DETERMINISTIC SYSTEMS ANALYSIS

by

Anton Burtsev

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2013

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Anton Burtsev

has been approved by the following supervisory committee members:

John Regehr	, Chair	March 14, 2013 Date Approved
Eric Eide	, Member	March 13, 2013 Date Approved
Bryan Ford	, Member	April xx, 2013 Date Approved
Matthew Might	, Member	March 13, 2013 Date Approved
Andrew Warfield	, Member	April 3, 2013 Date Approved

and by Alan Davis , Chair of the School of Computing

and by Donna M. White, Dean of The Graduate School.

ABSTRACT

A modern software system is a composition of parts that are themselves highly complex: operating systems, middleware, libraries, servers, and so on. In principle, compositionality of interfaces means that we can understand any given module independently of the internal workings of other parts. In practice, however, abstractions are leaky, and with every generation, modern software systems grow in complexity. Traditional ways of understanding failures, explaining anomalous executions, and analyzing performance are reaching their limits in the face of emergent behavior, unrepeatability, cross-component execution, software aging, and adversarial changes to the system at run time.

Deterministic systems analysis has a potential to change the way we analyze and debug software systems. Recorded once, the execution of the system becomes an independent artifact, which can be analyzed offline. The availability of the complete system state, the guaranteed behavior of re-execution, and the absence of limitations on the run-time complexity of analysis collectively enable the deep, iterative, and automatic exploration of the dynamic properties of the system.

This work creates a foundation for making deterministic replay a ubiquitous system analysis tool. It defines design and engineering principles for building fast and practical replay machines capable of capturing complete execution of the entire operating system with an overhead of several percents, on a realistic workload, and with minimal installation costs. To enable an intuitive interface of constructing replay analysis tools, this work implements a powerful virtual machine introspection layer that enables an analysis algorithm to be programmed against the state of the recorded system through familiar terms of source-level variable and type names. To support performance analysis, the replay engine provides a faithful performance model of the original execution during replay.

For my mother Lyudmila.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

First of all, this work would not be possible without the vision and inspiration of Jay Lepreau. Jay created the original idea of applying deterministic replay to analysis of security attacks, convinced me to join the University of Utah, and played a significant role in establishing me as a researcher. Jay changed me with his sharpness, humor, generosity, and the ability to get straight to the point: "*One makes you fat, another gives you cancer*". I do not drink soda, but I will always remember you Jay.

Equally important, this work would not have happened without the Flux Research Group. Eric Eide, David Johnson, Rob Ricci, Mike Hibler, and many other staff members and students were responsible for creating an environment which ensured my research progress. For many years, Eric provided his feedback on various research ideas and multiple aspects of my work. Eric Eide helped me to become a better writer by thoroughly editing my first manuscripts. Mike Hibler inspired me with his endless patience, and his true excitement to get through the hardest parts of the systems stack. Rob Ricci made it possible for me to co-lead and co-advise several students. The work with the students was among the brightest impressions from my graduate years.

Multiple people contributed to this dissertation. David Johnson committed more than a year of his time to work on the virtual machine introspection engine (VMI). At a high level, his work is described in Chapter 8; however, the devil is in the details, which are beyond the scope of this dissertation. David is responsible for understanding, developing, and supporting an infinite number of corner cases of the DWARF debugging format. His contribution made this dissertation possible as an analysis framework.

Chung Hwan Kim was the first to start working on the VMI engine. Chung and David expanded this early work to implement a number of building blocks for the security analysis framework, which is presented as one of the case studies in Chapter 10.

Being a passionate operating systems hacker, Mike Hibler could not refrain from fixing

several errors in the Xen shadow paging and Xenaccess page walking code. Whatever takes Mike a couple of days would take everyone else several weeks. In multiple ways, Mike's expertise helped me to move forward much more efficiently. Also, Mike was my gateway to unlimited hardware resources for this work – servers, disks, CPUs.

Kirk Webb contributed to this dissertation by setting up the Phoronix benchmark suite on a very short deadline.

While finishing this dissertation, I have spent several months working closely with my advisor, John Regehr. This relatively short period of time became one of the most exciting and productive periods of my graduate work. John provided brilliant insights and late-night teamwork on often unrealistic deadlines. I really enjoyed the rapid pace and commitment with which we moved forward.

Many people provided me with their help, love, and support on my long path to completing this work. My mother and my grandparents were shielding me from a variety of problems during the years I have spent in various schools. Their support was crucial for my freedom to concentrate on challenging tasks, which were much more exciting than a developer's job. My wife, Anna, turned my years at the University of Utah into a thrilling adventure. Being a graduate student and then a research scientist, Anna supported my passion to continue this work, even when it looked more like a death march. On multiple deadlines Anna was completely taking over our kids, letting me disappear in the lab. Jon Rafkind, Jennifer Ichida, and Jonathan Babcock became my close friends, who with their sharp humor, help, and support helped me to feel home in Utah. I am thankful for their insights into many aspects of the social and cultural life in America.

Finally, I would like to thank my committee members for agreeing to participate in this effort. I truly appreciate their diverse opinions, unbiased criticism, and insightful suggestions.

The research presented in this dissertation was supported by the Air Force Research Laboratory and DARPA under Contract No. FA8750–10–C–0242, and by the National Science Foundation under Grant No. 0524096. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 1

INTRODUCTION

Reasoning about complex software systems is hard. Composed from a number of relatively simple components, large systems promptly evolve into complex software artifacts with numerous cross-component interactions, unrepeatable run-time conditions, phenomena of software aging, emergent behavior, and a risk to completely change the intended meaning of their code under adversary attacks [125, 71]. Even if execution of individual components can be monitored and explained, the analysis of the system overall requires complex reasoning about its multiple parts, control and data flows across multiple system components, and repetability of run-time conditions. As commodity systems grow in complexity, the lack of support for many kinds of dynamic analyses – debugging, performance analysis, security analysis – becomes clear.

Two inherent problems traditionally complicate debugging of complex software – *irreproducibility* of the system's behavior and *observability* of the system's state. In addition to errors found in sequential programs [46], complex systems commonly experience faults related to schedule interleavings, deadlocks, livelocks, race conditions, response timing, and preceedence violations. Traditional debugging problems are amplified by a new factor – *scale* of the system. Modern systems scale in execution time, number of execution components, the amount of cross-component communication, and a variety of software technologies used for their implementation.

Traditional debugging and dynamic analysis approaches turn out to be inefficient for analysis of complex software systems. Reproduction of rare nondeterministic errors, if possible, requires multiple debugging runs and many hours of execution. Lack of a globally observable state complicates fault analysis as it is often impossible to obtain a truthful view of events in a system. An attempt to inspect a system with a debugger often changes execution of a system and suppresses the error (a phenomenon known as "heisenbug" [82],

and probe [74] effects). Crash dump analysis of faults is further complicated by stampede and bystander effects, i.e., destruction of the system's state right before the crash, and obfuscation of a real cause of the error in a chain of cross-component interactions. A large number of system components and the amount of cross-component interactions further restrain applicability of traditional human-centric debugging practices.

Designed to deliver the highest performance, modern systems utilize multiple forms of parallelism and a number of engineering optimizations that include asynchronous and preemptive execution, multiple layers of request buffering and data caching, multiple threads and contexts of execution, complex communication and synchronization patterns, and so on. The performance of such systems is largely determined by the availability of data, latency of communication, delay-free synchronization, and efficiency of scheduling algorithms. Even if performance of individual components can be understood and profiled, the analysis of a system overall requires complex reasoning about multiple parts of the system, control, and data flow between them, performance of individual requests, dynamic state of buffers and caches, availability of individual resources for parallel and pipelined execution, and delays of communication and synchronization mechanisms.

Traditionally, performance tools balance complexity of analysis and run-time overhead by splitting analysis into two stages: *collecting* and *processing*. During the first stage, analysis tools collect a minimal set of performance metrics and some relevant run-time state of a system. Collected data are then fed as input to the processing stage, which runs off-line [146, 21, 3, 14, 114]. Off-line execution of an analysis introduces a gap between the analysis and the state of the system. Having captured only partial information about the system, the analysis has no means to reason about collected performance metrics with respect to actual run-time state of the system. Performance analysis degrades to performance debugging – a time-consuming, iterative, "measure and modify" procedure aimed at tuning the system [54]. Analysis is used to provide hints about performance bottlenecks, but not a general understanding of system's performance.

Despite a number of radical changes in how computer systems are used, the security model of a modern operating system is based on principles laid down by the first time-sharing computing environments four decades ago. Security attacks against these early

systems were a pastime for small communities of hobbyists. Today, the systems based on essentially the same security principles are required to operate in the face of targeted security attacks sponsored by a multinational malware economy, commercial espionage, and government intelligence agencies. Attacks on computer systems have undergone major changes in their exploit discovery tools, attack complexity, and targeted parts of the system. Attackers routinely use fuzzing tools [126, 127], and attack every possible layer of a computer system: application, operating system, device drivers, operating system services (e.g., network stack, stack of USB protocols, file systems), and so on. A sophisticated attack can exploit 6-10 different vulnerabilities, in a chain of up to 10 different stages [80]. Analysis of unknown attacks requires a set of new automated tools, which support determinism of reexecution, scriptable introspection mechanisms, and aid human centric efforts with an automatic support for program slicing.

1.1 Limitations of traditional dynamic analysis

Traditional approaches to dynamic analysis are inherently limited in their ability to scale and provide support for analysis of such complex multicomponent, multilayer systems. A dynamic analysis algorithm infers general properties of a system by observing concrete instances of its execution. An analysis runs concurrently with the system to monitor its execution and observe transitions in the system's state. An analysis either performs some computation on the system's state, or collects a small subset to the system's state for the off-line processing. A variety of techniques ranging from the manual source code instrumentation to an automatic generation of the dynamic binary instrumentation derived from a high-level SQL language [122, 79] rely on this model for structuring the dynamic analysis algorithms.

The above approach works if two conditions are true: the subset of the system state is small, and it is known in advance. If one of the conditions is violated, i.e., either the analysis requires an intrusively large subset of the system's state, or it is impossible to guess the subset of required information before instrumenting and running the system, the dynamic analysis becomes impossible. Unavailability of the complete state of the system, and its execution history are the main factors limiting the application of dynamic analysis

techniques to the complex systems.

A number of existing approaches try to address the problem of an incomplete system state by invoking lightweight parts of analysis at runtime [31, 144], defining a better subset of the collected data [14], and recreating correlation between observed execution run and semantics of a system by means of the statistical analysis [3, 14, 114]. While reducing the semantic gap between analysis and the system, these approaches still fail to provide a way to build precise reasoning about the system.

As a result, a range of dynamic analyses that inherently rely on interpreting the fine-grain run-time state of a system are prohibited in a production environment due to a multi-factor overhead [120, 130]. Filtering transient anomalies from the execution, statistical methods limit analyses to the reasoning about dominant faults. Furthermore, trying to better correlate execution anomalies with the execution, most analyses rely on some sort of a workload synthesis, which exercises different system properties in isolation. The complexity of the statistical methods and workload synthesis becomes unmanageable if the execution anomaly is determined by multiple overlapping factors.

1.2 Deterministic systems analysis

An alternative approach to performing dynamic analysis is to record execution of the system entirely, and run the analysis algorithm over the recorded execution history off-line. Availability of the complete state and execution history of the system for a dynamic analysis algorithm has a potential to change the way we reason about complex software systems. A complete state, determinism of re-execution, and absence of the limitation on the run-time complexity of the dynamic analysis algorithm enable the thorough, iterative exploration of the system's behavior with the help of automatic analysis algorithms.

This dissertation overcomes limitations of the traditional dynamic analysis approaches by decoupling analysis from execution of the system, while still providing the analysis algorithm with the access to complete execution history, and the entire state of the system (Figure 1.1). A recording layer captures the complete execution history of the system. The analysis algorithm runs against a copy of the execution recreated by the deterministic replay mechanisms off-line. Determinism guarantees that the execution histories of the original

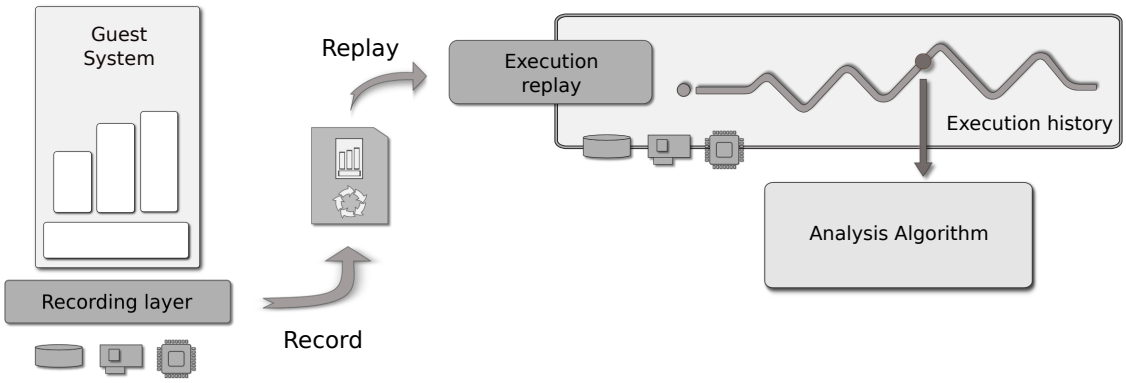

Figure 1.1. Decoupling dynamic analysis from execution.

and replayed executions are instruction-level identical.

