in separate tables. When integrating these two DBs, model mismatches may occur. Figure 5.6 illustrates this use case with an example.

Pub2, the main application, stores Users and their friends in a SQL DB. Sub2, an add-on recommendation engine, integrates the user and friendship information into Neo4j to provide users with recommendations of what their friends or network of friends liked. Its common type of query thus involves traversing the user's social graph, perhaps several levels deep. As in the previous examples, we see here that the programmer defines his subscriber's User model in the way that she would normally do so for that DB (the top of Sub2). However, in this case, SYNAPSE's default translation (achieved by just annotating data with publish/subscribe) would yield low performance since it would store both the user and the friendship models as nodes just like the publisher's SQL schema does, ignoring the benefits of Neo4j.

To instead store friendships as edges in a graph between users, the programmer leverages our observer

```
# Publisher 2 (Pub2).
# Runs on any SQL DB.
class User < ActiveRecord::Base
  publish do
    field :name
    field :likes
  end
  has_many :friendships
end
class Friendship < ActiveRecord::Base
  publish do
    belongs_to :user1, class: User
    belongs_to :user2, class: User
  end
end
```

```
# Subscriber 2 (Sub2).
# Runs on Neo4j.
class User # persisted model
    include Neo4j::ActiveNode
    subscribe from: :Pub2 do
      property :name
      property :likes
    end
    has_many :both, :friends, \
      class: User
end
class Friendship # not persisted
  include Synapse::Observer
  subscribe from: :Pub2 do
    belongs_to :user1, class: User
    belongs_to :user2, class: User
  end
  after_create do
    user1.friends << user2
  end
  after_destroy do
    user1.friends.delete(user2)
  end
end
```

Figure 5.6: **Example 2: SQL/Neo4j.** Pub2 (SQL) stores friendships in their own table; Sub2 (Neo4j) stores them as edges between Users. Edges are added through an Observer.

abstraction. She defines an observer model to subscribe to the Friendship model, which rather than persisting the data as-is, simply adds or removes edges among User nodes. This solution, which involves minimal and conceptually simple programmer input, lets the subscriber leverage Neo4j's full power.

Example 3: Matching Data Types with Virtual Attributes. At times, DBs may mismatch on data types. As an example, we present a specific case of integration between MongoDB and SQL. MongoDB, a document-oriented database, has recently become popular among startups thanks to its schemaless data model that allows for frequent structural changes. Since the DB imposes so little structure, importing data into or exporting data from MongoDB is typically similar to Figure 5.5. We choose here a more corner case example to show SYNAPSE's applicability to complex situations.

Figure 5.7 shows a MongoDB publisher (Pub3), which leverages a special MongoDB feature that is not generally available in SQL, Array types, to store user interests. Figure 5.7 shows two options for integrating the interests in a SQL subscriber, both of which work with all SQL DBs. The first option (Sub3a) is to automatically flatten the array and stores it as text, but this would not support efficient queries on interests.

The typical solution to translate this array type to a generic SQL DB is to create an additional model, Interest, and a one-to-many relationship to it from User. Sub3b shows how SYNAPSE's virtual attribute abstraction easily accomplishes this task, creating the Interest model and a virtual attribute (interests_virt) to insert the new interests received into the separate table.

5.4 SYNAPSE Architecture

Figure 5.8 shows the SYNAPSE architecture applied to an application with a single publisher and subscriber. The publisher and subscriber may be backed by different DBs with distinct engines, data models, and disk layouts. In our example, the publisher runs on PostgreSQL, a relational DB, while the subscriber runs on MongoDB, a document DB. At a high level, SYNAPSE marshals the publisher's model instances (i.e., objects) and publishes them to subscribers, which unmarshal the objects and persist them through the subscribers' ORMs.

SYNAPSE consists of two DB- and ORM-agnostic modules (SYNAPSE *Publisher* and SYNAPSE *Subscriber*), which encapsulate most of the publishing and subscribing logic, and one DB-specific module (SYNAPSE *Query Intercept*), which intercepts queries and relates them to the objects they access. At the publisher, SYNAPSE interposes between the ORM and the DB driver to intercept updates of all published

```
# Publisher 3 (Pub3).
# Runs on MongoDB.
class User
  include Mongoid::Document
  publish do
    field :interests
  end
end
```

```
# Subscriber 3a (Sub3a).
# Runs on any SQL DB.
# Searching for users based on
# interest is not supported.
class User < ActiveRecord::Base
  subscribe, from: :Pub3 do
    field :interests
  end
  serialize :interests
end
```

```
# Subscriber 3b (Sub3b).
# Runs on any SQL DB.
# Supports searching for users by interest.
class User < ActiveRecord::Base
  has_many :interests
  subscribe from: :Pub3 do
    field :interests, as: :interests_virt
  end
  def interests_virt=(tags)
    Interest.add_or_remove(self, tags)
  end
end
class Interest < ActiveRecord::Base
  belongs_to :user
  field :tag
  def self.add_or_remove(user, tags)
    # create/remove interests from DB.
  end
end
```

Figure 5.7: **Example 3: MongoDB/SQL.** Shows one publisher running on MongoDB (Pub3) and two SQL subscribers (Sub3a,b). Default translations work, but may be suboptimal due to mismatches between DBs. Optimizing translation is easy with SYNAPSE.

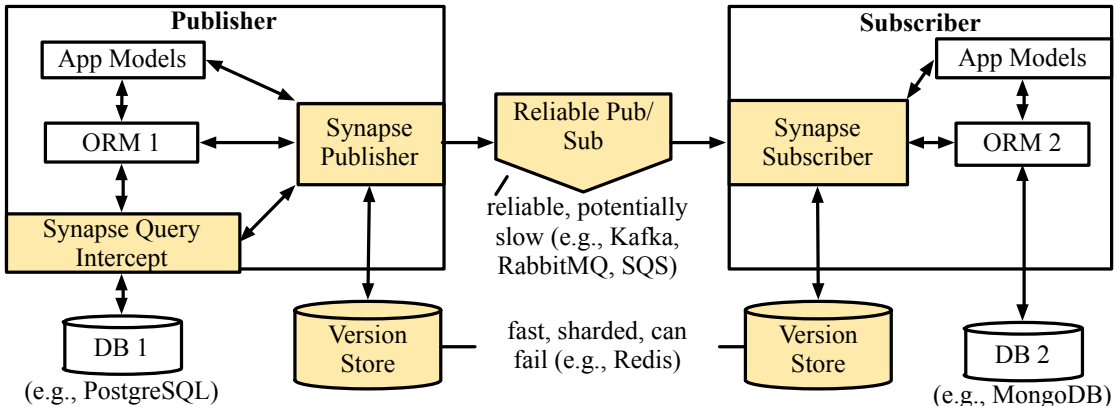


Figure 5.8: **The SYNAPSE Architecture.** SYNAPSE components are shaded. To replicate data between heterogeneous DBs, SYNAPSE marshals the publisher's objects and sends them to subscribers, which unmarshal and save them into their DBs.

models, such as creations, updates, or deletions of instances – collectively called *writes* – before they are committed to the DB. The interposition layer identifies exactly which objects are being written and passes them onto the SYNAPSE *Publisher*, SYNAPSE's DB-independent core. The Publisher then marshals all published attributes of any created or updated objects, attaches the IDs of any deleted objects, and constructs a *write message*. SYNAPSE sends the message to a reliable, persistent, and scalable message broker system, which distributes the message to the subscribers. All writes within a single transaction are combined into a single message.

The message broker reliably disseminates the write message across subscribers. Of the many existing message brokers [121; 8; 89], we use RabbitMQ [89] in our implementation, using it to provide a dedicated queue for each subscriber app. Messages in the queue are processed in parallel by multiple subscriber workers per application, which can be threads, processes, or machines.

When a new message is available in the message broker, a SYNAPSE subscriber worker picks it up and unmarshals all received objects by invoking relevant constructors and attribute setters (using the language's reflection interface). The worker then persists the update to the underlying DB and then acks the message to the broker.

```
{
  app: "pub3",
  operations: [ {
   operation: "update",
   type: ["User"],
   id: 100,
   attributes: {
    interests: ["cats", "dogs"]
   }
  } ],
  dependencies: { "pub3/users/id/100": 42 },
  published_at: "10/11/14 07:59:00",
  generation: 1
}
```

Figure 5.9: **Published Message Format (JSON).**

5.4.1 Model-Driven Replication

To synchronize distinct DBs, SYNAPSE needs to (1) identify the objects being written on the publisher, (2) marshal them for shipping to the subscribers, (3) unmarshal back to objects at the subscriber, (4) and persist them. Although steps 1 and 4 seem completely DB specific, we leverage ORMs to abstract most DB specific logic.

To intercept writes, SYNAPSE uses a DB-engine specific query interceptor. Collecting information about objects written is generally straightforward, as many DBs can easily output the rows affected by each query. For example, in SQL, an `INSERT`, or `DELETE` query ending with `RETURNING *` will return the contents of the written rows. Many DBs support this feature, including: Oracle, PostgreSQL, SQL Server, MongoDB, TokuMX, and RethinkDB. For DBs without this feature (e.g., MySQL, Cassandra), we develop a protocol that involves performing an additional query to identify data being written; it is safe but somewhat more expensive.

After intercepting a write, SYNAPSE uses the ORM to map from the raw data written back to application objects (e.g., an instance of the `User` model). The published attributes of these written object(s) are marshaled to JSON, and published along with object dependencies (described in §5.4.2) and a generation

number (for recovery, described in §5.4.4). When marshalling objects, SYNAPSE also includes each object's complete inheritance tree, allowing subscribers to consume polymorphic models. Figure 5.9 shows a write message produced upon a `User` creation; the post object's marshalling is in the message's `attributes` field.

On the subscriber, SYNAPSE unmarshals a new or updated object by (1) instantiating a new instance of that type, or finding it in the DB based on its primary key with the ORM's `find` method, (2) recursively assigning its subscribed attributes from those included in the message by calling the object setter methods, and (3) calling the `save` or `destroy` method on the object. For a delete operation, step (2) is skipped. Different ORMs may have different names for these methods (e.g., `find` vs `find_by`) but their translation is trivial. Any callbacks specified by the programmer are automatically called by the ORM.

5.4.2 Enforcing Delivery Semantics

SYNAPSE enforces update-message ordering from publishers to subscribers to be able to provide subscribers with a view of object updates that is consistent with what would be perceived if the subscribers had direct access to the publisher's DB. Specific consistency semantics are determined based on the choice of publisher and subscriber delivery order modes, global, causal, and weak. In all cases, SYNAPSE uses the same general update delivery mechanism, which is inspired by deterministic execution record-replay of SCRIBE and DORA.

The update delivery mechanism identifies dependencies on objects during persistence operations and tracks their version numbers to send to subscribers. SYNAPSE defines an operation as having a dependency on an object if the operation, or the construction of the operation, may reference the object. On the publisher, SYNAPSE tracks two kinds of dependencies: read and write dependencies. An operation has a read dependency on an object if the object was read, but not written, and used to construct the given operation. An operation has a write dependency on an object if the operation modifies the object. An operation may have a combination of both read and write dependencies on different objects. Since an object may have many versions due to updates, SYNAPSE uses version numbers to track object versions and expresses dependencies in terms of specific object versions. For each object, SYNAPSE maintains two counters at the publisher, ops and $version$, which represent the number of operations that have referenced the object so far and the version number of the object, respectively. For each operation at the publisher, SYNAPSE identifies its dependencies and uses this information to publish a message to the subscriber.

At the publisher, SYNAPSE performs the following steps. First, locks are acquired on the write dependencies. Then, for each dependency, a) SYNAPSE increments ops, b) sets $version$ to ops in the case of a write dependency, and c) use $version$ for read dependencies and $version - 1$ for write dependencies as the version to be included in the final message. Next, the operation is performed, written objects are read back, and locks are released. Finally, the message is prepared and sent to subscribers. In our implementation, the publishing algorithm is slightly more complex due to 2PC protocols at every step of the algorithm to allow recovery at any point in case of failures. At the subscriber, SYNAPSE maintains a version store that keeps track of the latest ops counter for each dependency. In contrast, the publisher maintains two counters per dependency. When the subscriber receives a message, it waits until all specified dependencies' versions in its version store are greater than or equal to those in the message, then processes the message and updates its version store by incrementing the ops counter for each dependency in the message. When subscribers publish messages (e.g., when decorating models), published messages include dependencies from reading other apps objects allowing cross-application dependencies. These *external* dependencies behave similarly to read dependencies, except they are not incremented at the publisher nor the subscriber, relaxing semantics to a level similar to traditional causal replication systems [65; 66].

With this mechanism in place, SYNAPSE can enforce various update-message ordering semantics. To support global order delivery from the publisher, SYNAPSE simply adds a write dependency on a global object for every operation which serializes all writes across all objects because all operations are serialized on the global object. To support causal order delivery from the publisher, SYNAPSE serializes all updates within a controller by adding the previously performed update's first write dependency as a read dependency to the next update operation. To serialize all writes within a user context, it is sufficient to add the current user object as a write dependency to each write operation (shown in Figure 5.10(b)). To support weak order delivery from the publisher, SYNAPSE only tracks the write dependency for each object being updated. To support the same delivery mode at the subscriber as provided by the publisher, SYNAPSE respects all the dependency information provided in the messages from the publisher. In the case of weak order delivery, the subscriber also discards any messages with a $version$ lower than what is stored in its version store. To support weaker delivery semantics at the subscriber, SYNAPSE ignores some of the dependency information provided in the messages from the publisher. For example, a causal order delivery subscriber will ignore global object dependencies in the messages from a global order delivery publisher. Similarly, a weak order

delivery subscriber will respect the dependency information in the message for the respective object being updated by the message, but ignore other dependency information from a global or causal order delivery publisher.

Figure 5.10 shows an example of how the read and write dependencies translate into dependencies in messages. Both publisher and subscriber use causal delivery mode. Figure 5.10(a) shows four code snippets processing four user requests. User1 creates a post, User2 comments on it, User1 comments back on it, and then User1 updates the post. SYNAPSE automatically detects four different writes, automatically detects their dependencies, updates the version store, and generates messages as shown in Figure 5.10(b). A subscriber processing these messages would follow the dependency graph shown in Figure 5.10(c) by following the subscriber algorithm.

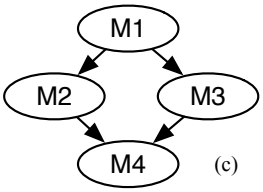
Tracking Dependencies. To enforce causal ordering semantics, SYNAPSE discovers and tracks dependencies between operations. SYNAPSE implicitly tracks data dependencies within the scope of individual controllers (serving HTTP requests), and the scope of individual background jobs (e.g., with Sidekiq [84]). Within these scopes, SYNAPSE intercepts read and write queries to transparently detect corresponding dependencies. In contrast, prior work relies on explicit dependencies and requires their use with *all writes* [12; 11], which is onerous for developers and error-prone.

SYNAPSE always infers the correct set of dependencies when encountering read queries that return objects, including joins. For example, when running a query of the form `SELECT id,... FROM table WHERE ...`, SYNAPSE registers an object dependency on each returned row. Note that SYNAPSE automatically injects primary key selectors in read queries if these are missing. In our experience, read queries returning objects constitute the vast majority of true dependency queries. The other types of read queries are aggregations (e.g., count) and their results are not true dependencies in practice. However, in the hypothetical case where one would need to track dependencies on such queries, SYNAPSE lets developers express explicit dependencies with `add_read_deps` and `add_write_deps` to synchronize arbitrary read queries with any write queries. In over a dozen applications we integrated with SYNAPSE, we have not encountered one single query that could be considered as a dependency but is not marked as such by SYNAPSE automatically.

SYNAPSE always infers the correct set of dependencies when encountering write queries. When encountering write queries issued to transactional DBs, SYNAPSE infers the updated objects from the result of the write query (with `RETURNING *`), or by performing an additional read query. It registers these ob-

```
# Users issue requests
# against the application.
# '—' separates controller executions.
current_user = User.find(1)
Post.create(
  author: current_user,
  body: "helo")
—
current_user = User.find(2)
post = Post.find(1)
Comment.create(post: post,
  author: current_user,
  body: "you have a typo")
—
current_user = User.find(1)
post = Post.find(1)
Comment.create(post: post,
  author: current_user,
  body: "thanks for noticing")
—
current_user = User.find(1)
post = Post.find(1)
post.update(body: "hello")
```

(a)



(c)

```
# Assume "post/id/1" hashes to "p1",
# "user/id/2" hashes to "u2", etc.
W1: write(...,
  read_deps: [],
  write_deps: ["user/id/1", "post/id/1"])
# u1.ops = 1, u1.version = 1
# p1.ops = 1, p1.version = 1
M1: {..., dependencies: ["u1": 0, "p1": 0]}
W2: write(...,
  read_deps: ["post/id/1"],
  write_deps: ["user/id/2", "comment/id/1"])
# u2.ops = 1, u2.version = 1
# c1.ops = 1, c1.version = 1
# p1.ops = 2, p1.version = 1
M2: {..., dependencies: ["u2": 0, "c1": 0, "p1": 1]}
W3: write(...,
  read_deps: ["post/id/1"],
  write_deps: ["user/id/1", "comment/id/2"])
# u1.ops = 2, u1.version = 2
# c2.ops = 1, c2.version = 1
# p1.ops = 3, p1.version = 1
M3: {..., dependencies: ["u1": 1, "c2": 0, "p1": 1]}
W4: write(...,
  read_deps: [],
  write_deps: ["user/id/1", "post/id/1"])
# u1.ops = 3, u1.version = 3
# p1.ops = 4, p1.version = 4
M4: {..., dependencies: ["u1": 2, "p1": 3]}
```

(b)

Figure 5.10: **Dependencies and Message Generation.** (a) shows controller code being executed at the publisher. (b) shows the writes SYNAPSE instruments with their detected dependencies, along with the publisher's version store state updates in comments, and the resulting generated messages. (c) shows a dependency graph resulting from applying the subscriber algorithm. M2 and M3 are processed when the typo is present in the post.

jects as write dependencies, which are later used during the commit operation. With non-transactional DBs, SYNAPSE supports write queries that update at most one, well identified object (so the corresponding lock can be acquired before performing the write query). Multi-object updates queries are seldom used as their usage prevents model-defined callbacks to be triggered. However, when encountering such query, SYNAPSE unrolls the multi-object update into single-object updates.

Scaling the Version Store. We implemented the version stores with Redis [93], an in-memory datastore. All SYNAPSE operations are performed by atomically executing LUA scripts on Redis. This technique avoids costly round-trips, and simplifies the 2PC implementation in the algorithm. Scaling can be problematic in two ways. First, the version store can become a throughput bottleneck due to network or CPU, so SYNAPSE shards the version store using a hash ring similar to Dynamo [38] and incorporates mechanisms to avoid deadlocks on subscribers as atomicity of the LUA scripts across shards can not be assumed. Second, the version store memory can be limiting, so SYNAPSE hashes dependency names with a stable hash function at the publisher. This way, all version stores consume $O(1)$ memory. When a hash collision occurs between two dependencies, serialization happens between two unrelated objects, reducing parallelism. The number of effective dependencies that SYNAPSE uses is the cardinal of the hashing function output space. Each dependency consumes around 100 bytes of memory, so a 1GB server can host 10M effective dependencies, which is more than enough in practice. As an interesting property, using a 1-entry dependency hash space is equivalent to using global ordering for both the publisher and its subscribers.

Transactions. When publishers support transactions, we enforce the same atomicity in messages delivery, allowing these properties to hold for subscribers. All writes in a transaction are included in the same message. Subscribers process messages in a transaction with the highest level of isolation and atomicity the underlying DB permits (e.g., logged batched updates with Cassandra). At the publisher, we also hijack the DB driver's transaction commit functions and execute the transaction as a two-phase commit (2PC) transaction instead. The 2PC lets us ensure that either the following operations all happen or that none do: (1) commit the transaction locally, (2) increment version dependencies and (3) publish the message to the reliable message broker. As an optimization, the publisher algorithm does not attempt to lock write dependencies as the underlying DB retains locks on the written objects until the commit is persisted.

5.4.3 Live Schema Migrations

When deploying new features or refactoring code, it may happen that the local DB schema must be changed, or new data must be published or subscribed. A few rules must be respected: 1) Updating a publisher DB schema must be done in isolation such that subscribers are not able to observe the internal changes done to the publisher. For example, before removing a published attribute from the DB, a virtual attribute of the same name must be added. 2) The semantics of a published attribute must not change; e.g., its type must not change. Instead of changing the semantics of a published attribute, one can publish a new attribute, and eventually stop publishing the old one. 3) Publishing a new attribute is often motivated by the need of a subscriber. When adding the same attribute in a publisher and subscriber, the publisher must be deployed first. Finally, once the new code is in place, a partial data bootstrap may be performed to allow subscribers to digest newly subscribed data.

5.4.4 Bootstrapping and Reliability

When a new subscriber comes online, it must synchronize with the publisher in a three-step bootstrapping process. First, all current publisher versions are sent in bulk and saved in the subscriber's version store. Second, all objects in the subscribed model are sent and persisted to the subscriber's DB. Third, all messages published during the previous steps are processed to finish synchronizing the objects and versions. Once all messages are processed, the subscriber is now *in sync* and operates with the configured delivery semantics. Subscriber code may call `Synapse.bootstrap?` to determine whether SYNAPSE is still bootstrapping or in sync. Figure 5.2 shows an example of how the mailer subscriber checks for bootstrapping completion before sending emails.

Should the subscriber fail, its queue may grow to an arbitrary size. To alleviate this issue, SYNAPSE decommissions the subscriber from the SYNAPSE ecosystem and kills its queue once the queue size reaches a configurable limit. If the subscriber comes back, SYNAPSE initiates a *partial bootstrap* to get the application back in sync.

Failures may also happen when the version store dies on either the publisher or subscriber side. When the subscriber's version store dies, a partial bootstrap is initiated. When the publisher's version store dies, a generation number reliably stored (e.g., Chubby [24], or ZooKeeper [7]) is incremented and publishing resumes. Messages embed this generation number as shown on Figure 5.9. When subscribers see this new generation number in messages, they wait until all the previous generation messages are processed,

DB	ORM	Pub?	Sub?	ORM LoC	DB LoC
PostgreSQL	ActiveRecord	Y	Y	474	44
MySQL	ActiveRecord	Y	Y	”	52
Oracle	ActiveRecord	Y	Y	”	47
MongoDB	Mongoid	Y	Y	399	0
TokuMX	Mongoid	Y	Y	”	0
Cassandra	Cequel	Y	Y	219	0
Elasticsearch	Stretcher	N/A	Y	0	0
Neo4j	Neo4j	N	Y	0	0
RethinkDB	NoBrainer	N	Y	0	0
Ephemerals	N/A	Y	N/A	N/A	N/A
Observers	N/A	N/A	Y	N/A	N/A

Table 5.5: **Support for Various DBs.** Shows ORM- and DB-specific lines of code (LoC) to support varied DBs. For ORMs supporting many DBs (e.g., ActiveRecord), adding a new DB comes for free.

flush their version store, and process the new generation messages. This generation change incurs a global synchronization barrier and temporarily slows subscribers.