Deterministic systems analysis provides a solution for problems, which inherently require a backward reasoning over the execution history of the system. For example, software debugging, which fundamentally is a task of constructing a backward slice of the execution – discovery of a chain of run-time conditions, which eventually force the system to take a faulty path. Determinism of replay provides a debugging algorithm with a way to drive its logic backward in the execution history, to iteratively recreate the backward slice. Deterministic backward reasoning creates a novel opportunity for the development of automatic means of software debugging. In a similar way, deterministic systems analysis can be applied to the explanation of multistage security attacks.

For the problems which require forward reasoning, deterministic replay provides an environment in which a constant, relatively low-overhead recording of the execution is decoupled from a computationally intensive analysis. Such decoupling enables development of the analysis algorithms which require access to the state of the system after every executed instruction, e.g., reconstruction of the data-, and control-flows, control flow integrity checks, data structure integrity checks, generation of path constrains, etc.

1.3 Thesis statement

The research thesis of this work is:

> *It is possible to enable deterministic systems analysis as a practical, ubiquitous mechanism for the deep, repeatable, and computationally unrestricted exploration of the dynamic properties of complex software systems through the application of novel, full-system, deterministic replay mechanisms, and by providing a rich, convenient introspection interface to the execution history and state of the system during replay.*

This dissertation is aimed to provide an engineering foundation for development of formal, precise, and computationally intensive methods of dynamic analysis. By providing a way to reason about behavior of a system in a more static manner, we can scale dynamic analysis to the complexity of the modern systems.

1.4 Overview of the suggested approach

To become a practical tool, deterministic replay and analysis mechanisms suggested in this dissertation are required to support analysis of realistic workloads on realistic systems, and require minimal installation costs. In other words, the mechanisms suggested in this work must be able to (1) support a transparent execution recording of the systems, which are typical for the modern deployments in the datacenter, and research facilities; and (2) provide a verbose analysis interface to both execution history of the system, and its state, which enables quick development of the analysis algorithms in an intuitive way.

The mechanisms developed by this dissertation consists of two main parts: (1) a set of extensions to the Xen virtualization platform that implement efficient, deterministic replay, and (2) a powerful analysis engine that extracts information from systems during replay executions (Figure 1.2). The deterministic replay framework presented in this dissertation is called XenTT for *time-traveling* Xen.

A number of careful design choices ensure that deterministic replay mechanisms developed in this work, which support replay of single-CPU, paravirtual, Linux guests, are efficient, maintainable, and extensible. The run-time checking and offline log-comparison tools enable an efficiently scaling of the recording layer through providing essential mechanisms for detecting and debugging errors in the determinism of replay.

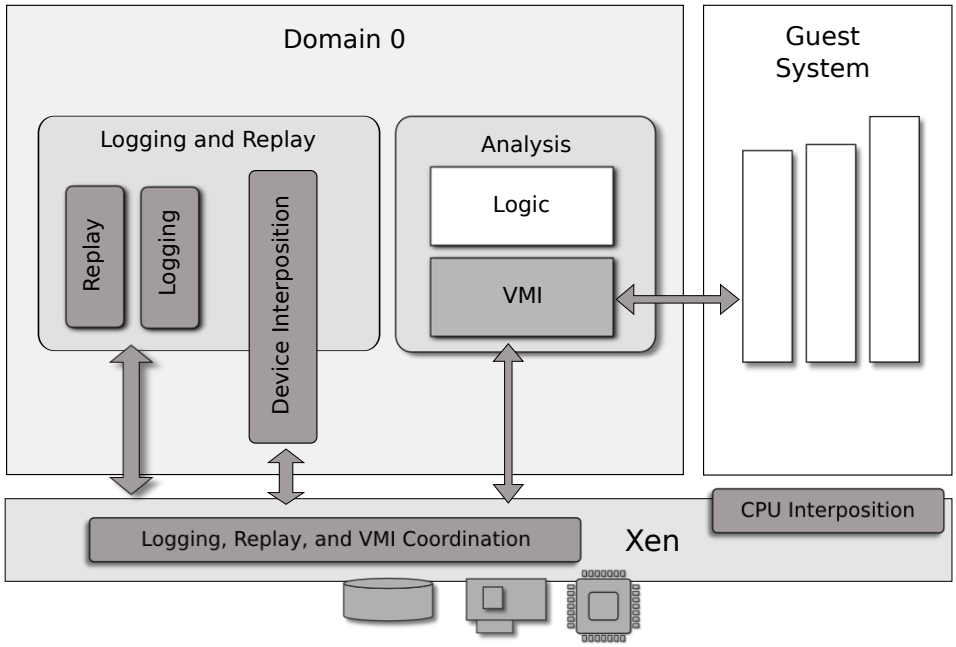

Figure 1.2. Architecture of the XenTT deterministic analysis framework

The analysis engine developed in this work seeks to overcome the semantic gap between an analysis algorithm and the low-level state of a guest. Using debugging information to reconstruct functions and data structures within the guest, the analysis engine provides a convenient API for implementing systems analyses. The XenTT analysis engine implements a powerful debug-symbol and virtual machine introspection library, which enables an analysis to access the state of the guest through familiar terms. To further simplify the development of new analyses, the engine provides primitives that support common exploration patterns, e.g., breakpoints, watchpoints, and control-flow integrity checking.

To enable performance analysis during replay, XenTT extends traditional replay algorithms with the ability to capture information about the performance of the system. At any step of the off-line execution, XenTT's performance model provides a dynamic analysis with a realistic model of system's performance during the original run.

1.5 Contributions

This dissertation argues that deterministic replay should become a ubiquitous mechanism, and an integral part of every software platform. Specifically, this work aims to

demonstrate that deterministic replay can be implemented for an industrial-quality platform in a way that addresses practical scale and usability concerns, and enables a future in which deterministic replay is the de facto basis for analysis of complex software. Design, and implementation of the deterministic analysis framework relies on the following steps, which become the natural contributions of this work:

- **Principles of building deterministic replay**

  This work creates a general engineering foundation for constructing practical replay machines. Several engineering principles are critical for enabling lightweight recording and replay of realistic software systems and realistic workloads. The principles are platform independent, and can serve as a basis for implementing deterministic replay engines in other systems.

- **Principles of implementing deterministic replay for the full-featured virtual machine platform**

  To create a practical analysis tool, this work implements a deterministic replay environment for a full-featured, de facto industry standard virtualization platform. XenTT implementation is designed to address complexity of scaling the replay engine in a real world, i.e., in face of numerous corner-cases inherent to the execution of full operating system stacks, and long, performance-intensive runs.

- **Design and implementation principles for the replay analysis interface**

  To provide a basis for development of a high-level analysis tools, this dissertation defines a minimal, but practical, interface which provides the analysis algorithm with a convenient means of accessing the state and execution history of the system during replay. XenTT's analysis interface implements a powerful debug-symbol and virtual machine introspection layers, which enables an analysis algorithm to access the state of the recorded system through familiar source-level terms. To enable performance analyses of recorded executions, this work develops a performance modeling interface which faithfully replays performance parameters of the original run.

- **Case studies of developing deterministic analysis algorithms**

  To demonstrate practicality of the deterministic replay analysis paradigm, this work presents several examples of replay analysis algorithms. These examples develop several general exploration patterns, which aid implementation of several classes of replay analyses.

Finally, to demonstrate that replay engines can support deterministic record-and-replay for realistic systems and workloads while imposing only small costs at run time, this dissertation evaluates performance of the deterministic replay mechanisms on a set of CPU, and I/O intensive systems benchmarks. In contrast to CPU-intensive workloads, I/O and memory intensive, systems workloads particularly stress the deterministic replay system by generating a large number of nondeterministic events. During development of XenTT, these workloads helped to reveal and eliminate several critical inefficiencies in the design and implementation of the replay system.

1.6 Relation to previous work

XenTT is an environment for the whole-system analysis based on deterministic virtual machine replay. Full-system replay can be implemented as a part of a virtualization platform [64, 63, 187, 55] or a hardware emulator [58] The ReVirt system by Dunlap et al. [64] implemented full-system replay atop the UMLinux virtual machine [28]; Dunlap and his coauthors developed many aspects of full-system replay on x86 and suggested a combination of branch counters, breakpoints, and single-stepping to implement the accurate emulation of nondeterministic events during replay. ReVirt was subsequently ported to User-Mode Linux [97] and Xen [63] and extended with an implementation of a CREW serialization protocol to support multiprocessor replay [65]; the Xen implementation is known as SMP-ReVirt. XenLR provides a partial implementation of a deterministic logging engine for the Xen [117]. VMRS extends XenLR to a full-system replay platform capable of supporting paravirtualized versions of Linux [55].

To the best of the author's knowledge, VMRS and SMP-ReVirt are the only two systems besides XenTT that implement deterministic replay for Xen. VMRS and its precursor XenLR are not publicly available. The author of this dissertation used a publicly available

ReVirt implementation [84] as a reference for the early development of XenTT. SMP-ReVirt was primarily meant to demonstrate replay of SMP systems. Unlike XenTT, it does not implement a general infrastructure to scale to realistic system workloads; does not provide support for the efficient interposition of I/O devices; does not record performance characteristics of the execution; and provides no mechanisms for deterministic replay analysis.

VMware Workstation also supports full-system replay debugging of Linux and Windows guests [187]. It provides support for fast replay navigation and integrates with Visual Studio and GDB. Due to its proprietary nature, the author of this dissertation was not able to use it in this work; XenTT seeks to implement a production-quality deterministic replay environment for whole-system analyses within an open-source VMM.

Several recent efforts have demonstrated techniques for decoupling analysis from execution. For example, iDNA [20], ReTrace [187], and Aftersight [43] rely on deterministic replay to reconstruct the execution of a system for offline analysis. In contrast to this work, none of these systems is able to recreate the performance of a system during replay. Analysis is therefore limited to reasoning about the functional properties of execution.

Tralfamadore supports *execution mining*, an approach that analyzes a complete instruction-level trace of a system [113]. Instead of replaying a system, Tralfamadore captures its complete instruction-level trace by running it under control of a full-system emulator [18]. Although execution mining is powerful, it can be an ill fit for many whole-system analyses because the trace-based representation makes it difficult for an analysis to arbitrarily query the current state of a system. In contrast, XenTT creates an actual virtual machine as part of replay, builds on the performance optimizations in Xen, and provides an API to observe all of a guest's current state.

Several projects provide deterministic replay within language runtimes [40, 189] and for processes [8, 19, 44]. In contrast to these techniques, full-system replay requires no simplifying assumptions about the (whole) system under study. As a result, it provides the most general implementation of system replay.

A common criticism of a full-system replay concerns the semantic gap introduced between a replay debugger and the actual program by a layer of operating system kernel and

other processes [36]. XenTT provides debugger-like access to source-level information and libraries for easing the implementation of analyses; these do not fully address the semantic gap, but they help to provide a basis on which high-level tools can be built.

1.7 Organization

Chapter 2 provides some background and design choices for implementing deterministic replay.

Chapter 3 surveys several decades of work in the area of reversible execution, and replay debugging. Deterministic replay builds on all the techniques, ideas, and mechanisms developed during these years. It is important to understand that this dissertation makes the next step by integrating replay with a state of the art virtualization platform, and thus giving replay a chance to become a ubiquitous mechanism.

Chapter 4 describes the general principles, which create the foundation for building practical replay machines.

Chapter 5 provides a high-level overview of the Xen virtual machine architecture. This chapter is aimed to help the reader to understand the mechanisms which are required to extend Xen with the deterministic replay.

Chapter 6 discusses a practical implementation of the principles presented in Chapter 4.

Chapter 7 provides a detailed description of the interposition boundary for the Xen virtual machine monitor, and implementation of the determinizing proxies for the Xen device drivers.

Chapter 8 describes the interface between analysis algorithms and the state and execution history of the system. This chapter details implementation of the two main components: the virtual machine introspection library, and debug-symbol library.

Chapter 9 presents a performance evaluation of the interposition layer on a set of realistic workloads.

Chapter 10 describes experiences with implementing several replay analysis algorithms.

Finally, Chapter 11 discusses the lessons learned over the course of the work on this dissertation, and provides some insights into the future work.

CHAPTER 2

BACKGROUND AND DESIGN CHOICES

Intuitively, the most straightforward way to replay execution of a system is to record a result of every CPU instruction it executes. On a modern hardware, this can be done by single stepping a system in a CPU debugging mode [78], or perhaps more efficiently by running the system under control of a full-system emulator [112]. Unfortunately, two inherent problems – a huge size of a replay log and a high run-time overhead – make this simple approach impractical in most cases.

To reduce logging overheads, a typical implementation of deterministic replay relies on a key observation that execution of a system is largely deterministic, and is only occasionally altered by external nondeterministic events. For example, being placed in the identical initial conditions, and processing the identical instruction stream, the CPU deterministically generates same values in registers and memory.[1] Determinism of execution holds until some external event, e.g., an external interrupt, or an I/O read from an external source, alters either the state of the CPU, or the system's memory. This observation leads to the following principle: an execution of a system can be viewed as a chain of deterministic executions connected by nondeterministic transitions.

This is a powerful idea. Starting from the initial state, one can force the system to repeat its original execution by just replaying external events. Two conditions should hold for the above approach to work: (1) determinism of execution must to be preserved between nondeterministic events; (2) all nondeterministic input must be available for interposition during logging and replay.

For a complex system, these two conditions are complementary. If it is impossible to provide determinism of execution between nondeterministic events, a violation of deter-

[1] There are anecdotal examples of a nondeterministic CPU behavior [27]

minism can be captured as an external nondeterministic event. Similarly, if it is possible to change a replay boundary including or excluding parts of a system without breaking determinism, one can reduce the amount of nondeterminism which has to be logged.

The above duality provides some flexibility in choosing the boundary separating a replayed system from the outside world. Since the ultimate goal for most replay systems is to minimize run-time overheads, replay interface is typically chosen to ensure the least amount of logging. Following this principle, replay systems extend replay interface to a group of communicating processes or a network of machines, and replay them together [17]. It is also common to extend replay system with a time-traveling copy-on-write storage [104] to avoid logging of enormous amounts of inherently deterministic disk communication [104].

Ideally, a replay interface should capture only the "true" nondeterminism, i.e., input from external sources altering execution of the system. In practice, however, a large fraction of nondeterministic events have to deal with ensuring deterministic execution of the external environment.

2.1 Choice of the replay boundary

An implementation of every replay system requires identification and interposition of all sources of nondeterminism. Complexity of tracking nondeterminism and the ultimate goal to minimize interposition overheads result in replay interfaces which typically follow one of the existing functional interfaces in the system. A logical separation of functionality and data flows simplifies analysis of external input sources and minimizes amount of logging.

Historically, two such interfaces exist in a commodity systems stack. First is a system call interface between the operating system and user-level applications. Second is a hardware CPU interface between the operating system and the CPU. These interfaces naturally define two most popular approaches to deterministic replay – the process level, and full system replay.

This dissertation argues that an essential step towards building a practical replay environment is to combine a deterministic replay and analysis engine with the performance

of an efficient, full-featured, production-quality virtualization platform. Three reasons motivate such binding: (1) the opportunity to minimize logging overheads, (2) the ability to apply analysis to the entire software stack of the system, and (3) the option to reduce setup and installation costs of the analysis solution.

Modern virtualization platforms are designed to provide a low-overhead virtualization of the system's resources. Virtual machine monitors are optimized to provide the high-throughput asynchronous I/O, low-latency interrupts, and fast memory management operations. The virtual machine interface is small, highly-optimized, and representing the minimal cut separating the data inputs form the software stack required to process this data. Extreme optimization of the virtual machine interface makes it a perfect candidate for implementing a low-overhead interposition layer required for logging and replaying all nondeterministic inputs entering a complex software stack of a modern system.

A general dynamic analysis tool is expected to provide a pervasive mechanism capable of reasoning about any part of the software stack – from the low-level parts of a kernel to user-level applications. Full system virtualization provides such option.

As virtualization gradually evolves into a de facto standard for a new computing environment – modern data centers run unmodified operating systems on a virtual hardware, desktop distributions are configured to run on top of a hypervisor. If the replay engine is part of the hypervisor environment, it requires virtually no installation or configuration costs for starting an analysis. Furthermore, most application and kernel-level components just run out of the box.

2.2 Choice of the virtualization technology

Full virtualization is a virtualization technique which implements complete virtualization of the host machine, and enables execution of unmodified guests. To support execution of unmodified guests, full virtualization implements virtualization of the two main components of the physical platform: CPU and peripherals. A virtual machine monitor is required to emulate execution of: privileged CPU instructions, the memory management unit (page tables, segments, etc.), hardware interrupts, traps and exceptions, timers, the main system board, device buses, functionality of the BIOS and the BIOS boot

protocol, and hardware devices like disk and network controllers, serial line, etc.

Until recently, the traditional way to implement efficient virtualization of the CPU required binary translation. Binary translation is used to parse the instruction stream of the guest system, and replace all sensitive CPU instructions with the calls to the virtual machine monitor. Since the mid 2000s, both Intel and AMD processors provide support for the full virtualization of the CPU in hardware. Hardware support eliminates the need for binary translation, and significantly simplifies parts of a fully virtualizing hypervisor, which are responsible for virtualization of the CPU. Virtualization of the peripheral devices remains unchanged, and still requires significant emulation efforts. A typical virtual machine monitor incorporates a hardware emulator like QEMU [18], which provides virtualization of the main system board and peripheral devices.

An alternative to full virtualization is paravirtualization. Paravirtualization is a technique which exposes parts of the virtual machine monitor to the guest system. Paravirtualization almost completely eliminates the need for emulating the real hardware like the motherboard, BIOS, the BIOS boot protocol, device buses, disk, and network devices. Instead of emulation, a paravirtualized guest system uses a set of explicit hypervisor calls to modify the state of the virtualized CPU, and memory management unit. Paravirtualization relies on a high-level device bus to configure virtual device drivers (console, device bus, disk, network, etc.). Instead of the I/O instructions and the DMA engine, the paravirtual device drivers communicate with the hypervisor via an explicit, high-level communication protocol. This protocol can be optimized to meet latency and throughput requirements of individual devices. The guest system controls its virtualized page tables, interrupt descriptor tables, and other parts of a virtualized CPU through the set of explicit hypervisor calls. Privileged CPU instructions are also replaced with explicit hypervisor calls. Traditional CPU ring levels are used to protect the hypervisor running in the Ring 0 from the guest system, which runs in Rings 1 and 3.

2.3 Why paravirtualization is the right choice

Introduction of the hardware-assisted virtualization created a misconception that paravirtualization is no longer a first class virtualization technique. To some extent, it became

a belief that only the full, hardware assisted virtualization is the only right design choice for a virtualization platform. It is true that full virtualization is required to run proprietary, or legacy operating systems, in which support for a virtualized interface is impossible. Paravirtualization, however, is the right choice for the systems, which are actively developed and supported.

The main advantage of paravirtualization is the ability to eliminate the components of the hardware emulator from the virtualization stack. First, without the emulator's code, a virtualized system has a smaller memory footprint – there is no need to pair every guest virtual machine with an instance of an emulator. Second, without the hardware emulator, the virtual machine monitor has a smaller attack surface. Finally, paravirtualized systems are faster, since an emulated APIC interrupt controller requires additional context switches on the interrupt delivery path.

2.3.1 Legacy problems of paravirtualization

Paravirtualization has a couple of legacy problems, which are often incorrectly interpreted to block the future of paravirtualized systems. First, while transitioning to the 64-bit execution mode, AMD and Intel CPUs removed support for the segment registers. The removal of segment protection effectively results in the removal of the traditional concept of protection rings. Without the segment registers, an operating system is left with only the "supervisor" bit in the page table, and thus can implement only two protection rings. Unfortunately, this has an immediate performance implication on the speed of the context switches between the user and kernel levels of the guest system.

A typical hypervisor implements protection across three layers of the system: hypervisor, guest kernel, and guest user. Guest user level applications run in the Ring 3, guest kernel runs in Ring 1, and the hypervisor runs in Ring 0. To protect guest kernel from the guest user-level applications, the hypervisor uses the user-system bit in the page tables. Since the x86 architecture treats rings 0, 1, 2 as supervisor mode, guest kernel, which runs in the Ring 1, can access the memory mapped with the system bit enabled, while the guest user applications, which run in Ring 1, cannot. However, since both Xen and the guest kernel run in the supervisor mode, the user-system bit protection is insufficient. To protect

the hypervisor from the guest kernel, Xen relies on the segment protection mechanism. Xen shares the same address space, i.e., same page table, with the guest system. However, Xen protects its memory with a segment which is accessible only from the Ring 0.

To protect itself, without the segment registers, the hypervisor needs to be the only code in the ”supervisor” mode. The hypervisor runs both guest user applications and guest kernel at the same protection level (Ring 3). To implement protection between guest user and kernel levels, Xen creates two separate address spaces, i.e., two sets of page tables. Isolated address spaces imply that every system call in the guest system requires an exit into the hypervisor, and then a complete context switch of the address space.

The second problem with paravirtualization and replay is related to the fact that deterministic replay requires support for virtualization of page tables. Traditional paravirtualization exposes the layout of the physical memory to the guest system. Unfortunately, the allocation of physical memory can be different across original and replay runs, and therefore can break determinism of execution if exposed to the guest system. Deterministic replay traditionally solves this problem by implementing virtualization of page tables in software. This special paging mode is known as the automatically translated physical map (`auto_translated_physmap`). It was originally implemented as part of the rapid, copy-on-write, virtual machine cloning project [176].

At some point around Xen version 3.0.4, the support for page table virtualization was broken which created a misconception that virtualization of page tables became unavailable for the paravirtualized kernels. Thus, deterministic replay of paravirtualized guest kernels is inherently limited to the Xen 3.0.4, which is used in this dissertation. Fortunately, this is not true. First, the code for the automated translated physical map has never left the Xen source tree. It is broken in the current unstable version of Xen 4.4; however, all the critical parts are there and it does not seem too hard to fix. But, more importantly, the support for hardware virtualization of the page tables is coming to the paravirtual Xen interface.

2.3.2 Future of paravirtualization

Both problems with paravirtualization will be fixed in the new versions of Xen, which is being extended with the support for the hardware-supported nested page table virtual-

ization. The idea is that a paravirtualized guests will rely on the hardware page tables to virtualize their memory instead of using the physical (machine) memory. The main motivation for introducing this change is the desire to simplify the memory management code in the hypervisor – direct access to page tables is fragile and often broken at the early boot stages [62]. Hardware virtualization of the page tables will eliminate the need for the support of automatically translated physical map in software. Furthermore, sine the Xen hypervisor will use the Ring -1, this change will enable to solve the problem of missing protection rings on the 64-bit AMD and Intel CPUs.

Today, paravirtualization became a default part of the main Linux kernel tree through the PVOPS project. Paravirtualization provides a clean and efficient virtualization technique, and might even become the only virtualization technique for the Linux family of operating systems [62]. Therefore, paravirtualization as the main virtualization technology for the deterministic replay is a meaningful choice for this dissertation, which aims to support efficient replay of the guest systems based on the open-source Linux kernel.

2.4 Choice of the hypervisor

Several virtualization platforms were possible candidates for choosing them as a basis for the full-system deterministic replay when this work was started: VMWare, QEMU, VirtualBox, KVM, and Xen. An optimal choice of the virtualization platform for this dissertation was a fast, mature, full-featured, open source hypervisor with large and active user and developer communities.

VMWare is a proprietary virtualization platform, which was available for this work through the academic license program. The advantage of using VMWare would be the ability to reuse VMWare's existing implementation of the deterministic replay, while concentrating the research effort on developing techniques and mechanisms for deterministic systems analysis. The disadvantage would be the limited audience for which this work would be available. The intuition of the author of this dissertation was that it will take at least a year to get familiar with the VMWare code base, and that in an equivalent amount of time, it would be possible to develop an open source replay engine for the Xen VMM.

In 2009, both QEMU and VirtualBox were providing a mature, fast, open source virtual-

ization solution. To implement virtualization, both VMMs were relying on the techniques of binary translation. The support for the hardware virtualization was planned in both projects, but was not complete. Several reasons were motivating the use of the binary translation engine as part of the deterministic analysis infrastructure. First, the binary translation engine is a critical building block for a variety of dynamic analyses, which require fine-grain instrumentation of the instruction stream, e.g., the data-flow and taint tracking. Second, binary translation can be used to implement the accurate instruction counting in software. The advantage of the software branch counting is reliability. The limitation of the software branch counting is the need to translate every part of the guest system – both the kernel and user-level code. A large amount of binary translation would likely have a prohibitive performance penalty on the execution of the guest system. Third, the technique of binary translation is a promising approach for implementing replay of the SMP systems. Binary translation can be used to implement lightweight recording of all parallel memory accesses to the memory shared across the cores [37]. The main reason to avoid a virtualization platform with the binary translation engine was the idea that hardware assisted virtualization with elements of paravirtualization will become the de facto virtualization technique in the nearest time. Furthermore, the author wanted to avoid the complexity of implementing replay for a seemingly more complex virtual machine interface provided by the binary translation engine. However, since the time this dissertation was started, several projects implemented deterministic replay for the QEMU emulator [37].

KVM and Xen were probably equally good choices for this work. The advantage of using KVM is the fact that it became a default part of the Linux kernel. Thus, the setup costs of the deterministic replay would be even smaller for the end users compared to Xen. On the other hand, the choice of KVM would limit implementation of deterministic replay to the hardware-assisted virtualization interface (see Section 2.2), and would require one to support determinism in the QEMU emulator, which is used to emulate hardware devices for the KVM guests. In 2009, Xen was more mature, and had a larger user base. The author of this work was also more familiar with the Xen architecture which promised him a faster development path.

Multiple reasons motivated the choice of Xen as the basis for this work. The Xen virtual machine monitor is a full-featured, open-source virtualization platform [15]. Xen offers an excellent virtualization performance [186, 174], support for a wide variety of guest systems and hardware platforms, and a full set of features required by *de facto* virtualization standards. Xen is widely used as a virtualization provider in commercial datacenters [6] and large-scale academic research facilities [182]. Xen is an open source what guarantees availability of XenTT for a broad variety of research and commercial projects.

CHAPTER 3

RELATED WORK

Accumulating several decades of active research, replay mechanisms create a solid foundation for the systems debugging and analysis tools. Recent replay implementations demonstrated the feasibility of capturing the execution of an entire system with a CPU instruction precision, and overhead of only several percents. Such nonintrusive recording inspires a new debugging paradigm, and uniquely enables application of sophisticated run-time systems analyses, overhead of which is traditionally considered prohibitive for the run-time execution. Replay mechanisms decouple analysis from execution, splitting it in two steps. First, the execution of a system is recorded in a live production environment. Then, the exact run-time state of a system is deterministically re-created off-line for iterative inspection and analysis.