5.4.5 Testing Framework

SYNAPSE provides a solid testing framework to help with development and maintenance of apps. For instance, SYNAPSE statically checks that subscribers don't attempt to subscribe to models and attributes that are unpublished, providing warnings immediately. SYNAPSE also simplifies integration testing by reusing model factories from publishers on subscribers. If a publisher provides a model factory [44] (i.e., data samples), then developers can use them to write integration tests on the subscribers. SYNAPSE will emulate the payloads that would be received by the subscriber in a production environment. This way, developers are confident that the integration of their ecosystem of applications is well tested before deploying into the production environment.

5.4.6 Supporting New DBs and ORMs

Adding subscriber support for a new ORM is trivial: a developer need only map the CRUD operations (create/read/update/delete) to SYNAPSE's engine. To add publisher support for a new ORM, a developer needs to first plug into the ORM's interfaces to intercept queries on their way to the DB (all queries for causality and writes for replication). Then, the developer needs to add two phase commit hooks to the DB driver for transactional DBs (as discussed in §5.4.2).

To illustrate the effort of supporting new DBs and ORMs, we report our development experience on the nine DBs listed in Table 5.5. A single developer implemented support for our first DB, PostgreSQL in approximately one week, writing 474 lines of code specific to the ORM (ActiveRecord) and 44 lines specific to the DB for two phase commit. After building support for this DB, supporting other SQL DBs, such as MySQL and Oracle, was trivial: about 50 lines of DB-specific, each implemented in only several hours. Supporting subsequent DBs (e.g., MongoDB, TokuMX, and Cassandra) was equally easy and took only a few days and 200-300 lines of code per ORM. We find that supporting various DBs is a reasonable task for an experienced programmer.

5.5 Applications

We and others have built or modified 14 web applications to share data with one another via SYNAPSE. Those built by others have been deployed in production by a startup, Crowdtap. The applications we built extend popular open-source apps to integrate them into data-driven ecosystems. We also split a popular open-source app into a service oriented architecture to demonstrate the usefulness of SYNAPSE in the context of single applications. Overall, our development and deployment experience has been positive: we made *no logical changes* to application code, only adding on average a single line of configuration per attribute of each model published.

5.5.1 SYNAPSE at Crowdtap

Crowdtap is an online marketing-services company contracted by major brands such as Verizon, AT&T, Sony and MasterCard. Crowdtap has grown rapidly since its founding, and by October 2014 has seen over 450,000 users. As Crowdtap grew and gained more clients, both their application offering and development team evolved. At first, engineers attempted to enhance their core application by building new features

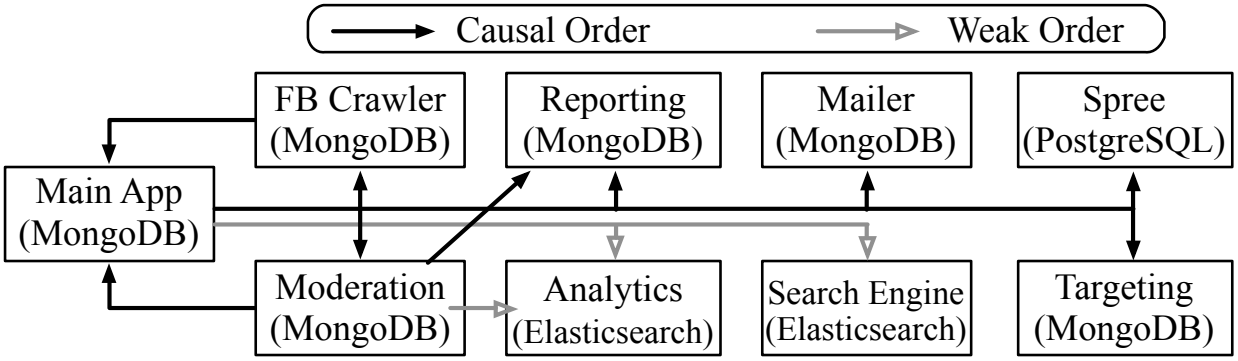


Figure 5.11: **Crowdtap's Services.** Arrows show SYNAPSE connections.

directly into the same codebase and DB. However, as new features were added, the data was used in different ways, requiring different indexes and denormalization, bloating the DB. When features were canceled, traces of their schema changes were often left orphaned. Moreover, it was difficult to bring newly hired engineers up to speed with the complex and rapidly evolving codebase and DB. To alleviate this issue, engineers factored out some features and used synchronous APIs to connect them to the core DB, but found this technique difficult to get right. Specifically, a bug in an e-commerce service, which accessed user data from Crowdtap's core app via a synchronous API, was able to bring down the entire app due to the lack of performance isolation. Installing rate limits between the components of their app was not good option for Crowdtap, and they preferred the de-coupling that a replication-based solution cross-service would provide. Synchronization of these DBs then became a challenge.

To address these challenges, Crowdtap began to experiment with SYNAPSE, first to provide synchronization between their original core app and a separate targeting service. These services had previously communicated over a synchronous API; this API was factored out, and replaced with SYNAPSE, reducing the app from 1500 LoC to 500 LoC. While previous attempts to integrate this feature required deep knowledge of the core application held by senior engineers, this integration was performed by a newly-hired engineer thanks to the abstractions provided by SYNAPSE.

After this initial integration, two other Crowdtap engineers extracted two other business features, the mailer and the analytics engine from the main application, using SYNAPSE. The email service subscribes to 24 of the core service's models to isolate all notification related aspects within their application, while the analytics engine subscribes to a similar number of models.

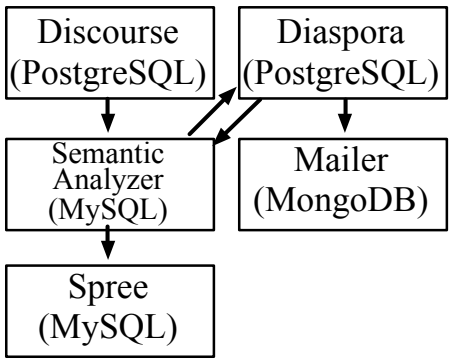


Figure 5.12: **Social Product Recommender.** Arrows show SYNAPSE connections.

Crowdtap's engineering management was so pleased with the ease of development, ease of maintenance, and performance of SYNAPSE, that after this experience, all major features have been built with it. Figure 5.11 shows the high-level architecture of the SYNAPSE ecosystem at Crowdtap with the main app supported by eight microservices. SYNAPSE-related code is trivial in size and logic. The Crowdtap main app consists of approximately 17,000 lines of ruby code, and publishes 257 attributes of 53 different models. However, SYNAPSE-related configuration lines are minimal: only 360 (less than 7 lines of configuration per model, on average).

Crowdtap has chosen different delivery semantics for their various subscribers (shown with different arrows in Figure 5.11). While all publishers are configured to support causal delivery mode, subscribers are configured with either causal or weak delivery modes, depending on their semantics requirements. The mailer service registers for data from the main app in causal mode so as to avoid sending inconsistent emails. In contrast, the analytics engine lacks stringent order requirements, hence selects a weak consistency mode.

5.5.2 Integrating Open-Source Apps with SYNAPSE

We used SYNAPSE to build a new feature for *Spree*, a popular open source e-commerce application that powers over 45,000 e-commerce websites world wide [101]. By integrating *Diaspora*, a Facebook-like open source social networking application and *Discourse*, an open source discussion board, with Spree, we were able to create a social-based product recommender. Figure 5.12 shows the architecture of the ecosystem. We configured Diaspora and Discourse to publish the models for posts, friends, and access control lists. We needed to add only several lines of declarative configuration each app: 23 for Diaspora (compared to its 30k lines of code), 5 for Discourse (compared to its 21k lines), and 7 for Spree (compared to its 37k lines).

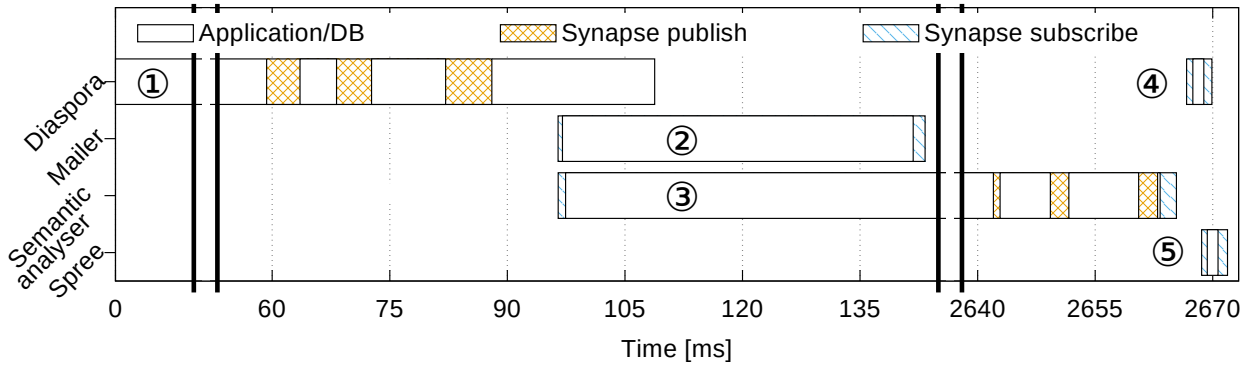


Figure 5.13: **Execution Sample.** ① A user posts on Diaspora. The mailer ② and semantic analyzer ③ receive the post in parallel. Diaspora ④ and Spree ⑤ each receive the decorated model with in parallel.

Next, we built a semantic analyzer that subscribes to these posts and extracts topics of interest, decorating Users with apparent topics of interest (using an out-of-the-box semantic analyzer, Textalytics [108]). The analyzer publishes its decorated `User` model (with user interests) to Spree.

Finally, since Spree did not have *any* recommendation mechanism in place, we added several lines of code to it to implement generic targeted searching. With this code in place, one can construct as complex of a recommendation engine as desired, although our prototype uses a very simple keyword-based matching between the users' interests and product descriptions. Such code need not be concerned with where the user's interests come from as they automatically exist as part of the data model (thanks to SYNAPSE).

5.5.3 Splitting a monolithic app into services with SYNAPSE

We used SYNAPSE to split an open source monolithic application, GitLab, into a service oriented architecture application. GitLab is a source revision system frontend similar to GitHub. GitLab is built on Ruby-on-Rails and used by more than 100,000 companies. With SYNAPSE, we were able to extract all email functionality into a separate service, independent of the core application, running on its own database.