Efficient implementations of deterministic replay and applications of the replay engines remain an active area of systems research. Existing applications of deterministic replay include debugging of the user-level and system-level code, debugging of real-time, concurrent, and distributed systems, model-checking and delta-debugging, fault-tolerant full-system replication, checkpointing and migration, resilience to transient and nontransient faults, forensic analysis, lightweight execution trace collection and compression, automated search for configuration errors, and many more.

This chapter surveys evolution of the systems replay tools. It presents a historic perspective on the development of early replay implementations, surveys a range of existing replay approaches, and then concentrates on the most recent applications and analyses enabled by modern replay mechanisms.

3.1 Historical perspective

Early implementations of the checkpoint and undo capabilities root back to the first computing systems. Leeman mentions that in 1948, the ENIAC computer [29] relied on the system checkpointing to recover the computation of a nonlinear differential equations [98]. Undo recovery features can be found in early fault-tolerant systems, databases, programming languages, parallel debuggers, and text editing applications. Early database systems suggest a combination of checkpointing and logging to replay execution of a system by rolling it back to the nearest checkpoint and re-executing the transaction log [172]. Distributed checkpointing protocols replay communication between nodes of a distributed system to bring recovered nodes to the most up-to-date consistent state [16]. Theoretical approaches study repeatable execution from the point of function inversion [123]. Undo actions are implemented in the context of text editing and formatting applications [88]. Recent replay tools seek to provide an application and language independent support for debugging of complex long-running applications, and even entire operating systems.

3.1.1 Early work

An excellent overview of the early work in the area of reversible execution is given by Leeman [98]. Several surveys provide taxonomy of an early [60] and more recent work [93, 47] in the area of replay debugging. An excellent work by Elnozahy et al. gives a comprehensive survey of the rollback recovery and checkpointing protocols [67].

Early fault-tolerant systems develop methods of backward recovery [33]. Notions of checkpointing and transaction logging exist in the design of the earliest database systems [172]. Multiple researchers create a formal foundation for reversible functions [123, 157, 57] and backtracking. Korf suggests a method of inverting programs in Lisp [106]. Several programming languages offer undo capabilities for deleted objects [166], checkpointing [72], and a reversible assignment operator [83].

A number of text editing applications introduce support for undoing the most recent changes. Hansen uses an undo buffer for the most recent change [88]. Berstis designs a text editor with a general checkpoint/restore mechanism (see[98] for details). Bravo editor is capable of logging and replaying all transactions in the editing session [109].

3.1.2 Decoupling analysis from execution

Exdams is the first debugging system to suggest a two-stage replay debugging paradigm, which decouples analysis from execution [10]. Exdams runs the program once recording its execution on a history tape. During debugging or analysis stage Exdams reads recorded information from the tape. To record execution, Exdams saves values on the left-hand side of the assignment statement, directions taken in conditional statements and loops, and program points, from which `goto` or `call` instructions are invoked. During debugging sessions, Exdams supports program replay and reverse execution.

3.2 Development of system replay

Contemporary notion of system replay originates as a part of distributed checkpoint protocols [16], replay debuggers for parallel systems [56], and fault tolerant replication approaches [25].

3.2.1 Replay debugging of parallel applications

Parallel replay debuggers aim to bring cycling debugging experience to nondeterministic parallel programs. Inherent nondeterminism of acquiring synchronization primitives, and receiving messages make traditional cycling debugging approaches impractical. By eliminating all sources of nondeterminism, replay debuggers force concurrent systems to repeat original run. During replay, the system can be inspected with traditional cycling debugging tools.

There are two approaches to handle nondeterministic communication in parallel systems known as *value* and *order* determinism. Value logging records all values read from shared memory, and received in external messages [138]. In case of frequent communication, this straightforward approach results in a high logging overhead and large log files. Order logging makes an observation that it is possible to log only the order of communication between the nodes [34, 110, 131, 165, 17, 65]. Determinism of replay ensures that system resends same messages during a debugging run. It is possible to gain further logging optimizations if semantics of the system allows coarse-grained sharing of memory via explicit synchronization primitives [165] or object interfaces [34, 110, 115, 61].

BugNet is probably the first debugger designed to provide system replay for analysis

of time-sensitive errors in a distributed system [56, 183, 184]. BugNet assumes explicit message-based communication, and synchronous signals. To ensure correct ordering of events across the network during replay, BugNet implements an optimistic synchronization scheme based on virtual time [96]. Instead of synchronizing the system on every message, BugNet lets the system run until re-ordering happens. Detecting a violation, BugNet tries to rollback the system by sending an "undo" message for each message, which logically appears later in time. Undo message can either cause annihilation of all messages with re-ordered timestamps in the receiver's queue, or if it is too late force a cascading rollback.

Instant Replay is a system for debugging parallel shared memory applications [110]. Instant Replay explores the idea of cooperative replay, emphasising that only the order of accesses to shared variables needs to be logged when all processes participate in replay. Instant Replay leverages coarse-grained sharing of memory by requiring that all accesses to shared variables are guarded by operating system primitives. These primitives serialize and log access to shared memory enforcing a concurrent-read-exclusive-write protocol (CREW) [52]. Instant Replay algorithm was later improved to trace only a minimal subset of memory accesses required to replay execution [131, 132]. Run-time checks allow to eliminate 99% of messages, which need to be logged.

Recap supports replay debugging of parallel applications communicating via message passing or shared memory [138]. Recap relies on a compiler support to generate interposition code for shared memory reads. Run-time library records results of the system calls, messages, and asynchronous signals. Static analysis and dynamic checks are used to infer whether variable is shared. To replay asynchronous signals, Recap relies on a branch counting support in hardware [32]. Alternatively, Recap approximates exact event time with a system time. Recap combines replay with checkpointing to provide abstraction of reverse execution. Recap records both order and content of all messages to provide replay of individual processes.

Russinovich and Cogswell suggest repeatable scheduling algorithm for debugging multithreaded applications on uniprocessor machines [149]. They realise that in a uniprocessor system, all shared memory accesses are replayed correctly as soon as reply preserves the scheduling order among threads. Comparing to multiprocessor machines, which provide

true concurrency, this approach greatly simplifies implementation of the replay.

3.2.2 Fault-tolerant replication and checkpointing

Fault-tolerant systems leverage deterministic replay as an effective means of transferring system state between nodes [16, 25, 26, 27]. A similar idea is used by log-based checkpoint protocols, which bring individual nodes to a globally consistent state by replaying stale messages instead of freezing the system for a globally consistent checkpoint [67].

NonStop online transaction processing system is one of the first systems to implement a log-based rollback-recovery protocol for fault-tolerant replication [16]. During recovery, NonStop uses message replay to bring failed processes to a state consistent with the rest of the system. Several design choices simplify implementation of replay. First, all communication is message based; therefore, messages are the only source of nondeterminism. This simplifies the task of identifying nondeterminism and provides convenient replay interface. Second, a synchronous message interface eliminates the need to implement branch counting for replay of asynchronous signals.

Auragen (later Targon/32) is the first system, which explicitly uses deterministic replay for fault-tolerant replication of processes in a UNIX system [25, 26]. Developers of Targon/32 recognize that providing determinism of execution is the most challenging part of the project due to a compilcated UNIX process interface. To capture nondeterminism, Targon/32 implements a message-based IPC system forcing processes to communicate via messages [25]. Since determinism of user-level processes depends on the state of the kernel, Targon/32 tries to move most of the kernel functionality to the user-level processes. This results in a microkernel architecture – the system is a set of servers. Such a message-based approach does not work for a highly-optimized low-level system services, e.g., meile-systems, block, and character devices. These services have to implement custom replication strategies. Having no means to support determinism in face of asynchronous UNIX signals, Targon/32 forces a system checkpoint on every signal. To limit recovery time, replay is used in combination with a periodic checkpointing.

3.3 Instruction conting and asynchronous signals

A major drawback of all early replay systems is inability to handle asynchronous events. As a result, a system is either restricted to communicate exclusively via synchronous signals, or alternatively a checkpoint is required after every asynchronous signal. An inherent reason for the lack of asynchronous event support is the absense of mechanisms which can uniquely identify a place in the execution of a program at which an asynchronous event occurs during the logging stage, and can stop the system at the very same point during replay.

Cargill and Locanthi are the first to advocate implementation of a simple instruction counting mechanism in hardware [32]. They suggest a hardware counter, which decrements on every instruction, and raises an exception when it reaches zero. By programming an initial value of such counter, it is possible to implement periodic exceptions which can provide interface for detecting updates of a memory location (i.e., watchpoints), system profiling, and reverse execution. In a suggested implementation of reverse execution, Cargill and Locanthi rather naively assume that execution of a system is deterministic. Therefore, it is enough to restart the system with a counter programmed to cause an exception just before the instruction at which previous run ends.

Multiple researchers recognized the power of instruction counting mechanism to enrich replay systems with general support for asynchronous signals. Facing the lack of hardware support, Mellor-Crummey and LeBlanc are the first to implement an instruction counting algorithm in software [124]. Extending their earlier Instant Replay work, they provide the first implementation of a replay algorithm with support for asynchronous signals. Efficiency of instruction counting comes from the insight that it is not necessary to count every instruction, or even every basic block. Instead, it is enough to count a number of taken backward branches and function calls. Number of taken backward branches and the instruction pointer uniquely identify a place in program's execution. On a typical workload, this approach results in a 10% overhead.

Several replay implementations rely on software instruction counting to provide support for replay of asynchronous events [61, 149, 158, 155, 159]. To further reduce overheads of the software counters suggested by Mellor-Crummey and LeBlanc, Slye and Elnozahy

argue the use of two separate versions of a program instrumented individually for a logging and replay stage [158, 159].

Targeting debugging of real-time systems, and seeking to preserve real-time properties of a system, HMON stresses transparency and low overhead of the logging primitives [61]. HMON implements application replay with support for asynchronous signals, system calls, and shared memory accesses. Although HMON dedicates some system hardware for logging components, it requires no hardware modifications and relies on a software instruction counter for replay of asynchronous events. For tracking shared memory accesses, HMON implements a mutual exclusion algorithm [52]. HMON results in an under 20% overhead.

Bressoud and Schneider rely on one of the first hardware branch counting implementations in their hypervisor-based full-system replication solution [27]. Unfortunately, hardware counter implementation they use fails to provide a way to sample the counter. To ensure determinism, they program the counter to overflow after a fixed number of steps. All asynchronous signals are queued and delivered at the point when the counter overflows.

Russinovich and Cogswell rely on the software branch counter to implement repeatable scheduling algorithm for the Mach kernel [149].

RDB is a first complete implementation of a unique spanning reads algorithm [133]. RDB develops a framework for program instrumentation with support for replay of system calls, synchronous, and asynchronous signals [155]. Implementation of asynchronous signals relies on software branch counting similar to the one suggested by Mellor-Crummey and LeBlanc [124]. RDB results in a factor of two slowdown during program execution.

Bidirectional debugger (bdb) provides an extended notion of software counters to implement common debugging abstractions like step over a function, and finish until function ends [24]. Bdb relies on two counters: *step* and *call depth*. The step counter is inserted at compile time at the beginning of each source line or basic code block. The call depth counter counts depth of the call stack and is used to provide step over a function and finish until function ends abstractions. Compared to traditional debugging techniques, which introduce overhead of a million instructions on every trap, software counters provide a lightweight solution. A nonoptimized counter instrumentation results in less than a factor of two overhead. Furthermore, counter checks provide a good way to implement software

watchpoints. To ensure determinism of execution, bdb replays results of UNIX system calls.

Thane et al. suggest an interesting alternative to branch counting [168]. Instead of implementing a software or hardware branch counting algorithm, Thane et al. suggest to pair the instruction pointer with the checksum of registers and a program's stack to uniquely identify the place at which an asynchronous event occurs. Although not being strictly unique, this approach provides a practical solution for realistic programs.

3.4 Choices of the replay boundary

3.4.1 Process level replay

Process-level replay aims to provide an application and language-independent mechanism for replay of a broad class of programs [155, 48, 161, 151, 77, 85]. Process-level replay leverages a natural boundary of the operating system call interface to isolate an application from the external nondeterministic environment.

A process level replay solution can be implemented inside [48, 161] or outside [151, 77, 85] of the operating system kernel, i.e., as an extension to a kernel or as a library within a process. The latter approach offers greater portability, operating-system independence, and simpler deployment, but often lacks completeness of replay implementation. For example, unless implementing a software branch counting, library-level replay tools fail to support asynchronous signals due to a complex branch counting logic, which has to distinguish between replay and nonreplay code inside a single process [151, 77, 85].

Tornado is a kernel-level implementation of process level replay [48]. Tornado has a simplified model of nondeterminism, which allows only nondeterminism caused by system calls.

Flashback implements a transparent process-level replay for Linux applications [161]. Aimed to provide a light-weight debugging solution, Flashback is designed for short replay runs, during which it keeps all replay information in memory. Flashback creates shadow copies of all kernel structures describing a process to checkpoint its state. This state includes process memory (stack and heap), registers, file descriptor tables, signal handlers, synchronization primitives for multithreaded processes, and other in-memory state associated with the process [161]. Content of all disk and network communication is logged and

replayed. Flashback discusses support for multithreaded applications on a uniprocessor machine as a future work. Flashback implements limited support for memory mapped files by saving content of the pages during the first page fault. Naturally, this approach does not work for memory mapped files used concurrently by multiple processes. For the same reason, shared memory is not supported. As Flashback implements no instruction counting, logic replay of asynchronous signals is not supported.

Aiming for transparency of deployment, Jockey implements process-level replay as a library [151]. Since Jockey shares an address space with the target process, a seriously damaged application can possibly harm the replay engine itself. To minimize likelihood of damage, Jockey applies several techniques to conceal its resources. It allocates its resources in a custom mmapped region of memory, relies on its own stack, and accesses file system via private file descriptors. Jockey cannot control kernel-based thread implementation, but implements support for user-level thread packages, such as Capriccio [175]. To intercept system calls and nondeterministic CPU instructions, Jockey scans the text section of a program and injects interception code. Jockey handles asynchronous signals by converting them to synchronous upcalls. Jockey intercepts the signal, and defers signal delivery until the program executes one of the system calls or nondeterministic CPU instructions. For file system accesses, Jockey relies on the "undo" logging, and records enough information to restore content of the file during replay. During replay, the "undo" log is applied in reverse order to recreate original file. For the rest of communication, Jockey logs content of a message. To log access to memory mapped files, Jockey intercepts the mmap system call, ensures the memory region is mapped read-only, and intercepts a page-fault signal similar to Flashback.

Liblog is another library implementing replay of distributed applications [77]. Liblog does not support cooperative replay and logs content of all incoming messages on all nodes. Liblog enables off-site replay; however, it requires that software and hardware match between original and replay run. To support replay of multithreaded applications, Liblog implements repeatable scheduling algorithm [149], serializing scheduling of threads inside a process, and effectively implementing a cooperative scheduler. Threads volonteerely yield their quantum upon return from a system call. Asynchronous signals are queued until

one of the threads receives them returning from a system call. Cooperative scheduling implies that Liblog cannot be used for applications executing tight infinite loops.

R2 is a recent library-based process level replay tool for the Windows platform [85]. R2 allows users to pick replay boundary choosing arbitrary functions. This interface has to be faithful and ensure that all sources of nondeterminism are captured. Implementation of recording and replay functionality relies on the Detours Windows library [92]. R2 automatically generates record and replay stub functions from user-provided annotations. Annotations serve the purpose of describing arguments passed by reference via pointers. Similar to Jockey, R2 implements separate allocators and stacks for replayed and nonreplayed parts of application. R2 relies on Lamport clock [108] to preserve causality between events on a multiprocessor machine.

3.4.2 Full-system replay

A low-level machine interface is another natural boundary separating a system from external environment. Full-system replay records execution of the entire operating system along with all user-level processes [64, 63, 187, 55]. It can be implemented as a part of a virtualization platform [64, 63, 187, 55] or as an extension to a hardware emulator [58]. Deployed as a part of commercial virtualization, replay becomes a transparent low-overhead debugging tool which can be enabled on demand to analyse bugs in a production environment. Capturing execution of the system at the hardware level, full-system replay offers a unique way of debugging low-level operating system code [104].

In contrast to process-level solutions, full-system replay is capable of providing a complete implementation of replay requiring no simplifying assumptions about the replayed system except the ones required by virtualization. A common criticism of a full-system replay concerns the semantic gap introduced between a replay debugger and an actual program by a layer of operating system kernel and other processes [36]. A recent approach from VMWare addresses this issue by erasing unneeded parts of a recorded execution [44]. During replay, it is possible to re-record execution of the system while erasing unneeded periods of system execution, filtering sensitive data from the recorded session, and if needed, moving from a full-system to a process-level replay.

ReVirt is a pioneering work in the area of full-system replay [64]. ReVirt implements a full-system replay on a UMLinux virtual machine [28]. It develops approaches for using hardware branch counters, and logging nondeterministic instructions like `rdtsc`. ReVirt's implementation of a branch counting is greatly simplified by a hardware support provided by the recent processors. Hardware provides separate counters for lower and higher hardware protection levels. This separates branch counting between a replay layer, which is implemented inside a hypervisor running at the lowest protection level, and a recorded guest system. ReVirt suggests a combination of branch counters, breakpoints, and single-stepping to inject nondeterministic events during replay. To uniquely identify event occurence in the instruction stream, ReVirt uses a triple of number of taken branches, instruction pointer, and the `ECX` register. ReVirt adds 0-8% logging overhead to the existing 13-58% overhead of the UMLinux virtualization.

ReVirt was originally designed to provide a replay platform for intrusion analysis [64]. Later ReVirt was ported to User-Mode Linux [59] and Xen [15] virtualization platforms [97, 63, 65]. It was extended with implementation of a CREW serialization protocol [52] to support multiprocessor replay [65], and cooperative logging for replay of a network of machines [17]. ReVirt became a platform for intrusion analysis [103, 97], debugging of a low-level system code [104], and live log-replay based migration of virtual machines [116].

ExecRecorder [58] implements full-system replay on the Bochs processor emulator [23]. To support replay of asynchronous events, ExecRecorder relies on a time ticks provided by the Bochs emulator (tick is one executed instruction). To limit the growth of a log size, ExecRecorder saves only a time difference between consecutive events.

Starting from version 6.0, VMWare Workstation includes support for full-system replay debugging of Linux and Windows guests [187]. VMWare Workstation can replay a wide variety of device drivers, provides support for fast replay navigation, and implements integration with Visual Studio development evnironment and GDB. At the moment, it does not include support for multiprocessor replay.

XenLR provides a partial implementation of a deterministic logging engine for the Xen virtual machine monitor [117]. XenLR is aimed to become a practical replay solution implemented as a part of a popular Xen virtualization technology. Initially, XenLR imple-

mented only logging support and was limited to a restricted set of virtual device drivers.

VMRS, extends XenLR to a full-system replay platform capable of supporting paravirtualized versions of Linux [55]. Unfortunately, VMRS leaves unclear many details specific to the implementation of replay mechanisms for paravirtualized guest systems.

3.4.3 Language-level replay

Platform-independence motivates implementation of program replay mechanisms at the language level [165, 105, 162]. Language-level replay can be implemented by transforming a program to include an interposition code logging and replaying a nondeterministic input [11, 165, 140], or inside a language run-time [105, 162].

Tai et al. rely on a source-to-source transformation to record and replay Ada applications [165]. A compiler produces two versions of a program instrumented for recording and replay.

DejaVu suggests a solution to replay multithreaded Java applications [41]. DejaVu implements a version of a repeatable scheduling algorithm [149], but enhances it introducing a notion of a logical schedule, i.e., combining multiple consecutive schedules in a single schedule in case this does not affect behaviour of a system during replay. DejaVu is implemented at the Java virtual machine level. It is capable of handling nondeterminism of Java synchronization constructs, and some windowing events. DejaVu neglects nondeterminism of a file system input, and is incapable of replaying native methods. Later versions of DejaVu were extended to support replay of distributed applications across multiple virtual machines [105], and replay of entire Java Virtual Machine including Java native interface methods [4]. On a uniprocessor system, DejaVu results in up to 88% recording overhead.

jRapture [162]. Schuppan et al. suggest a JVM independent approach to replay Java applications [153]

3.5 Replay applications

Probably the most unique capability provided by deterministic replay is a way to replicate execution of a system with an overhead acceptable for production environments. Execution of a system can be replicated in time and space. It can be recorded in a production environment and recreated for off-line debugging, and heavyweight analyses [43]. Al-

ternatively, replicas of a running system can be kept on-line, providing a fault-tolerant execution [16, 25, 26, 27] or enforcing various access control and security policies [140, 154, 191, 135, 164, 164].

Aside from replicating execution, deterministic replay proves to be an effective debugging means for nondeterministic programs, i.e., programs which may return different results on the same input. Replay debugging is a means to debug "heisenbugs" [82], i.e., transient faults like race-conditions, which show up once and are likely to disappear in subsequent debugging runs.

3.5.1 Debugging

Replay debugging was originally aimed to bring cycling debugging experience to nondeterministic programs. It was later realized that combination of a lightweight recording and deterministic replay provide a unique opportunity for debugging systems, in which traditional inspection with a debugger is impossible [11, 171, 167, 169].

3.5.1.1 Application debugging

Process-level replay implementations create a foundation for application debugging [155, 48, 161, 151, 77, 85]. Flashback provides a light-weight application debugging solution [161]. Liblog and its distributed extension Friday implement a library-based approach for debugging Linux applications [77, 76].

GDB version 7 implements support for replay debugging and reverse execution [78]. GDB implements replay by recording every instruction and its side-effects in registers and memory. During reverse execution, GDB undoes changes and re-executes an instruction again. This straightforward approach results in high run-time overheads and large logs.

3.5.1.2 System-level debugging

King et al. are the first to suggest full-system deterministic replay as an effective means of debugging low-level operating system code [104]. Operating systems are complex software artefacts which run for a long time, exhibit nondeterministic behaviour, interfere directly with hardware, and often crash under transient conditions. King et al. use ReVirt [64] to capture execution of an operating system for iterative off-line debugging.

A guest operating system has to be changed to run in a paravirtualized UMLinux environment. This restriction prohibits debugging of the lowest hardware-dependent parts of the operating system code.

3.5.1.3 Multiprocessor debugging

Instant Replay [110] and Recap [138] are probably the first debuggers providing replay of parallel shared memory and message-passing applications (Section 3.2.1).

SMP-ReVirt relies on a hardware page protection mechanism to implement logging and replay of multiprocessor virtual machines on comodity hardware [65]. SMP-ReVirt serializes shared accesses with a CREW protocol [52]. Coarse-grained page-size serialization is aimed to reduce recording overheads, which are high for a multiprocessor system. For many workloads, SMP-ReVirt shows overheads acceptable for debugging. Replay system is implemented as a part of the Xen virtualization platform.

3.5.1.4 Pseudo-deterministic debugging

Facing the high costs of recording in a multiprocessor system, replay tools borrow an insight from relaxed memory consistency models. Several approaches relax fidelity of replay aiming to record incomplete information and recompute it during replay step.

Replicant suggests to relax determinism of replay on multiprocessor machines [143]. Instead of forcing ordering of all communication between threads, Replicant only requires that replay matched original execution by producing identical external output. This reflects the idea that, in many cases, differences in the order of communication between threads do not change behaviour of the entire program. In case logic of a program depends on event ordering, a programmer has to annotate source code of the application so Replicant can preserve ordering of the events during replay.

ODR argues that although a high-fidelity determinisitc replay is sufficient for debugging of a multiprocessor system, it is not necessary. Instead, during replay, it suffices to produce *any* execution which has outputs identical to the recorded run [5]. Relaxing fidelity of determinism, ODR avoids the price of recording outcomes of all data-races in shared memory. Lacking this information, during replay ODR searches through all possible executions to find runs with required outputs. To improve the search, ODR tries to direct

the search towards executions which share properties of the original run. Further, ODR assumes a null memory consistency model making no guarantees that reads return values of the most-recent writes. This may complicate the process of tracking the cause of an error across multiple threads.

Similar to ODR, PRES relaxes determinism of recording by giving up a traditional replay debugging goal to reproduce a bug at the first replay attempt [139]. PRES records only partial execution information (execution "sketches") and explores the execution space during replay to recreate the bug. PRES suggests several recording schemes: record only global orders of synchronization, global order of synchronization primitives and system calls, global order of function calls, and global order of every N-th basic block. Recording global order of synchronization and system calls, PRES manages to keep logging overhead under 20% on 8 core machine, and in most cases is able to reproduce a bug within 10 replay attempts.

3.5.1.5 Distributed systems debugging

BugNet [56] is probably the first implementation of a replay debugger for distributed systems (Section 3.2.1). Harris argues that wide-area distributed computing needs a pervasive debugging support, and suggests to use virtualization to seamlessly integrate a debugger with a system [89, 91].

DejaVu targets replay debugging of distributed Java applications [105]. In their technical report, Basrai and Chen suggest to extend ReVirt [64] with support for cooperative-debugging of a network of machines [17].

Friday is a library-based replay debugging tool for distributed applications [76]. Friday implements a traditional symbolic debugger and a language to express high-level distributed debugging actions. Friday is shown to debug state abnormalities caused by routing errors in overlay networks. Friday is criticised for a high run-time overhead on data intensive applications.

MaceMC is a software model-checker for finding liveness violations in distributed systems [102]. It implements model-checking for distributed applications written in the Mace language, which represents software in a way similar to state machines.

3.5.1.6 Real-time systems debugging

A progress of a real-time system heavily depends on interaction with the external environment, which progresses in real physical time and requires a timely response from a system to move forward. This creates a major problem for applying traditional cycling debugging techniques to a real-time system. In a typical debugging scenario, a cycling debugger suspends execution of a system to inspect its state. Missing a response from the system, the external environment progresses on its own. Becoming incoherent with the environment, the real-time system is unable to make any further progress.

In most cases, it is impossible to force a real-time system to make progress and reproduce errors during a cycling debugging session. This leads to a development of replay approaches suitable for debugging of real-time systems. If execution of a real-time system can be recorded with a low overhead, then traditional cycling debugging can be performed on a replay session. Daniel Sundmark provides a brief state of the art report covering replay debugging of embedded real-time systems [163].

Banda and Volz are the first to propose a nonintrusive hardware recording and replay mechanisms [11]. They suggest to split debugging of a system in two phases: a nonintrusive hardware-based recording, and a replay debugging session. Banda and Volz rely on a special hardware to monitor nondeterministic events. A nondeterministic input is detected at compile time which requires compiler support.

Tsai et al. rely on a hardware monitoring unit to implement a noninterference logging of real-time programs [171]. The system is criticized for recording unnecessary data and introducing unpredictable run-time overheads [61].

HMON provides support for replay debugging of distributed real-time systems [61]. In contrast to previous work, HMON implements logging mechanisms entirely in software and requires no special hardware support. To reduce interference with a system, HMON allocates one general purpose processor on every multiprocessor node for logging.