We extracted 276 lines from GitLab core application into the mailer. 10 models had to be published out of a total of 22. To increase our confidence that the extraction was successful, we leveraged SYNAPSE testing framework. We were able to migrate 124 tests cases (700 lines of code) related to the mailer from the GitLab core application to the extracted mailer service. We did not have to rewrite or modify any of these test cases. SYNAPSE transparently reuse the data factories to mock SYNAPSE messages as if they were received from the main application, exactly like it would happen in a deployed system. These passing test

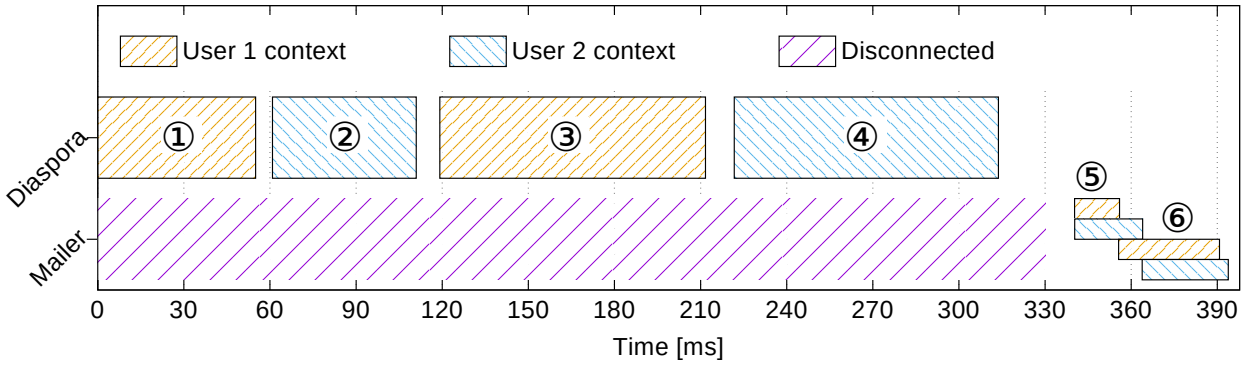


Figure 5.14: **Execution with Subscriber Disconnection.** ① and ③ User 1 posts. ② and ④ User 2 posts. Mailer comes online and processes the each users first request ⑤, then each user's second request ⑥ in parallel.

cases demonstrate the correctness of the feature extraction into its own service.

Even through SYNAPSE greatly eases data sharing between the two services, extracting the mailer functionality from the core GitLab application took a significant effort due to tight coupling between components in the original application.

5.6 Evaluation

5.6.1 Sample Executions

We leverage both our deployment and the applications we built to answer three core evaluation questions about SYNAPSE: (Q1) How expensive is it at the publisher? (Q2) How well does it scale? (Q3) How do its various delivery modes compare? and (Q4) How useful is it in practice?

To answer these questions, we ran experiments on Amazon AWS with up to 1,000 c3.large instances (2-core, 4GB) running simultaneously to saturate SYNAPSE. We use the in-memory datastore Redis [93] for version stores. As workloads, we used a mix of Crowdtap production traffic and microbenchmarks that stress the system in ways that production workload cannot. Unless otherwise noted, our evaluation focuses on the causal delivery mode for both publishers and subscribers, which is the default setting in our prototype. After providing some sample executions, we next discuss each evaluation question in turn.

To build intuition into how SYNAPSE behaves and the kinds of overheads it brings, we show two sample executions of our open-source ecosystem applications (see §5.5.2). All applications are configured with a

Most Popular Controllers	% Calls (of 170k)	Published Messages		Dependencies per Message		Controller Time (ms)		SYNAPSE Time (ms)	
		mean	99^{th}	mean	99^{th}	mean	99^{th}	mean	99^{th}
awards/index	17.0%	0.00	0	0.0	0	56.5	574.1	0.0 (0.0%)	0
brands/show	16.0%	0.03	2	1.0	2	97.6	333.4	0.8 (0.8%)	44.9
actions/index	15.0%	0.67	3	17.8	339	181.4	1676.8	14.4 (8.6%)	114.7
me/show	12.0%	0.00	0	0.0	0	14.7	39.3	0.0 (0.0%)	0
actions/update	11.5%	3.46	6	1.8	4	305.9	759.0	84.1 (37.9%)	207.9

Overhead across all 55 controllers: mean=8%

Table 5.6: **Crowdtap Dependencies and Overheads.** For each of the five most frequently invoked controllers in Crowdtap, shows the percent of calls to it, average number of published messages, average number of dependencies between messages, the average controller execution time, and the average overhead from SYNAPSE. Data sampled from production data.

causal delivery mode, hence the examples reflect this mode's functioning.

Figure 5.13 shows a timeline of the applications' execution starting with a user's post to Diaspora and ending with Spree's receipt of the semantically-enhanced User model. We observe that SYNAPSE delivers messages shortly after publication (within 5ms), in parallel to both the mailer and the semantic analyzer. Figure 5.14 illustrates visually SYNAPSE's causal engine in action. It shows two users posting messages on different Diaspora profiles app while a Mailer subscriber is deployed to notify a user's friends whenever the user makes a new post. Initially, the mailer is disconnected. When the mailer comes back online, it processes messages from the two users in parallel, but processes each user's posts in serial order, thereby enforcing causality.

5.6.2 Application Overheads (Q1)

We next evaluate SYNAPSE's publishing overheads in the context of real applications: Crowdtap and the open-source apps we modified. For Crowdtap, we instrumented Crowdtap's main Web application to record performance metrics and recorded accesses to the application over the 24 hour period of April 16, 2014. In total, we recorded one fifth of the traffic totaling 170,000 accesses to application controllers. For each controller, we measured the number of published messages, the number of dependencies per message published, the total execution time of the controller, and the SYNAPSE execution time within these controllers.

For each of these quantities, we measured the arithmetic mean and 99^{th} percentile.

Figure 5.6 shows our results. In total, 55 controllers were invoked. We show average overheads across them, as well as detailed information about the five most frequently accessed controllers, which account for over 70% of the traffic. On average, SYNAPSE overheads are low: 8%. For the most popular two controllers (`awards/index` and `brands/show`), the overheads are even lower: 0.0-0.8%. This is because they exhibit very few published messages (writes). As expected, SYNAPSE overhead is higher in controllers that publish more messages, showing an average overhead of 37.9% for the controller `actions/update`. The number of dependencies stays low enough to not become a bottleneck with `actions/index` showing 17.8 dependencies per message on average. To further illustrate SYNAPSE's impact, we also show the 99^{th} percentile of controller execution time and SYNAPSE's execution time within the controller. Some controller executions are long (> 1 second). These high response times may be attributed to network issues, and Ruby's garbage collector. Indeed, SYNAPSE is unlikely to be the cause of these latency spikes as the 99^{th} percentile of its execution time remains under 0.2s.

To complement our Crowdtap results, we measured controllers in our open-source applications, as well. Figure 5.15 shows the SYNAPSE overheads for several controllers within Diaspora and Discourse (plus Crowdtap for consistency). Grey areas are SYNAPSE overheads. Overheads remain low for the two open-source applications when benchmarked with synthetic workloads. Read-only controllers, such as `stream/index` and `topics/index` in Diaspora and Discourse, respectively, exhibit near-zero overheads; write controllers have up to 20% overhead.

These results show that SYNAPSE overheads with real applications are low and likely unnoticeable to users. However, the results are insufficient to assess performance under stress, a topic that we discuss next.

5.6.3 Scalability (Q2)

To evaluate SYNAPSE throughput and latency under high load, we developed a stress-test microbenchmark, which simulates a social networking site. Users continuously create posts and comments, similar to the code on Figure 5.10. Comments are related to posts and create cross-user dependencies. We issue traffic as fast as possible to saturate SYNAPSE, with a uniform distribution of 25% posts and 75% comments. We run this experiment by deploying identical numbers of publishers and subscribers (up to 400 for each) in Amazon AWS. We use several of our supported DBs and combinations as persistence layers. We applied different DBs as publishers and subscribers. We measure a variety of metrics, including the overheads for creating a

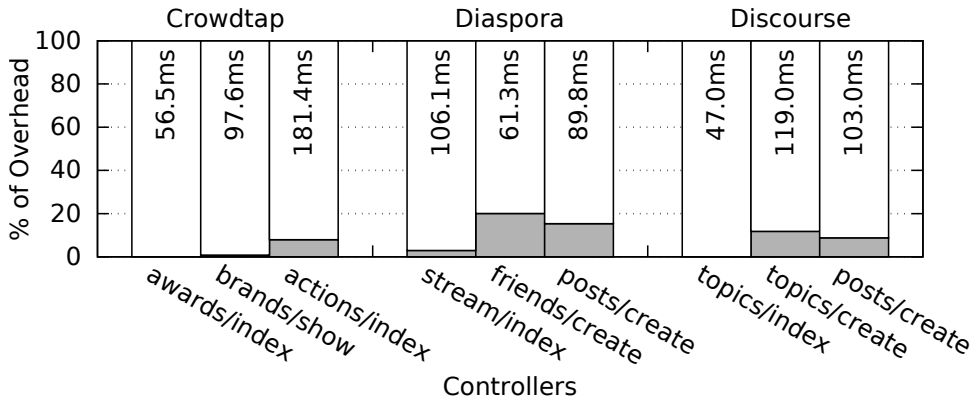


Figure 5.15: **SYNAPSE Overhead.** Shown for 3 controllers in 3 different applications. Gray bars depict overhead. Labels give the total controller times.

post, as well as SYNAPSE's throughput.

Overheads under Heavy Load. Figure 5.16 shows the overheads for different DBs with increasing numbers of dependencies. Focusing on the one-dependency case (x=1), SYNAPSE adds overheads ranging from 4.5ms overhead on Cassandra to 6.5ms on PostgreSQL. This is in comparison to the 0.81ms and 1.9ms latencies that PostgreSQL and Cassandra, respectively, exhibit *without* SYNAPSE. However, compared to realistic Web controller latencies of tens of ms, these overheads are barely user-visible. As the number of dependencies increases, the overhead grows slowly at first, remaining below 10ms for up to 20 dependencies. It then shoots up to a high 173ms for 1,000 dependencies. Fortunately, as shown in Figure 5.6, dependencies in real applications remain low enough to avoid causing a bottleneck.

Cross-DB Throughputs. Figure 5.17 shows how SYNAPSE's end-to-end throughput scales with the number of publisher/subscriber workers, for various DB combinations, as well as for our DB-less models (observer to ephemeral). We keep the number of dependencies per message constant at 4 and shard the version stores on 80 AWS instances. We have not sharded any of the DBs. For ephemerals, SYNAPSE scales linearly with the number of workers, reaching a throughput of more than 60,000 msg/s. Even at such high rates, SYNAPSE does not become a bottleneck. When DBs are used to back the publishers and subscribers, the throughput grows linearly with the number of workers until one of the DBs saturates. Saturation happens when the slowest of the publisher and subscriber DBs reaches its maximum throughput. For each combination, we label the limiting DB with a *. For instance, PostgreSQL bottlenecks at 12,000 writes/s, and Elasticsearch at 20,000 writes/s.