Thane et al. implement replay debugging for the VxWorks real-time operating system [167, 169]. Instead of relying on a software or hardware branch counter mechanism, authors identify the exact location of asynchronous events by checksumming a task's registers and stack.

3.5.2 Delta debugging and model checking

Choi and Zeller use deterministic replay of Java applications to implement a *Delta Debugging* approach [192]. It automatically isolates a minimal scheduling sequence resulting in a concurrency bug in a multithreaded application [42]. Choi and Zeller replay the application multiple times to generate alternate schedules and systematically narrow the difference between a correct and failing thread schedule.

Chess is a model checker, which relies on deterministic replay to eliminate nondeterminism in a multithreaded execution and systematically test all possible thread interleavings [128]. Chess implements a cooperative thread scheduler allowing thread preemption only at designated synchronization points. Execution between such points is deterministic. During the recording phase, Chess instruments synchronization points with a logging code. During replay, it systematically explores all possible schedules across all active tasks. Chess does not ensure complete determinism. Instead, discovering violation of replay, Chess switches back from replay to a recording mode.

3.5.3 Replay-based predicate checking

WiDS checker uses replay to extend predicate checking to a distributed system [118]. Since it is impossible to efficiently check predicate assertions across multiple distributed nodes at run-time, WiDS moves predicate testing to a slow replay stage. WiDS represents execution of a system as a sequence of events and checks assertions at event granularity. An event can be a timer expiration, message receive, scheduling of a process, or a synchronization event.

3.5.4 Desktop search

DejaView provides a transparent low-overhead recording of a user's desktop [107]. A user can browse, search, or interact with the desktop by resuming execution from a nearest checkpoint. To index the execution history, DejaView uses Linux accessibility interfaces, which can capture text displayed on the screen. DejaView relies on a display virtualization to record low-level display commands [12]. Application and file-system state is captured with incremental checkpointing [137].

3.5.5 Automated problem diagnosis

Chronus automates search of a critical configuration change, which possibly leads to a system malfunction [179]. Chronus takes a user-provided software probe which can verify that the system is in correct state. Using this probe, Chronus performs a binary search instantiating the system from a recorded disk state. For every check, Chronus boots the system under control of a Denali virtual machine [180]. To keep track of a past state, Chronus relies on a Peabody time-traveling disk [95].

3.5.6 Forensic analysis

ReVirt is one of the first systems advocating that replay is not only a useful debugging mechanism but also a powerful system analysis tool [64]. Originally, ReVirt envisions two types of analysis: passive and active. Passive tools passively observe system state during replay. For example, ReVirt can replay a graphical output of the X server screen. Active tools run inside a guest virtual machine and examine its state. Potentially, each interaction of a tool changes virtual machine state and execution of the system, forcing replay to stop. However, since the same state can be replayed multiple times, active tools can perform analysis actions inside the system iteratively.

BackTracker is a tool aimed to automatically identify a sequence of steps inside an intrusion attack [103]. BackTracker runs as a part of the ReVirt platform and collects information flow across processes, files, and filenames. Although BackTracker can be implemented separately from ReVirt, ReVirt is used to aid the analysis process by replaying execution of a system.

IntroVirt uses the ReVirt engine to check for past security intrusions [97]. IntroVirt checks security predicates during replay trying to find if one of the recently announced vulnerabilities was already exploited in the system. IntroVirt relies on virtual machine introspection [75] to trigger predicate checks, and inspect the state of the system.

Taser compares execution of a system on tainted and untainted data to discover dependencies between sessions of a multisession attack [154]. Taser leverages the idea that a difference between tainted and untainted execution with a high probability indicates that execution depends on a tainted data. Therefore, there is a dependency between attack

sessions which access this tainted data. Taser relies on deterministic replay to compare execution of tainted and untainted replicas.

3.5.7 Replication and live migration of virtual machines

Bressoud and Schneider suggest the use of deterministic replay for a full-system replication [27]. The Hypervisor system runs two copies of a system on two different processors of a multiprocessor machine. It delivers an identical set of nondeterministic events to both replicas.

CR/TR-Motion applies deterministic logging and replay to migration of virtual machines [116]. Typical solutions implement live migration by precopying a system's state between machines. A set of dirty pages is iteratively transferred between two machines until it becomes smaller than a certain threshold value [45]. After that, migration completes by suspending the virtual machine and transferring the final dirty set. CR/TR-Motion argues that it is possible to reduce the downtime of the final step by recording and replaying execution of the system between the nodes instead of transferring a set of dirty pages.

3.5.8 Trace generation

CITCAT is probably the first system to rely on deterministic replay for generation of an unperturbed instruction trace of a long realistic workload [148, 147]. CITCAT uses a hardware logging mechanism to observe all nondeterministic events on a cache bus. From the cache-filtered log, CITCAT extracts initial CPU state, interrupts, external I/O, and DMA events. Then, CITCAT uses a processor simulator to replay execution of the system and collect the instruction trace.

In contrast to CITCAT, iDNA requires no hardware support for trace collection. Instead, it uses a dynamic binary translation to deterministically log execution of a user-level program [20]. During replay, iDNA generates an instruction-level trace, and provides interface for reverse execution, and replay debugging.

ReTrace further develops the idea that deterministic replay is an extremely effective way of collecting and compressing full-system traces [187]. ReTrace splits trace collection in two stages: collection and expansion. During the collection stage, ReTrace creates a deterministic log of a full-system execution. During the expansion stage, ReTrace deter-

ministically replays execution of a system collecting required trace information. Implemented as a part of the VMWare Workstation 6.0 virtualization platform, ReTrace is able to collect deterministic trace with a run-time overhead of only 5% and space overhead of 0.5 byte per thousand instructions. Although replay and trace generation can take as long as the original system execution, replay is perfectly parallelizable – starting from multiple checkpoints, the system can be replayed in parallel.

3.5.9 Shadow processes

Patil and Fischer are among the first to suggest the use of shadow processes for run-time checking and profiling [140]. They use compile time instrumentation to ensure replication of all nondeterministic events between two copies of a process. Shadow replica runs in parallel with the original process and performs checks for out-of bound array violations, invalid pointer accesses, and memory leaks.

SuperPin parallelizes execution of run-time dynamic analyses, but relies on binary translation to implement this technique transparently to the application [178].

TightLip replicates execution of a process to analyse propagation of sensitive data [191]. TightLip makes sensitive data available to only one replica. If execution of replicas diverges, then with a high probability, it depends on the sensitive input. TightLip uses a kernel-level implementation of a process-level replay. It interposes on all external events comparing execution of two processes. TightLip implements support for asynchronous signals by deferring them until a process exits the kernel space.

Speck uses a process-level deterministic replay to parallelize security checks on a multicore platform [135]. Speck explicitly defines three components which are prerequisite for all systems replicating execution: system support for speculative execution [134], buffering of external output, and kernel support for process-level deterministic replay.

A recent workshop paper suggests to move run-time checks (e.g., out-of-bound array access checks) to a separate shadow process running in parallel with the main application [164].

3.5.10 System replication for off-line analysis

Aftersight uses deterministic replay to decouple dynamic program analysis from the system execution [43]. Run-time overhead of many dynamic program analyses is prohibitive in a production environment. Deterministic log of a system allows off-line execution of any analysis during replay. Aftersight can operate in three modes: in parallel synchronizing inputs with the original execution; in parallel, but providing only the best effort analysis service; and fully off-line. Aftersight can run multiple analyses in parallel. It also suggests the idea that replay can be performed on a platform different from the one on which the original log is taken. For example, Aftersight relies on the QEMU whole-system emulator [18] to implement system instrumentation during replay. An obvious shortcoming of the Aftersight's approach is that analysis is not allowed to change execution or state of the system.

Tralfamadore records execution of a system on a realistic workload to provide a combined source and execution analysis over the execution trace [112]. By querying an example execution, Tralfamadore helps to understand meaning of the source code and various run-time conditions, for example, a maximum stack size during this execution. Taking a crash dump as an input, Tralfamadore compares the crash stack against stacks in the execution history. Finding a closest stack, it locates an execution path, which is similar to the one a system encountered before the crash. To capture and replay execution, Tralfamadore relies on the QEMU hardware emulator [18]. Talfamadore uses an extremely slow logging mechanism. It records the entire execution of the system with all instructions and register values.

3.5.11 Fault-resilient execution

Rx suggests a method to survive software errors by replaying execution in a slightly modified environment [145]. Rx uses the idea that many bugs are correlated with the execution environment. Modifying the environment, it is possible to avoid the bug. To change the environment, Rx changes memory allocations, timing, and user input to a crashing application.

3.6 Discussion

Deterministic replay has a long history of research, development, and applications. Surprisingly, despite an early origin of the idea to decouple analysis from execution [10], replay mechanisms neither became a standard part of the systems stack, nor a de facto mechanism for systems analysis. Several reasons might be responsible for slowing development and spread of the replay tools.

First, the lack of a clean interface boundary naturally isolating resources of individual applications in a traditional time-sharing UNIX system complicates development of a simple, efficient interposition layer. The semantically-reach system call interface forces the otherwise isolated applications to have a shared state in the operating system kernel. Building efficient replay for the system call interface is hard. Virtualization interface is a better target for implementing the general replay framework.

Second, deterministic replay suffers from the lack of adequate hardware support. Branch counting mechanisms required to replay asynchronous signals are still imprecise, which results in an unjustifiably complex branch counting logic. The need to single-step the system affects performance of the system during replay.

Finally, capturing of a complete execution is traditionally associated with the high processing and storage overheads. This is no longer true. A proper interposition boundary allows efficient recording of the nondeterministic replay. It might be surprising, but recording an entire Linux system might be sometimes less intrusive than tracing the same system with the `printf`, or `tcpdump`. Furthermore, a modest 1TB hard disk can provide enough storage space to record the execution of the system for 30 days.

CHAPTER 4

PRINCIPLES OF BUILDING REPLAY MACHINES

Techniques to log and replay state date back to the earliest computing systems. For example, in 1948, the ENIAC relied on system checkpointing to recover computations interrupted by frequent component failures [98]. Over the last few decades, many research projects have used deterministic replay for many purposes such as debugging, performance analysis, and forensics [64, 104, 20, 43, 113, 8].

Despite the obvious utility of logging and replay, and the existence of a number of prototype implementations, mainstream operating systems and virtualization platforms have largely failed to support them.[1] Why is this? Historically, the development and deployment of deterministic replay has been hindered by several factors. First, general-purpose operating systems fail to provide a clean boundary between the replay mechanism and the replayed part of the system. The resulting process-level replay mechanisms have been hindered by the complexity of the system call interface, resulting in complex solutions that often only support a subset of applications. Second, recording and replay has had a high cost in terms of storage and CPU time.

Today, these problems are much less pressing. First, virtualization is becoming a default part of the operating system stack. The virtual machine interface is narrow and supports a relatively clean separation between the replayed code and true nondeterminism in the external world. Second, virtualization makes logging fast. Virtual machines are optimized to provide the low-overhead interposition layer: high-throughput asynchronous I/O, low-latency interrupts, and fast memory management operations. Also, the abundance of storage and processor power on modern systems reduces the effective penalties

[1] Starting with version 8, VMWare Workstation abandoned support for the replay debugging functionality, which it had provided for the previous three years.

of recording and replay, making it feasible to record an entire virtual machine even if only a single process needs to be replayed [44]. Virtual machine introspection [75] provides a convenient mechanism for analyzing the target process during replay.

Despite the advantages offered by VMMs, deterministic replay is still difficult to implement. Abstractions are badly needed. The author's anecdotal experience was that it took three person-years to implement a deterministic replay engine for Xen 3.0.4. Despite the existence of earlier replay implementations (one of them in Xen[65]), reuse of code and even of strategies for implementing replay did not appear to be possible. The author of this dissertation spent a great deal of time re-analyzing sources of nondeterminism, reinventing debugging tools, and rediscovering a way to split the system into deterministic and nondeterministic parts such that recording and replay had good performance.

The thesis of this chapter is that an effective recipe—a collection of techniques and abstractions—that can serve as a practical guide for creating deterministic virtual machine replay has not yet appeared in the literature. The author believes first that record/replay is a useful part of the virtual machine toolkit and second that this recipe can substantially simplify future replay implementations, giving this technology a better chance of becoming a part of the mainstream systems software stack.

4.1 Deterministic virtual machines are hard

The basis for deterministic replay is simple: the execution of a system is mostly deterministic by default, with only occasional interruption by external nondeterministic events such as interrupts, values read from I/O ports, etc. In other words, replay can be implemented by replicating the system's initial state and then letting the CPU execute in its normal, deterministic fashion, interrupting it only to replay the stream of previously recorded external events. However, the devil is in the details. Creating a low-overhead record and a high-performance replay requires not only an uncomfortable intimacy with often-buggy hardware features such as performance counters, but also a finely tuned line between the deterministic and nondeterministic elements of the system.

Two conditions should hold for the above approach to work: (1) the determinism of execution must be preserved between nondeterministic events; and (2) all nondeterministic

input must be available for interposition during logging and replay. These conditions are surprisingly hard to enforce in a real system.

Although the virtual machine is designed to have a rigid isolation boundary, in a real system, it has a number of architectural dependencies on multiple parts of the virtual machine monitor: virtual CPU and the MMU units emulated by the hypervisor, device configuration and VM creation tools, device emulation code, etc. All of these update the state of the virtual machine through a number of different protocols. The implementation of an interposition boundary requires extension of the replay layer into the code of many components of the virtual platform.

The problem is that interposing on all of these protocols causes the complexity of the record/replay machinery to explode. Nondeterministic components, which interface with the replayed system, are implemented at different layers of the VMM: inside the hypervisor-, kernel-, and user-level execution contexts. Some components run in parallel with the replayed system, while some atomically preempt execution of the guest. Most components are reentrant, and under high load may generate nondeterministic events in an order that is acceptable for the replayed system, but meaningless for the replay engine.

All virtualization platforms share identical design and implementation principles and present a replay engine with the same set of problems. Replay needs general, reusable mechanisms, which can help its implementation: hardware instruction counting logic, general tracing primitives, which can be safely used in any execution context, locking and synchronization mechanisms to ensure sane ordering of events, and atomicity of recording with respect to the system's state, fast communication mechanisms to communicate event and control information across the entire system, and coordinate execution of all the components during replay, event and execution scheduling during replay, etc.

4.2 A recipe for deterministic replay

A VMM-based replay engine can be built around two basic mechanisms: *nondeterministic events* and *interposition functions*. Events carry the information about nondeterministic updates to the system's state, such as those that are performed by interrupt handlers, while interposition functions collect, filter, and inject nondeterministic events. For ex-

ample, when the replayed system executes an `rdtsc` instruction to read the processor's timestamp counter, an interposition function supplies the appropriate previously recorded value. To scale the replay engine to the complexity of realistic systems and workloads, it is critical that this core is built on a set of concise, well-understood principles.

4.2.1 A three-part model

The replay system can be divided into the replayed system, a deterministic execution environment, and a nondeterministic external world (Figure 4.1). In contrast to a simpler two-part model, i.e., replayed system and external world, the three-part model reflects the practical need to extend the replay boundary well outside of the virtual machine itself. The three-part model reflects the practical goals of (1) reducing the amount of nondeterminism, and (2) reusing the functionality of existing deterministic or semideterministic components to simplify the implementation of the replay layer.

The replay implementor's job is to classify every interaction between the replayed VM and its environment as belonging to one of the following four categories.

4.2.1.1 Deterministic updates

The state of the replayed system and its execution environment evolve together by updating each other. Figure 4.1(a) shows an interaction where the update originates inside the replayed system and changes the state of the deterministic environment. This corresponds to, for example, a hypercall. Analogously, the hypervisor can update the state of the replayed guest. Deterministic interactions do not need to be logged, but they do need to be reexecuted during replay to ensure that both parts of the system evolve their state in the same way that they did in the original recorded run. Deterministic interactions can also serve as useful points for comparison between the original and replay executions in order to debug the replay engine.

4.2.1.2 Synchronous events

Reading a time stamp counter is an example of accessing nondeterministic data from otherwise deterministic code; this is shown in Figure 4.1(b). To ensure that the replayed machine follows the original execution path, synchronous events are replayed "in-place."

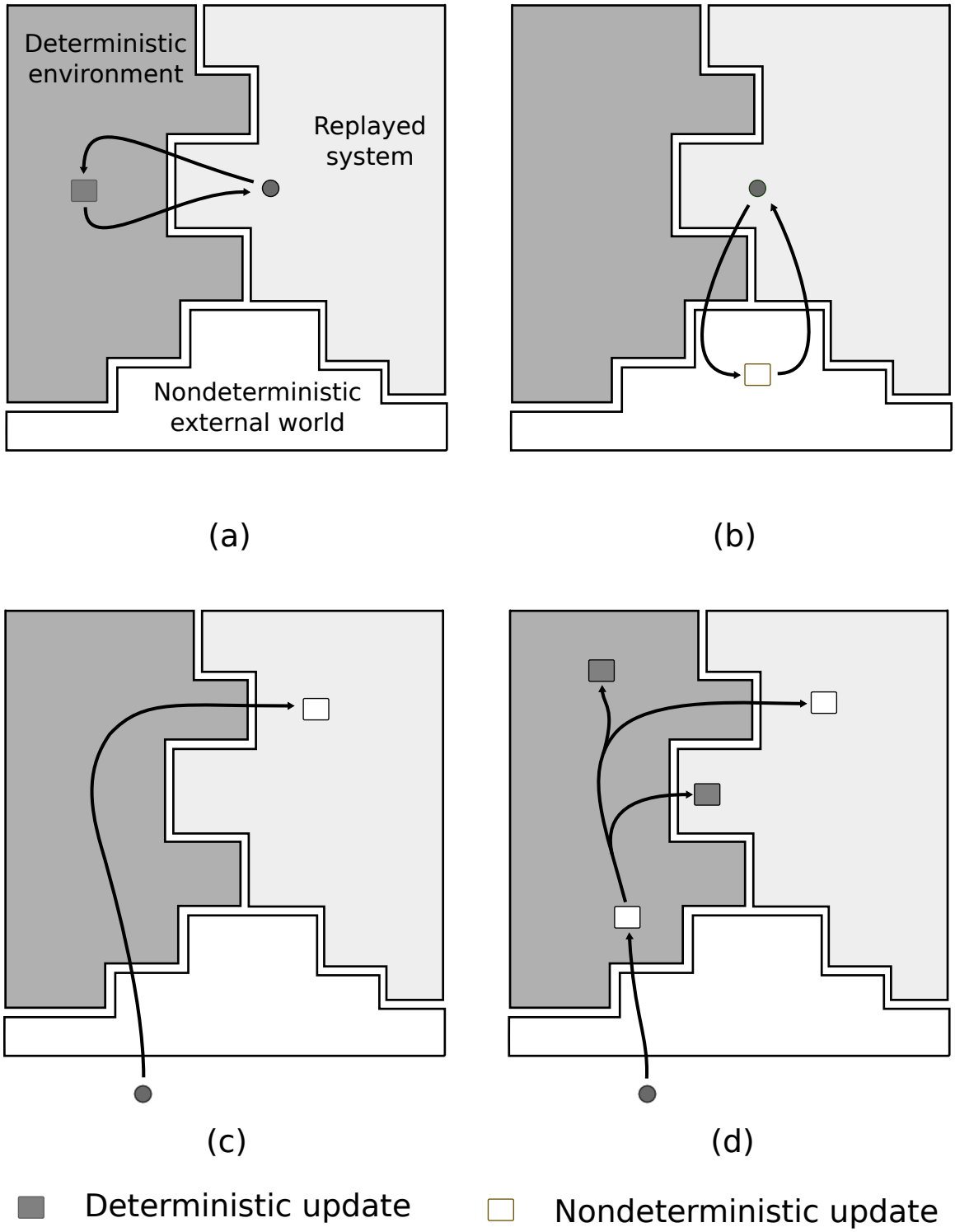

Figure 4.1. External world, deterministic environment, and replayed system

Replay interposition primitives query the replay engine and return to the system the value of the nondeterministic variable that was observed during the original run.

4.2.1.3 Asynchronous events

Asynchronous events, depicted in Figure 4.1(c), represent external updates to the replayed system. These include interrupts and updates to shared memory from virtual device drivers running in parallel with the replayed system. In contrast to a synchronous event—where the replay machine effectively schedules the state update itself—an asynchronous event must be replayed in instruction-accurate fashion by the execution scheduling loop.

While asynchronous events from the replay log must be delivered on schedule, an important role played by the replay engine's interposition functions is to prevent unscheduled asynchronous events from updating the replayed system's state. These undesirable events are generated by the large parts of the hypervisor and device drivers that were not modified to support replay and are therefore unaware that they are dealing with a replayed system. The interposition functions serve as a firewall that prevents nondeterminism from leaking into the replayed system.

4.2.1.4 Grouping events

Figure 4.1(d) illustrates the common case where an asynchronous event triggers execution of a function that performs multiple deterministic and synchronous updates. For example, an interrupt event updates flags, registers, and stack of the guest system. While it is possible to record all these updates as asynchronous events, it is easier and more efficient to record a single asynchronous update, and treat the rest of updates as synchronous events originating from the code of the handler. Of course, the replay system must ensure atomicity of the entire handler. In many cases, this is easy, since the hypervisor is already designed to make sure that the code of the interrupt handlers is atomic.

4.2.2 Replay building blocks

To implement the replay interposition boundary, we utilize a coarse-grained plan of action inspired by the three-part model: for synchronous events, the interposition functions record and replay events "in-place"; for asynchronous events, they record and firewall

events “in-place,” but replay is done from the execution scheduling loop.

Virtual devices are not inside the deterministic environment. However, since their execution during replay is driven by requests from the replayed system, they are “nearly deterministic”—the only nondeterministic aspect of their execution is the time at which they respond. *Determinizing proxies* are used to interpose on the communication protocol between the replayed system and the device, ensuring that updates between them are propagated in a deterministic way. The following parts constitute the main building blocks of the replay engine (Figure 4.2):

4.2.2.1 Pluggable interposition primitives

Tracing functions implement a general logging, replay, and filtering interface for nondeterministic events. They are designed to work correctly in different contexts of execution, and provide an identical interface at multiple layers of the software stack. The tracing functions rely on the fast data bus to implement an allocation-, lock-, and block-free tracing on the critical path.

4.2.2.2 Data and control bus

Implemented as multiple lock-free shared memory queues, the data bus provides a communication primitive that connects all components of the replay engine at all levels of the system. The interposition primitives attach to the data bus to relay the stream of nondeterministic events to and from the logger and replay daemons. The control logic is used to request replay of the specific event from the determinizing proxies. When a proxy receives a replay request, it replays communication with the proxied device up to the requested point.

4.2.2.3 Determinizing proxies

Determinizing proxies ensure determinism of the communication protocol between the replayed system and virtual devices. A virtual device accesses the state of the guest system through two mechanisms: (1) memory remapping, and interrupt signaling hypervisor calls; and (2) a region of shared memory. Determinism of the hypervisor calls is ensured by the interposition layer inside the hypervisor. To ensure determinism of direct memory updates,

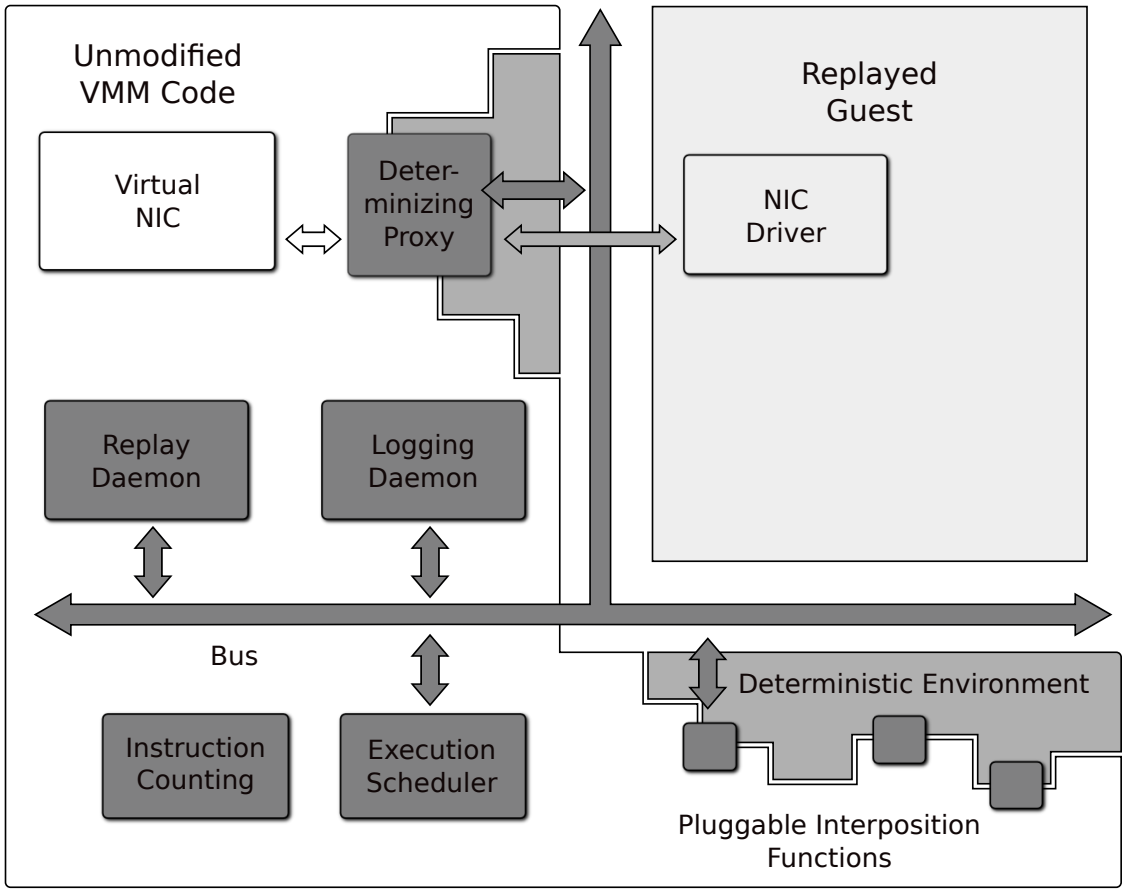

Figure 4.2. Seven components of the replay engine: pluggable interposition functions, data bus, logging and replay daemons, execution scheduler, instruction counting logic, and determinizing proxies.

the determinizing proxy inserts itself between the guest system and the virtual device, and mirrors all updates to and from the guest system in a deterministic way.

Some devices, e.g., network and console, require replay of the device I/O payload. For the console device, its proxy replays the console input itself. However, for the more complex network device, the proxy avoids emulation of the full device protocol. Instead, it replays the network payload into the device by using the device functions. The network device had to be modified to prevent it from dropping replayed packets.