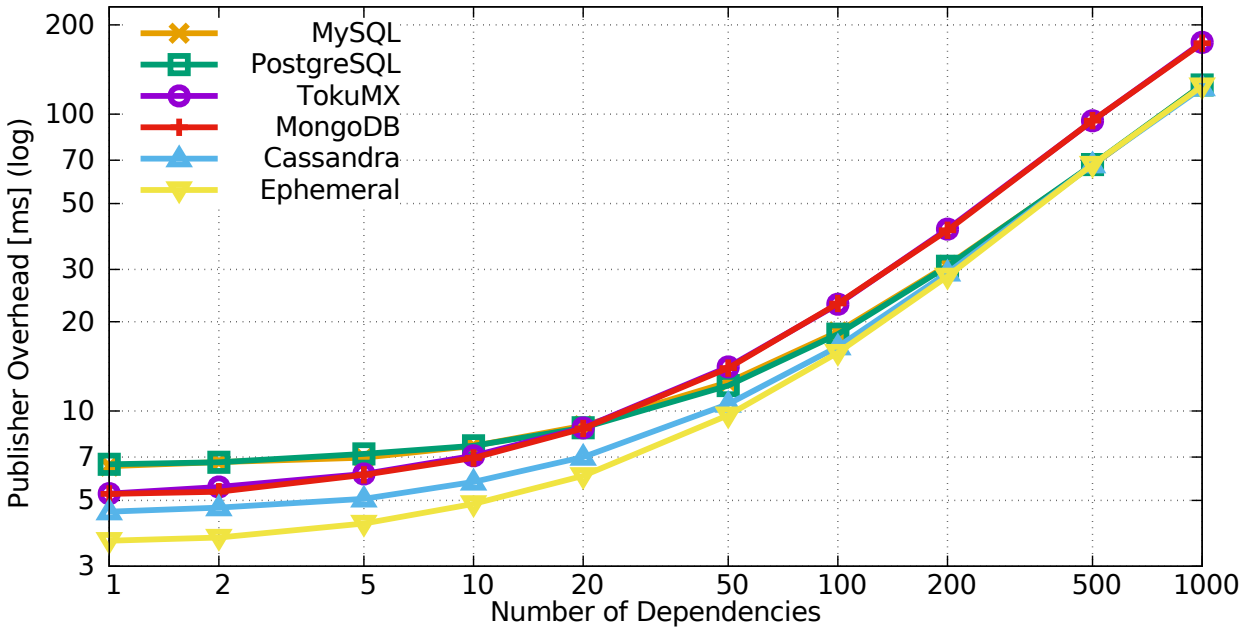


Figure 5.16: **Publisher Overhead on Different DBs.** Each line represents a different DB.

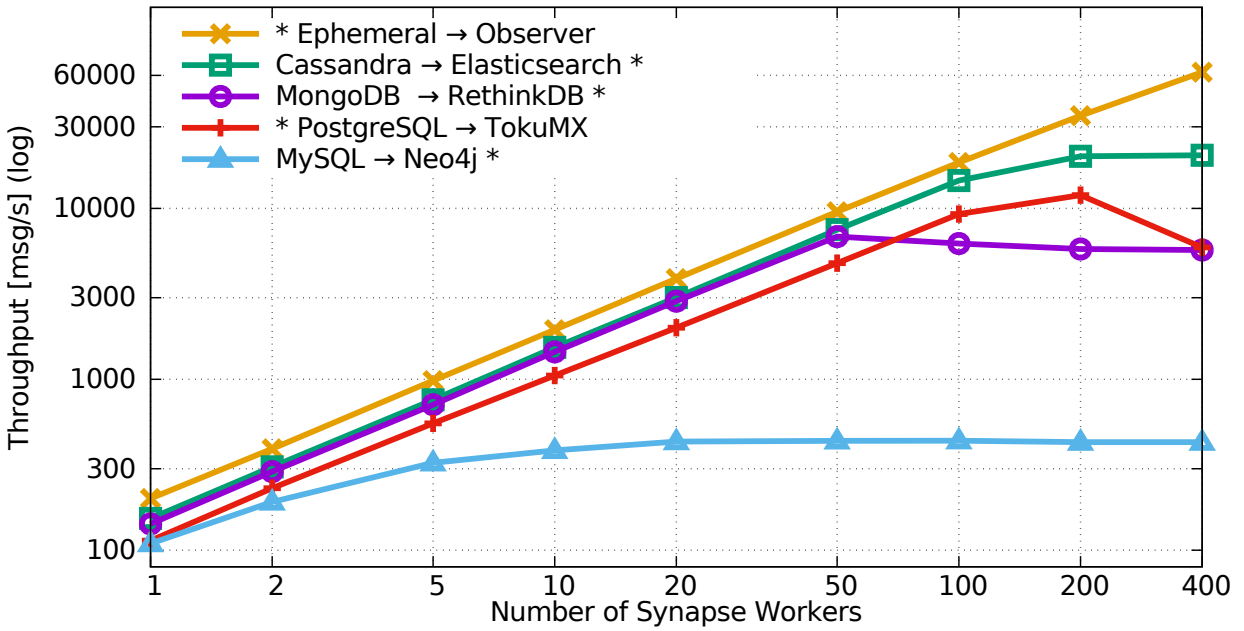


Figure 5.17: **Throughput vs Number of Workers.** End to end benchmark. Each line represents a different DB setup. The slowest end in each pair is annotated with a (*) symbol.

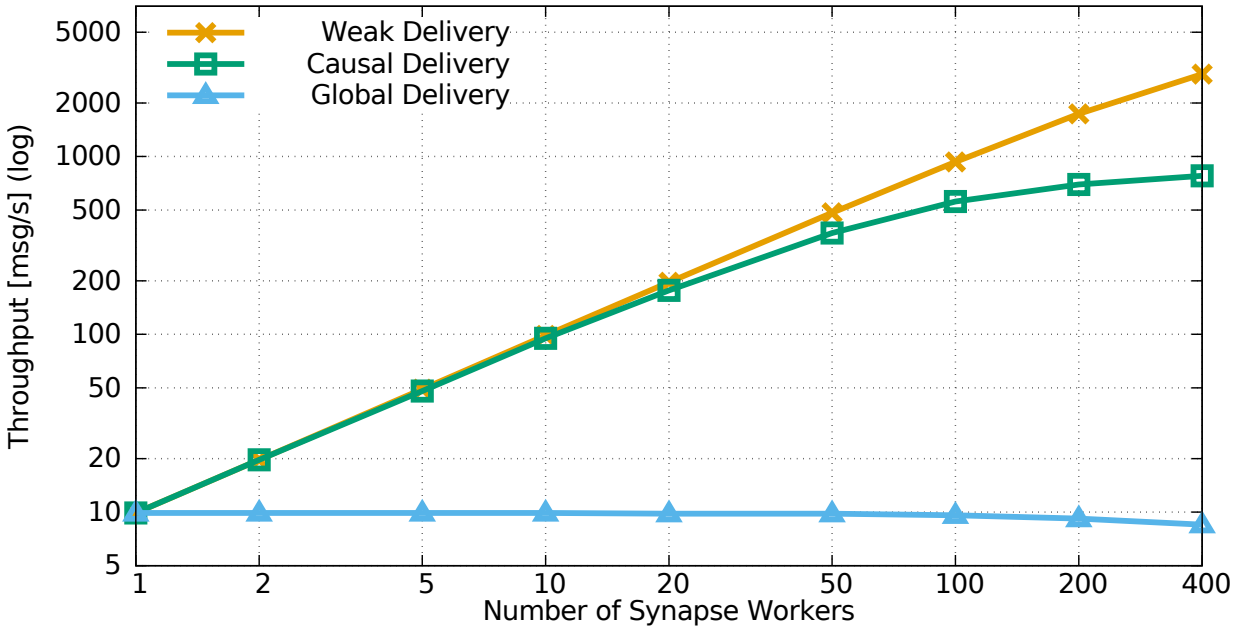


Figure 5.18: **Delivery Performance.** Subscribers are running a 100ms callback on each message. Each line represents a different delivery mode.

5.6.4 Delivery Semantic Comparison (Q3)

SYNAPSE supports three delivery modes – global, causal, and weak – which provide different scaling properties. Figure 5.18 compares subscriber scalability with increased number of subscriber workers available to process writes in parallel. We configure subscribers with a 100-ms callback delay to simulate heavy processing, such as sending emails. Each line shows a different delivery mode, for which we both configure the publisher and the subscriber to operate under that delivery mode. The global delivery mode, which requires the subscriber to commit each write serially, scales poorly. The causal delivery mode, which only requires the subscriber to serialize dependent updates, provides much better scalability. Its peak throughput is limited by the inherent parallelism of the workload. Finally, the weak delivery mode scales perfectly, never reaching its peak up to 400 subscriber workers. In practice, we recommend choosing causal for the publisher and either the causal or weak mode for subscribers.

5.6.5 Production Notes (Q4)

Crowdtap has given us very positive feedback on SYNAPSE's usability and value, including the following interesting stories from their use of SYNAPSE in production.

Supports Heavy Refactoring: Crowdtap discovered a new use for SYNAPSE that we had not anticipated: implementing live DB migrations. Unhappy with MongoDB's performance, they migrated their Main App to TokuMX, another document-oriented DB. To do so, they bootstrapped a subscriber app implementing the same functionality as the original app but running on TokuMX. The subscriber registered for all the Main App's data. Once it was up to date, developers just switched their load balancer to the new application and the migration was completed with little downtime. They also applied this mechanism to address otherwise difficult schema migration challenges. For example, after performing a heavy refactor on one of their services, instead of updating the deployed service, they deployed the new version as a different service with its own DB to ensure that everything was running as expected. For some period of time, the two different versions of the same service run simultaneously, enabling the team to perform QA on the new version, while keeping the possibility to rollback to the old version if needed. This mechanism allow no downtime procedures.

Supports Agile Development: A key aspect in a startup company is agility. New features must be rolled out quickly and securely evaluated. According to Crowdtap, SYNAPSE helps with that. One developer said: "It allows us to be very agile. We can experiment with new features, with real data coming from production." For example, during a hackathon, one of the developers implemented a new reporting prototype. He was able to subscribe to real time production data without impacting the rest of the system thanks to SYNAPSE's isolation properties. The business team immediately adopted this reporting tool, and has been using it ever since.

Flexible Semantic Matters: Interestingly, Crowdtap initially configured all of its services to run in causal mode. However, during an upgrade of RabbitMQ, the (otherwise reliable) message queuing system that SYNAPSE relies upon, some updates were lost due to an upgrade failure. Two subscribers deadlocked, and their queues were filling up, since they were missing dependencies and could not consume the updates. After timeouts, SYNAPSE's recovery mechanisms, which rebootstrap the subscribers, kicked in and the system was unblocked. However, the subscriber apps were unavailable for a long period of time. Crowdtap now chooses between causal and weak delivery modes for each of its subscribers, taking into account its availability/consistency needs. It is not an easy choice, but it *can* and *must* be done in a production environment

where even reliable components can fail. We recommend other engineers implementing causal systems to make message loss recovery an integral part of their system design, specifically to avoid human intervention during failures. Ideally, subscribers would operate in causal mode, with a mechanism to give up on waiting for late (or lost) messages, with a configurable timeout. Given these semantics, SYNAPSE's weak and causal modes are achieved with the timeout set to $0s$ and ∞, respectively.

5.7 Related Work

SYNAPSE builds on prior work in DB replication, data warehousing, federated DBs, publish/subscribe systems, and consistency models. We adopt various techniques from these areas, but instantiate them in unique ways for the domain of modern MVC-based applications. This lets us break through challenges incurred by more general prior approaches, and design the first real-time service integration system that supports heterogeneous DBs with strong delivery semantics.

Same-DB Replication. The vast majority of work in DB replication, as surveyed in [26], involves replicating data across different instances of *the same DB engine* to increase the DB's availability, reliability, or throughput [33]. Traditional DB replication systems plug in at low levels [26], which makes them DB specific: e.g., they intercept updates inside their engines (e.g., Postgres replication [87]), between the DB driver and the DB engine (e.g., MySQL replication [80]), or at the driver level (e.g., Middle-R [83]). SYNAPSE operates at a much higher level – the ORM – keeping it largely independent of the DB.

Data Warehousing and Change Capture Systems. Data warehousing is a traditional approach for replicating data across heterogeneous DBs [40; 28]. While many warehousing techniques [27; 53; 126; 118; 122; 105] are not suitable for real-time integration across SQL and NoSQL engines. Replication is usually implemented either by installing triggers that update data in other DBs upon local updates, or by tailing the transaction log and replaying it on other DBs, as LinkedIn's Databus does [36]. Although transaction logs are often available, it is generally agreed that parsing these logs is extremely fragile since the logs are proprietary and not guaranteed to be stable across version updates. SYNAPSE differs from all of these systems by replicating at the level of ORMs, a much more generic and stable layer, which lets it replicate data between both SQL and NoSQL engines.

In general, existing systems for replicating between SQL and NoSQL DBs, such as MoSQL [105], or MongoRiver [122] work between only specific pairs of DBs, and offer different programming abstractions

and semantics. In contrast, SYNAPSE provides a unified framework for integrating heterogeneous DB in realtime.

DB Federation. The DB community has long studied the general topic of integrating data from different DBs into one application, a topic generally known as DB federation [91]. Like SYNAPSE, federation systems establish a translation layer between the different DBs, and typically rely on DB views – materialized or not – to perform translations. Some systems even leverage ORMs to achieve uniform access to heterogeneous DBs [13]. However, these systems are fundamentally different from SYNAPSE: they let the *same* application access data stored in different DBs uniformly, whereas SYNAPSE lets *different* applications (subscribers) replicate data from one DB (the publisher). Such replication, inspired by service-oriented architectures, promotes isolation and lets the subscribers use the best types of DBs, indexes, and layouts that are optimal for each case.

Similar to DB federation is projects that aim to create "universal" ORMs, which definite a common interface to all DBs (SQL or otherwise), such as Hibernate [92], DataMapper [58] and CaminteJS [51]. Such ORMs should in theory ease development of an application that accesses data across different DBs, a problem complementary to that which SYNAPSE solves. However, since they expose a purely generic interface, such an ORM will encourage a design that does not cater to the individual features provided by each DB. In contrast, SYNAPSE encourages developers to use different ORMs for different sorts of DBs, providing a common programming abstraction to replicate the data across them.

Publish/Subscribe Systems. SYNAPSE's API is inspired by publish/subscribe systems [25; 94; 4; 104; 85]. These systems require programmers to specify which messages should be included in which unit of order, while SYNAPSE *transparently* intercepts data updates, compiles their dependencies automatically, and publishes them.

5.8 Summary

SYNAPSE is an easy-to-use framework for structuring complex, heterogeneous-database Web applications into ecosystems of microservices that integrate data from one another through clean APIs. SYNAPSE builds upon commonly used abstractions provided by MVC Web frameworks, such as Ruby-on-Rails, Python Django, or PHP Symfony. It leverages models and ORMs to perform data integration at data object level, which provides a level of compatibility between both SQL and NoSQL DBs. It leverages

controllers to support application-specific consistency semantics without sacrificing scalability. We have implemented SYNAPSE for Ruby-on-Rails, shown that it provides good performance and scalability, released it on GitHub [116], and deployed it in production to run the microservices for a company. Under the cover, SYNAPSE embeds a record-replay engine that implement a partial ordering mechanism akin to SCRIBE's rendezvous points. We have shown many similarities between the SCRIBE record-replay engine and SYNAPSE. Due to these similarities, we expect record-replay applications such as RACEPRO and DORA to be implementable for distributed applications using SYNAPSE. We discuss such possibilities in the next chapter.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We presented four different record-replay systems to achieve different goals. We introduced SCRIBE, an OS-level deterministic execution record-replay system to capture and reproduce hard to find bugs; RACEPRO, a process-race detection system to improve software correctness in deployed systems; DORA, a mutable replay system to reproduce nondeterministic bugs with retroactive debugging; and SYNAPSE, a replication system for distributed applications to facilitate the use of heterogeneous databases in service-oriented architectures. Throughout each system we presented, we gradually moved away from deterministic replay, opening the range of use cases.

First, we presented SCRIBE, the first operating system mechanism to provide transparent, deterministic execution record and replay of multi-threaded and multi-process applications on commodity multiprocessors and operating systems. SCRIBE records and replays multiple processes by accounting for nondeterministic interactions among processes and their execution environment. SCRIBE introduces *rendezvous points* to ensure correct partial ordering of execution based on system call dependencies, and *sync points* to convert asynchronous interactions that can occur at arbitrary times into synchronous events that are much easier to record and replay. SCRIBE can transition an application to running live at any time, and use checkpoints to record and replay from any point in time. We have implemented SCRIBE without changing, relinking, or recompiling applications, libraries, or operating system kernels, and without any specialized hardware support. It works on commodity Linux operating systems, and commodity multi-core and multiprocessor hardware. Our evaluation shows for the first time that an operating system mechanism can correctly

and transparently record and replay multi-process and multi-threaded applications on multiprocessors. The evaluation also provides strong empirical evidence that real server and desktop applications perform frequent operating system activities which can serve as sync points. SCRIBE recording overhead is modest for server applications including Apache and MySQL, and for desktop applications including Firefox, Acrobat, OpenOffice, parallel kernel compilation, and movie playback.

Second, we have presented the first study of real process races, and the first system, RACEPRO, for effectively detecting process races beyond TOCTOU and signal races. Our study has shown that process races are numerous, elusive, and a real threat. To address this problem, RACEPRO automatically detects process races, checking deployed systems in vivo by recording live executions and then checking them later. It thus increases checking coverage beyond the configurations or executions covered by software vendors or beta testing sites. First, RACEPRO records executions of multiple processes while tracking accesses to shared kernel resources via system calls. Second, it detects process races by modeling recorded system calls as load and store micro-operations to shared resources and leveraging existing memory race detection algorithms. Third, for each detected race, it modifies the original recorded execution to reproduce the race by changing the order of system calls involved in the races. It replays the modified recording up to the race, allows it to resume live execution, and checks for failures to determine if the race is harmful. RACEPRO heavily relies on record-replay mechanisms to perform its tasks. Its engine extends SCRIBE's to allow specific modifications at the end of a recorded log file. We have implemented RACEPRO, shown that it has low recording overhead so that it can be used with minimal impact on deployed systems, and used it with real applications to effectively detect 10 process races, including several previously unknown bugs in shells, databases, and makefiles. While RACEPRO's engine supports minimal modifications at the end of a recorded log file, it must go live once these modifications are executed during the replay due to divergence issues. We explored how to alleviate these issues with DORA.

Third, we introduced the concept of mutable replay, and the first transparent mutable record-replay system with DORA, allowing a recorded execution of an application to be replayed with a modified version of the application. This feature, not available in previous record-replay systems, enables powerful new functionality. In particular, DORA can help reproduce, diagnose, and fix software bugs by replaying a version of a recorded application that is recompiled with debugging information, reconfigured to produce verbose log output, modified to include additional print statements, or patched to fix a bug. This is made possible by the use of lightweight operating system mechanisms to record and replay without imposing unnecessary timing

and ordering constraints. DORA introduces an explorer that directs the replay mechanism to identify a mutable replay of the modified application that minimizes differences with the original unmodified application execution. DORA's feature set is a superset of SCRIBE's. When the replayed application has not been modified, DORA behaves similarly to SCRIBE. Thus, mutable replay supports at least all use cases deterministic replay offers. To implement mutable replay, DORA extends RACEPRO's record-replay engine that paved the way to enable modifications in a recorded execution. Our experimental results on a Linux prototype demonstrate that mutable replay is feasible across a wide range of real-world applications and application changes which can reach thousands of lines of code, even without support for major changes to core application semantics. We show that mutable replay is useful for enabling common debugging techniques not possible with previous record-replay systems. We also show that mutable replay enables validation of security patches against both exploits and production workloads. This is all accomplished without requiring source code modifications and with low recording overhead, enabling usage on production systems. These results demonstrate that mutable replay has the potential to enable new techniques for debugging and patch testing and validation, which can lead to substantial improvements in software reliability and developer productivity. Until this point, we only explored OS-level record-replay mechanisms. These mechanisms have limited support for distributed applications such as web applications.

Lastly, we presented SYNAPSE, an heterogeneous database replication system specifically designed for web applications in a service-oriented architecture. These applications run on top of their own databases, whose layouts, and engines can be completely different, and incorporate read-only views of each others' shared data. SYNAPSE synchronizes these views in real-time using a new scalable, consistent replication mechanism that leverages the high-level data models in popular MVC-based Web applications to replicate data across heterogeneous databases. This replication is performed with record-replay techniques similar to the ones introduced with our OS-level record-replay engines. Execution is partially ordered by DB object access sequence numbers, in a similar way our OS-level record-replay engines order OS resources accesses. SYNAPSE leverages models and ORMs to perform data integration at data object level, which provides a level of compatibility between both SQL and NoSQL DBs. It leverages controllers to support application-specific consistency semantics without sacrificing scalability. We have implemented SYNAPSE for Ruby-on-Rails, shown that it provides good performance and scalability, and deployed it in production for a company.

We have shown four systems implementing transparent application execution record-replay to achieve various use cases. These four systems are presented in an order where replay semantics are increasingly

broaden to enable new use cases. With SCRIBE, we explored an implementation of deterministic record-replay where the replayed execution appears to be identical to the original one from an application perspective, but in reality, differs due to differences in scheduling to improve performance. With RACEPRO, we showed how to apply minor changes at the end of a recorded execution and replay such changes before going live. With DORA, we introduced mutable replay which allow a modified version of the same application to be replayed. With SYNAPSE, we achieved heterogeneous database replication with distributed record-replay mechanisms implemented in web frameworks. Our findings showcase numerous use cases of record-replay, ranging from (1) transparent fault tolerance by recording a primary and replaying on replicas, (2) dynamic application analysis by performing costly instrumentation on replicas that replay application behavior recorded on production systems, such as process-race detection, (3) debugging applications by capturing hard-to-find bugs and reproducing them with recompiled and reconfigured version of applications to greatly facilitate debugging, and (4) heterogeneous database replication. We gradually broaden the replay semantics throughout our systems to show

This dissertation has shown how to make deterministic record-replay fast and efficient, how broadening replay semantics can enable powerful new use cases, and how choosing the right level of abstraction for record-replay can support distributed and heterogeneous database replication with little effort.

6.2 Future Work

The work developed in this dissertation raises the possibility for a number of improvements and challenging questions to consider for future research directions. While SCRIBE shows excellent recording overhead in many applications, it can exhibit high overhead with applications where multiple threads write at a very high rate to the same memory page, causing page ownership to ping-pong between threads degrading performance significantly. Page ownership ping-ponging may also occur due to false data sharing among threads. For example, if two private objects of two different threads are placed on the same page, SCRIBE would serialize accesses to these objects even though these objects are private to each thread. To alleviate this overhead issue, applications could cooperate with SCRIBE: instead of recording page access ordering transparently at the kernel level, applications could inform the recording engine about which objects are accessed. This cooperation can be done in two ways. With the first one, SCRIBE still performs memory tracking as usual, except that the recorded application is modified in a way to acquire and release pages ownership in

specific hot code paths. In addition, the application would have to store unrelated objects on different pages to avoid false sharing effects. This method is tolerant to application bugs as the kernel would still serialize all memory accesses even when hints given by the applications are incorrect. The second method is to offload the serialization of all object accesses to the application Since the application should already have serialization primitives to protect multiple writers accessing the same object, this mechanism is analogous to SCRIBE's rendezvous points, but operating with the application objects as opposed to kernel objects. This mechanism would thus have a very modest overhead, despite high contention, and have no false sharing issues. However, if developers misuse these user-space rendezvous points, SCRIBE would no longer be able to guarantee deterministic replay, as opposed to the first method. This second method is thus more suitable to implement in runtime environments, such as the Java Virtual Machine (JVM). In these cases, assuming the runtime environment is correctly implemented, SCRIBE would be able to guarantee deterministic replay with a much lower overhead.

As an increasing number of applications are using higher level languages running atop of virtual machines, such as the JVM, the Ruby runtime, Python's interpreter, the V8 engine for JavaScript to name a few, it becomes natural to offload SCRIBE's recording engine of certain tasks. As discussed earlier, using user-space rendezvous points is beneficial for performance, but would also allow better application semantics extraction for use cases such as RACEPRO or DORA. Other tasks can be offloaded, such as the interaction tracking of the runtime environment with the kernel. For example, certain user-space APIs could cooperate with SCRIBE to disable kernel-level recording of certain system calls, while performing its own recording. For example, when performing an HTTP request, only the request headers and body could be recorded, as opposed to recording the low-level interaction with the underlying sockets. In this case, achieving deterministic replay is still guaranteed while the recorded log no longer needs to contain the information of doing DNS lookups, whether a keep-alive connection was used, if the connection was redirected through 302 status codes, or if the stream was encrypted through SSL. This hybrid approach can be done in many ways as user-space applications often have deep call stacks spawning multiple libraries. For example, in Ruby, a call to `Net::HTTP.get(URI("www.google.com"))` invokes a few libraries written in Ruby, invoking the Ruby runtime, then libresolv, libnss, zlib, openssl, which in turn rely on libc to finally invoke around 50 system calls. Using a hybrid approach is versatile as one could instrument a specific set of APIs while retaining the original SCRIBE behavior for other unrelated APIs. This way, deterministic recording guarantees can still be preserved, even when a high-level application uses a non-instrumented C extension. Tools that lever-

age record-replay mechanisms such as DORA and RACEPRO can benefit hugely from these higher-level of recording abstractions as the recorded log contains a more precise representation of application semantics.

RACEPRO benefits directly from this hybrid approach. Since RACEPRO's race detection engine feeds directly from recorded object accesses via rendezvous points, using user-space rendezvous points would allow RACEPRO to detect not only process races, but also application races. It would be able to do so much faster as the recorded log would be smaller and contain a lot fewer rendezvous points.

DORA is currently limited as it explores program execution by removing and adding system calls. Leveraging application semantics allows much better exploration. For example, consider a recorded application that performs two HTTP GET requests that are sharing the same socket through HTTP pipelining. When operating at the kernel-level, DORA is not be able to find a good mutable replay as it would have to understand the underlying HTTP protocol to discard the appropriate data from the received buffer on the corresponding socket, which may be infeasible when using SSL connections. On the other hand, when recording at the HTTP API level, DORA would be able to just simply discard that one GET request from the recording to find the optimal replay. By using a hybrid recording approach, DORA not only would be able to find better mutable replays, but would do so much quicker as there are less nodes to explore in the execution graph.

Additionally, both RACEPRO and DORA would benefit from a in-memory cloning mechanism applied to an entire process container, with copy-on-write optimizations. Indeed, both engines replay many executions that are identical from their start up to a certain point. A lot of CPU is wasted replaying many times the same execution up to that point. For example, when RACEPRO explores a race after that point, it has to replay an entire execution up to that point, and then to the race to be analyzed. Having checkpoints to replay from during the original execution would greatly improve performance as the majority of the CPU is wasted replaying the beginning of the execution. Similarly, DORA replays entire executions, while it would be much more efficient to replay from the point of divergence. Implementing an in-memory cloning mechanism entails a substantial amount of work as many kernel objects would have to implement a copy-on-write mechanism.

Earlier, we introduced SYNAPSE which does record-replay in a distributed environment. While SYNAPSE records all application state modifications by interposing on the database layer, it does not record enough information to deterministically replay an execution. For example, SYNAPSE does not record any of the incoming HTTP requests made to the application. Together with a hybrid record-replay implementation of SCRIBE, a useful distributed deterministic record-replay system can be built. While previous work has

explored the concept of distributed deterministic record-replay, it has failed to provide useful semantics to build higher level tools. For example, DDOS [55] records information at the packet level, and is not suitable for high level frameworks used in web applications. With a hybrid record-replay system based on SCRIBE and SYNAPSE, we could build a distributed deterministic record-replay system for web applications suitable for implementing higher level mechanisms such as race detection and mutable replay.

This hybrid record-replay system would complement the original premise of SYNAPSE which was to cleanly share data among services. SYNAPSE currently sends application state modifications from publishers to subscribers through a message broker. The new hybrid system could publish additional messages interleaved with the existing state modifications messages, providing necessary information to do deterministic record-replay. For example, messages would include received HTTP requests, or API call responses from third-party services such as payment gateways. Recording data based nondeterminism is easy considering that these data accesses are typically made through well-defined and simple APIs (unlike a UNIX kernel API). However, recording time based nondeterminism is hard. Thankfully, the underlying mechanism of the causal delivery mechanism of SYNAPSE is a mechanism similar to SCRIBE's rendezvous points allowing record-replay. The hybrid record-replay system can record both data- and time-based sources of nondeterminism in an efficient manner, send the recorded log to the message broker alongside with regular SYNAPSE messages, and let subscriber services implement useful tools atop of record-replay.

Being able to subscribe to message streams from publishers that contain enough information to perform deterministic record-replay allows, in addition to all the traditional SYNAPSE use cases, a set of new possibilities. A subscriber service can be deployed to perform race detection in the background with a similar engine to RACEPRO. Race detection can be very useful to save money. For example, Crowdtap suffered from a bug in an open-source application they used for their e-commerce store which allows users to redeem their points for Amazon gift cards. The application was decrementing the inventory item quantity by reading the quantity and writing it back in a read-committed transaction (the default in most SQL databases) as opposed to a serializable transaction. The race manifested in giving away thousands of Amazon gift cards in excess. Having a subscriber service that detects races in the background could be valuable to improve software reliability. Further, a subscriber akin to DORA can be deployed to implement mutable replay. This can be useful to reproduce and debug elusive bugs. Adding logging after-the-fact can be very useful. At Crowdtap, a bug resulted in throwing away acquired gift cards due to a faulty retry loop in the gift card acquisition process. The bug was discovered when the accounting team confronted their Amazon bills with the num-

bers of given gift cards. Sadly, the developer implemented this feature without logging any requests which made the recovery of gift cards difficult. It would have been useful to replay the application with a code modification allowing logging of gift cards. To summarize, useful services can be implemented on top of deterministic record-replay mechanisms for distributed web applications. We discussed race detection and mutable replay, but other use cases could be envisioned, like performing after-the-fact metric extraction, or performance analysis. Further, these added-value services could be implemented by third-party companies which would consume the message stream from the existing SYNAPSE message broker of the application.

In addition to backend record-replay, it would be useful to perform frontend record-replay as an increasing number of web applications are deploying substantial amount of logic in user devices. This avenue has already been explored by Mugshot [70], a lightweight record-replay engine for JavaScript that can run on unmodified browsers. By combining the recording of the client application and the backend application, a developer could replay a user session entirely from the frontend interactions down to the database, including possible recorded races. This would provide the ability to reproduce user behavior on the frontend, introspect interactions with other users through backend services, and provide an accurate time machine for after-the-fact auditing and code modifications.

In our experience, the success of mutable replay heavily relies on the quality of the semantics extracted from the recorded execution. The ability to distil core application state changes depends on the abstraction level at which data is recorded. For example, recovering meaningful application state changes by analyzing a hardware level recording would be near impossible. On the other hand, if the application is written in a way where state changes are explicitly described, usually in the form of immutable domain logic events (e.g. `event(UserSignUp, params)`), mutable replay becomes trivial to perform. An incarnation of this design pattern is event sourcing [47]. In the context of Web applications, Event Store [127] is an example of database with native support for such design pattern. Redux [3] is an example of application state store for frontend applications. The latter is particularly interesting as it already provides mutable replay features allowing hot code reloading and time travel capabilities [2].

To conclude, we only scratched the surface of what we can do with record-replay mechanisms. Applications are increasingly built with high level abstractions, on heterogeneous machines, with complex distributed semantics. Our current set of record-replay tools are incapable of providing value for these applications. We discussed feasible research avenues to build tools suitable for these distributed applications.

Bibliography

[1] Mozilla rr.
http://rr-project.org/. (pages 31, 39).

[2] D. Abramov. Live React: Hot Reloading with Time Travel.
https://www.youtube.com/watch?v=xsSnOQynTHs. (page 159).

[3] D. Abramov. Redux: Predictable State Container for JavaScript Applications.
https://github.com/reactjs/redux. (page 159).

[4] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A Client Notification Service for Internet-Scale Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011. (page 150).

[5] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Oct. 2009. (pages 82, 108).

[6] Apache Bug 53131. https://issues.apache.org/bugzilla/show_bug.cgi?id=53131. (pages 85, 101, 102).

[7] Apache. Hadoop ZooKeeper. http://hadoop.apache.org/zookeeper/. (page 136).

[8] Apache. Kafka – a high-throughput distributed messaging system.
http://kafka.apache.org/. (page 129).

[9] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010. (page 81).

[10] D. F. Bacon and S. C. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991. (page 39).

[11] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The Potential Dangers of Causal Consistency and an Explicit Solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)*, Oct. 2012. (page 133).

[12] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, June 2013. (page 133).

[13] H. Balsters and B. Haarsma. An ORM-Driven Implementation Framework for Database Federations. In R. Meersman, P. Herrero, and T. S. Dillon, editors, *OTM Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 659–670. Springer, Nov. 2009. (page 150).

[14] R. M. Balzer. EXDAMS: Extendable Debugging and Monitoring System. In *Proceedings of the 1969 Spring Joint Computer Conference (AFIPS '69)*, May 1969. (page 39).

[15] R. Barcia, G. Hambrick, K. Brown, R. Peterson, and K. Bhogal. *Persistence in the Enterprise: A Guide to Persistence Technologies*. IBM Press, first edition, May 2008. (pages 113, 115).

[16] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010. (page 81).

[17] P. Bergheaud, D. Subhraveti, and M. Vertes. Fault Tolerance in Multiprocessor Systems Via Application Cloning. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*, June 2007. (pages 32, 40, 41).

[18] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, June 2006. (pages 82, 108, 109).

[19] E. W. Biederman. Multiple Instances of the Global Linux Namespaces. In *Proceedings of the Linux Symposium*, July 2006. (page 93).

[20] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991. (page 122).

[21] M. Bowers. Database Revolution: Old SQL, New SQL, NoSQL... Huh? https://www.marklogic.com/resources/database-revolution-old-sql-new-sql-nosql-huh/resource_download/presentations. (page 112).

[22] T. C. Bressoud. TFT: A Software System for Application-Transparent Fault Tolerance. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing (FTCS '98)*, June 1998. (pages 27, 40, 41).

[23] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995. (pages 27, 39, 41).

[24] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Nov. 2006. (page 136).

[25] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Trans. Comput. Syst.*, 19(3), Aug. 2001. (page 150).

[26] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, June 2008. (page 149).

[27] G. K. Y. Chan, Q. Li, and L. Feng. Design and Selection of Materialized Views in a Data Warehousing Environment: A Case Study. In *Proceedings of the 2nd ACM International Workshop on Data Warehousing and OLAP (DOLAP '99)*, Nov. 1999. (page 149).

[28] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1), Mar. 1997. (page 149).

[29] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SIGMETRICS '98)*, June 1998. (page 39).

[30] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of the USENIX 2008 Annual Technical Conference (ATC '08)*, June 2008. (pages 44, 80, 82, 108, 109).

[31] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage Replay with Crosscut. *SIGPLAN Not.*, 45(7):13–24, Mar. 2010. (page 108).

[32] M. Chu, C. Murphy, and G. Kaiser. Distributed In Vivo Testing of Software Applications. In *Proceedings of the 1st IEEE International Conference on Software Testing, Verification, and Validation (ICST '08)*, Apr. 2008. (page 80).

[33] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008. (page 149).

[34] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *Communications of the ACM*, 14(10), Oct. 1971. (pages 21, 41).

[35] 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 (OSDI '10)*, Oct. 2010. (page 43).

[36] S. Das, C. Botev, and K. Surlaker, et.al. All Aboard the Databus! LinkedIn's Scalable Consistent Change Data Capture Platform. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)*, Oct. 2012. (page 149).

[37] The Debian Almquist Shell. `http://gondor.apana.org.au/~herbert/dash/`. (page 50).

[38] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007. (page 135).

[39] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, Mar. 2009. (page 39).

[40] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration.* Morgan Kaufmann, July 2012. (page 149).

[41] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002. (pages 27, 39, 41).

[42] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, Mar. 2008. (pages 12, 27, 39, 41).

[43] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Oct. 2003. (pages 43, 48, 79).

[44] J. Ferris and J. Clayton. Factory Girl. A library for setting up Ruby objects as test data. https://github.com/thoughtbot/factory_girl. (page 137).

[45] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages (POPL '05)*, Jan. 2005. (page 68).

[46] P. Fonseca, C. Li, and R. Rodrigues. Finding Complex Concurrency Bugs in Large Multi-Threaded Applications. In *Proceedings of the 6th ACM European Conference on Computer Systems (EUROSYS '11)*, Apr. 2011. (pages 71, 79).

[47] M. Fowler. Event Sourcing: Capture all changes to an application state as a sequence of events. http://martinfowler.com/eaaDev/EventSourcing.html. (page 159).

[48] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Nov. 2002. (page 115).

[49] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: Towards Manifesting Hidden Concurrency Typestate Bugs. In *Proceedings of the 16th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, Mar. 2011. (page 79).

[50] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the USENIX 2006 Annual Technical Conference (ATC '06)*, June 2006. (pages 17, 39).

[51] A. Gordeyev. Camintejs. http://www.camintejs.com/. (page 150).

[52] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, Dec. 2008. (pages 39, 53, 82, 89, 108).

[53] H. Gupta and I. S. Mumick. Selection of Views to Materialize in a Data Warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):24–43, Jan. 2005. (page 149).

[54] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA '08)*, June 2008. (page 39).

[55] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble. DDOS: Taming Nondeterminism in Distributed Systems. In *Proceedings of the 18th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, Mar. 2013. (page 158).

[56] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, 2005. (pages 82, 109).

[57] I. Kravets and D. Tsafrir. Feasibility of Mutable Replay for Automated Regression Testing of Security Updates. In *the 2012 Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE '12)*, March 2012. (pages 85, 86, 89, 99, 109).

[58] D. Kubb. Datamapper. http://datamapper.org/. (page 150).

[59] O. Laadan, R. A. Baratto, D. Phung, S. Potter, and J. Nieh. DejaView: A Personal Virtual Computer Recorder. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007. (pages 14, 95).

[60] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the USENIX 2007 Annual Technical Conference (ATC '07)*, June 2007. (pages 14, 95).

[61] O. Laadan and J. Nieh. Operating System Virtualization: Practice and Experience. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR '10)*, May 2010. (page 34).

[62] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, 1978. (page 54).

[63] Launchpad Software Collaboration Platform. https://launchpad.net/. (page 43).

[64] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987. (pages 21, 39, 41, 82, 108).

[65] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011. (page 132).

[66] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*, Apr. 2013. (page 132).

[67] 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 Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, Mar. 2008. (pages 49, 81).

[68] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, Oct. 2006. (page 79).

[69] F. Mattern. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 15th Annual Symposium on Principles of Programming Languages (POPL '88)*. Oct. 1988. (pages 55, 65).

[70] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI '10)*, Apr. 2010. (page 159).

[71] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA '08)*, June 2008. (page 39).

[72] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, Mar. 2009. (page 39).

[73] 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 (OSDI '08)*, Dec. 2008. (page 79).

[74] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection For Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI '06)*, June 2006. (page 79).

[75] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *Proceedings of the 12th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, Oct. 2006. (page 39).

[76] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA '05)*, June 2005. (page 39).

[77] 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 Con-*

ference on Programming Language Design and Implementation (PLDI '07), June 2007. (pages 52, 69, 79, 80, 82, 108, 109).

[78] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, Mar. 2008. (page 80).

[79] M. Olszweski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, Mar. 2009. (page 39).

[80] Oracle Corporation. MySQL 5.5 Reference Manual – Replication. http://dev.mysql.com/doc/refman/5.5/en/replication.html. (page 149).

[81] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002. (pages 14, 16, 34, 72, 95).

[82] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Oct. 2009. (pages 82, 108).

[83] M. Patiño Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Trans. Comput. Syst.*, 23(4), Nov. 2005. (page 149).

[84] M. Perham. Sidekiq: Simple, efficient background processing for Ruby. http://sidekiq.org/. (page 133).

[85] P. R. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, July 2002. (page 150).

[86] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating System Transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Oct. 2009. (page 81).

[87] Postgres-R. A Database Replication System for PostgreSQL. http://www.postgres-r.org/. (page 149).

[88] D. P. Quigley, J. Sipek, C. P. Wright, and E. Zadok. UnionFS: User and Community-oriented Development of a Unification Filesystem. In *Proceedings of the 2006 Linux Symposium*, July 2006. (page 67).

[89] RabbitMQ – Messaging that just works. http://www.rabbitmq.com/. (page 129).

[90] Resource races study in RACEPRO. http://rcs.cs.columbia.edu/projects/racepro/. (page 47).

[91] R. Ramakrishnan and J. Gehrke. *Database Management Systems, 3rd Edition*. McGraw-Hill Education, Aug. 2002. (page 150).

[92] Red Hat Community. Hibernate. http://hibernate.org/. (page 150).

[93] Introduction to Redis. http://redis.io/topics/introduction. (pages 135, 142).

[94] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Proceedings of the International COST264 Workshop on Networked Group Communication (NGC '01)*, Nov. 2001. (page 150).

[95] M. Russinovich and B. Cogswell. Replay for Concurrent Non-Deterministic Shared-Memory Applications. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation (PLDI '96)*, May 1996. (page 39).

[96] Y. Saito. Jockey: A User-Space Library for Record-Replay Debugging. In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG'05)*, Sept. 2005. (pages 17, 39, 82, 89, 108).

[97] K. Sen. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, June 2008. (pages 43, 52, 79).

[98] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *Proceedings of the 3rd workshop on on Hot Topics in System Dependability (HotDep'07)*, June 2007. (page 109).

[99] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic Software Self-healing Using REscue points. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, Mar. 2009. (page 110).

[100] J. H. Slye and E. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, June 1996. (pages 27, 41).

[101] Spree. Ecommerce platform. http://spreecommerce.com/. (page 140).

[102] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference (ATC '04)*, June 2004. (pages 17, 40, 53, 82, 108).

[103] D. Stodden, H. Eichner, M. Walter, and C. Trinitis. Hardware Instruction Counting for Log-Based Rollback Recovery on x86-Family Processors. In *Proceedings of the 3rd International Service Availability Symposium (ISAS '06)*, May 2006. (pages 32, 41).

[104] R. Storm, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '98)*, Oct. 1998. (page 150).

[105] Stripe. A MongoDB to SQL Streaming Translator. https://github.com/stripe/mosql. (page 149).

[106] D. Subhraveti and J. Nieh. Record and Transplay: Partial Checkpointing for Replay Debugging Across Heterogeneous Systems. In *Proceedings of the 2011 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS '11)*, June 2011. (pages 82, 108).

[107] C. Tang. DSF: A Common Platform for Distributed Systems Research and Development. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware '09)*, Nov. 2009. (page 109).

[108] Textalytics. Web Services for Text Analysis and Mining. https://textalytics.com/. (page 141).

[109] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *Proceedings of the 12th Euromicro Conference on Real-Time System (Euromicro-RTS'00)*, June 2000. (page 40).

[110] *The Guinness Book of World Records*. Sterling Publishing Co., Inc, 1985. (page 34).

[111] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, Feb. 2008. (page 80).

[112] E. Tsyrklevich and B. Yee. Dynamic Detection and Prevention of Race Conditions in File Accesses. In *Proceedings of the 12th Conference on USENIX Security Symposium (SSYM '03)*, Aug. 2003. (page 80).

[113] J. Tucek, W. Xiong, and Y. Zhou. Efficient Online Validation with Delta Execution. *SIGPLAN Not.*, 44(3):193–204, Mar. 2009. (pages 85, 86, 109).

[114] University of California at Berkeley. Open-Source Software for Volunteer Computing and Grid Computing. http://boinc.berkeley.edu/. (page 72).

[115] Upstart: An Event-Based Replacement for System V Init Scripts. http://upstart.ubuntu.com/. (page 47).

[116] N. Viennot. Synapse Sources. https://github.com/nviennot/synapse. (pages 113, 151).

[117] VMware, Inc. http://www.vmware.com/. (page 39).

[118] K. Wang. A DynamoDB to Elasticsearch Translator. https://github.com/kzwang/elasticsearch-river-dynamodb. (page 149).

[119] V. Weaver, D. Terpstra, and S. Moore. Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2013)*, Apr. 2013. (page 31).

[120] J. Wei and C. Pu. TOCTTOU Vulnerabilities in UNIX-Style File Systems: an Anatomical Study. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, Dec. 2005. (page 80).

[121] Wikipedia. A Java message oriented middleware API for sending messages between two or more clients.
http://en.wikipedia.org/wiki/Java_Message_Service. (page 129).

[122] R. Willy. A MongoDB to Elasticsearch Translator.
https://github.com/richardwilly98/elasticsearch-river-mongodb/. (page 149).

[123] J. Wu, 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 (OSDI '10)*, Oct. 2010. (pages 43, 79).

[124] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA '03)*, June 2003. (page 39).

[125] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Apr. 2009. (page 80).

[126] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*, Aug. 1997. (page 149).

[127] G. Young. Event Store.
https://geteventstore.com/. (page 159).

[128] 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 (SOSP '05)*, Oct. 2005. (pages 43, 48, 79).

[129] M. Zalewski. Delivering Signals for Fun and Profit. Bindview Corporation, 2001. (page 43).

[130] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *Proceedings of the 16th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, Mar. 2011. (page 79).