4.2.2.4 Logging and replay daemons

The logging daemon is responsible for flushing the data bus to a permanent store. The logger runs on a separate CPU, attaches to the data bus, and makes sure that all processing

of the nondeterministic stream is done off the critical path. The tracing functions, which are on the critical path, resume processing immediately after putting events on the data bus. The replay daemon provides the stream of nondeterministic events during replay.

4.2.2.5 Instruction counting logic

The position of each event in the replay log is defined by the number of instructions executed since the system's start. The challenge is to implement accurate instruction counters on top of realistic hardware performance counters, which were originally intended to support performance analysis of the compiler and low-level system optimizations. The problem with hardware performance counters is that they are only statistically correct. From the point of view of a replay engine, "statistically correct" is not hugely better than "wrong."

4.2.2.6 Execution scheduler

A replayed VM is induced to reproduce the recorded execution path by injecting each nondeterministic event at its instruction-accurate position in the instruction stream. Even small errors tend to lead to divergence. The execution scheduler implements controlled execution of the system between nondeterministic events during replay. For synchronous events, the scheduling engine lets the system run until it reaches the point in execution at which it asks to replay that specific event.

Replay of asynchronous events uses hardware branch counters to get close to the required point, and then single-steps the processor. First, the execution scheduler configures branch counters to overflow and raise a nonmaskable overflow interrupt around 100 branches before the original event takes place. This is done to address a hardware delay in receiving the interrupt. After execution of the system is preempted with the overflow interrupt, the execution scheduling component continues execution of the system in a single-step mode. Reaching the target place in the execution, the replay engine replays the asynchronous event, and continues execution by scheduling execution of the system to the next nondeterministic event.

4.3 Scaling development

Several debugging and analysis tools are essential for development of deterministic replay. These tools are critical for scaling the replay engine to support replay of the full-featured guest systems on long, I/O intensive runs with a large amount of frequent nondeterminism. First, to aid the manual analysis of nondeterminism, this project relies on a run-time *page guarding* mechanism, which detects if an uninterposed nondeterministic event "leaks" into a guest system. Second, to detect divergence between the original and replay runs, this project relies on an off-line execution and trace comparison tool. Third, to detect divergence in the run-time state of the hypervisor, e.g., the hypervisor returns a different result from the hypercall during the original and replay runs, this work implements a general run-time state comparison mechanism.

4.3.1 Automatic analysis of nondeterminism

The implementation of any replay system requires identifying all sources of nondeterminism. This is trivial when nondeterminism is restricted to a high-level messaging interface [16, 56, 26]. A many-hour analysis is required to design a replay interface for a system with a highly optimized low-level interface that involves communication via shared memory, asynchronous interrupts, virtual memory management primitives, virtualization of time, and sensitive machine instructions.

It is essential to aid the manual analysis of nondeterminism with a run-time mechanism capable of automatically detecting, when an undetected nondeterministic event leaks into a guest system. The main insight behind the *page guarding* approach used in this work is the observation that it is possible to model the majority of nondeterministic events as updates to the memory of the guest virtual machine. To detect a nondeterministic event, XenTT guards all the pages shared between the Xen virtual machine and the guest system. The guard mechanism is implemented with the help of the hardware page-level memory protection mechanisms.

4.3.2 Off-line comparison of original and replay runs

The page guarding technique enables detection of a large subset of all nondeterministic events at the level of memory updates. Not all the events can be represented as updates to

the memory of the guest, and therefore, the page-guarding technique leaves some sources of nondeterminism undetected; e.g., a nondeterministic input can update the state of the guest system in registers, update state of the hypervisor or device drivers, etc. The execution of the guest system diverges during replay without any hints for the source of determinism violation.

To detect violations of determinism, a deterministic replay engine needs to implement a powerful off-line execution comparison tool. This work extends the Xen hypervisor with support for recording execution of the system with the hardware Branch Tracing Store (BTS) facility provided by the Intel CPUs [51]. BTS records all branches taken by the CPU in a memory buffer. The content of the BTS buffer is traced in the nondeterministic log. To compare executions of the system, a comparison tool reads the nondeterministic log, and resolves machine addresses into readable symbol names. XenTT uses the GDB debugger as a library to parse and resolve the guests symbol information into readable symbol names. The human readable files are then compared with one of the text comparison tools, e.g., `vmidiff`.

4.3.3 Run-time comparison of the hypervisor's state

To detect divergence in the state of the hypervisor, XenTT implements a general run-time state comparison tool. During the original run, any part of the hypervisor's or system's state can be logged as a nondeterministic event. During the replay run, a special event flag asks to compare the state saved in the logged event with the state of the system at the moment when event is replayed.

4.4 Discussion

This section develops a set of general design principles which enable construction of deterministic replay engines for various virtualization platforms. XenTT is an example implementation which follows these principles. Built on top of Xen, XenTT has only minimal dependencies on the hypervisor.

CHAPTER 5

OVERVIEW OF THE XEN ARCHITECTURE

The Xen virtual machine platform consists of two main parts: (1) the hypervisor; and (2) a special virtual guest virtual machine used for hosting device drivers, and provide a general administrative infrastructure for device discovery, virtual machine creation, etc.

The Xen hypervisor is a minimal microkernel which provides a hardware-like interface sufficient for running traditional operating systems inside isolated virtual machine containers. Xen implements a small subset of traditional operating system abstractions: virtual machine scheduling, page-level memory management and address spaces, memory sharing across domains, and interrupt-like notification channels. In contrast to many microkernels, the Xen hypervisor does not implement any IPC mechanism besides a single-bit, interrupt-like notification channel. Guest virtual machines are free to implement any IPC mechanism on top of two primitives: the shared memory mechanisms, and the event channels.

The hypervisor does not implement device drivers, except the ones which are required for ensuring isolation of virtual machines (timer, PCI-bus), and for performing boot-time serial line output. Traditional device drivers are hosted inside one or several privileged virtual machine guests called *device driver domains*. Xen enables direct access to a hardware device for such privileged domains via a PCI passthrough mechanism. In a default Xen setup, a single Linux virtual machine called *Domain 0* hosts all device drivers, and multiplexes individual devices between multiple guests (Figure 5.1).

A typical guest virtual machine, called *Domain U*, starts with four virtual devices: console, Xenstore, disk, and network. A guest virtual machine does not have access to real physical devices. Instead, Xen provides the guest with a notion of a virtual device, which is implemented with two components – a backend and frontend split device. The backend and frontend devices run in the Domain 0, and the guest virtual machines, and

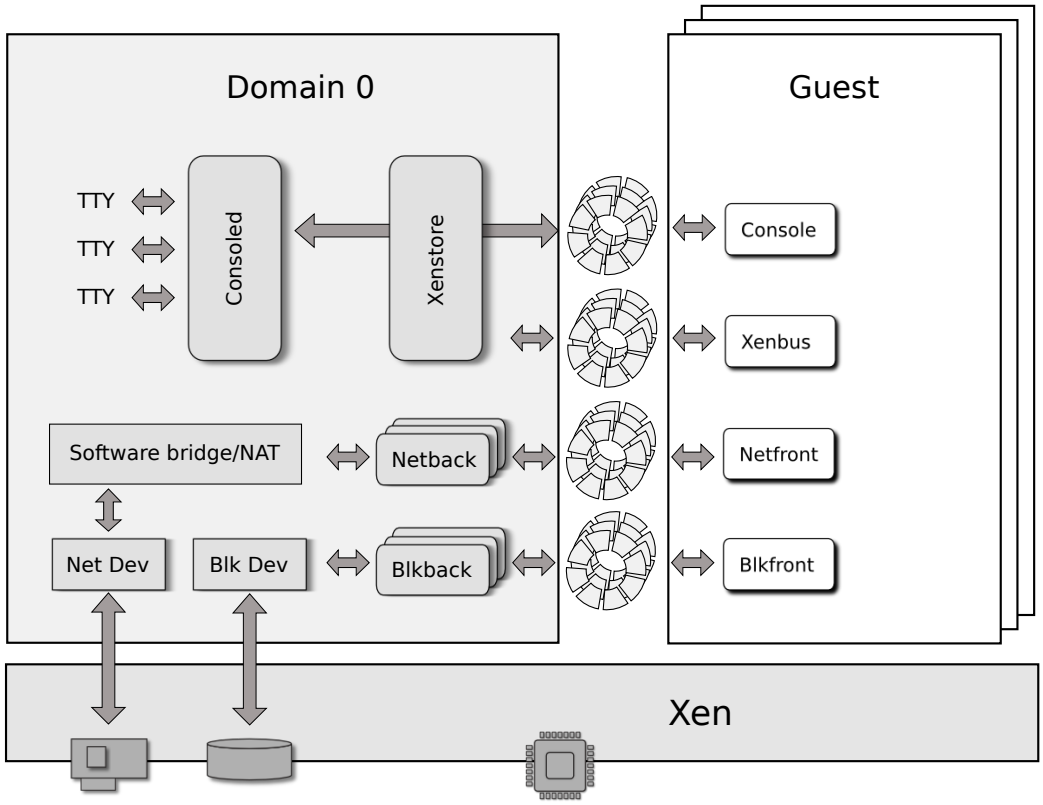

Figure 5.1. General architecture of Xen

work as a proxy-stub pair, which provides access to a physical device running inside a privileged device driver domain.

The frontend part of a split device runs as a normal device driver inside the guest's kernel. The difference between the frontend and a regular device driver is the fact that the frontend does not manage an actual physical device. Instead, the frontend serves as a stub, which routes all device requests to the backend part of the split driver.

A backend device driver runs inside a device driver virtual machine (in a default Xen setup, this is Domain 0). Similar to the frontend, the backend driver is a kernel module running inside the kernel of a device driver virtual machine. The backend does not pretend to be a device driver for the kernel, since it stays transparent to the Domain 0, and does not serve any requests from the Domain 0 kernel. Instead, the backend serves as a proxy, which dispatches all requests from the frontend device, and sends them to an actual physical device driver running in the same Domain 0 kernel. The physical device is controlled by a

normal device driver; i.e., since normally Domain 0 is a Linux virtual machine, the physical device is controlled by a Linux device driver. The Xen virtualization platform reuses the device drivers implemented for the Linux kernel, and potentially for other operating systems if they are started as device driver virtual machines.

A frontend-backend device pair relies on the shared memory and event channel communication primitives to establish a high-throughput cross virtual machine communication mechanism. A typical way to implement this communication channel in Xen is to share a page of memory containing a lock-free, producer-consumer buffer, through which the backend and frontend exchange information about device requests. A shared ring mechanism works well for implementing a throughput-oriented communication mechanism, and suits well the need of the split device drivers.

5.1 Paravirtualized hardware interface

The original Xen architecture was based on a powerful idea of *paravirtualization* [181]. [1] In contrast to virtualization approaches, which aim to run unmodified guest virtual machines in a completely transparent fashion, Xen leverages paravirtualization to make the guest kernel aware of the fact that it runs on top of the hypervisor.

Originally, explicit changes to the guest kernel, required as part of the paravirtualized interface, were critical for enabling significant simplification of the hypervisor, which no longer requires a complex binary translation engine. Later, Xen was extended to support execution of unmodified guests through the hardware virtualization. Paravirtualization, however, remains an important part of the virtualization spectrum [185].

XenTT only supports replay of paravirtualized guest systems. It is important to understand how the hardware interface is exposed on top of Xen for the guest kernel.

5.1.1 Hypercalls

The majority of communication between guest virtual machines and the hypervisor happens via a clean hypercall interface. The Xen hypervisor defines a set of functions (the hypercall interface), which provides guest virtual machines with an explicit way of

[1] Paravirtualization was a well-known principle used in many hypervisors before Denali; however, the term was introduced in the work by Whitaker et al.

configuring its execution environment provided by the hypervisor. The hypercall interface is implemented similar to a traditional system call interface of the Linux kernel; the guest virtual machine invokes a special software interrupt (interrupt 0x82) passing the hypercall number in one of the registers. Hypercalls reflect the low-level, CPU-like nature of the Xen interface which is exported to the guest systems; e.g., they provide a way to yield CPU when idle, configure the interrupt descriptor table, configure and manage the guest's virtual memory management unit, send event notifications, etc.

5.1.2 Interrupts and exceptions

The Xen hypervisor provides the guest system with a hypercall for configuring the virtualized interrupt descriptor table (`trap_info` table). The `trap_info` allows the guest system to specify a function pointer for each exception vector of the x86 hardware interface. A typical Linux guest configures the lower exception vectors 0-19, and the interrupt 0x80, which is used for the Linux system call interface. Xen allows direct exception for the interrupt 0x80; this way, the guest system can perform a system call without exiting to the hypervisor.

To reflect the trap into the guest kernel, the Xen hypervisor creates a special trap invocation stack (trap bounce frame), and redirects control to the handler specified in the `trap_info` table.

5.1.3 Memory management

Xen allocates a set of machine pages for the guest virtual machine when it is created. The guest uses these pages as its physical memory. To optimize execution of the pravirtualized guests, the Xen hypervisor exposes actual machine addresses (machine frame numbers (MFNs)) of the machine pages allocated for the guest to the guest system. To ensure isolation of the guest virtual machines, they cannot access their hardware page tables directly. Instead, the guest system manipulates its virtualized page tables through a set of memory management hypercalls.

5.1.4 Memory sharing

The Xen hypervisor implements memory sharing via two primitives: grant tables, and grant hypercalls. The grant table is directly accessible to the guest system. The guest uses flags in the grant table to grant access permissions to a particular page to a receiver domain. If permissions are set up, the receiver can invoke a grant hypercall and either map or transfer a page from the sender domain.

5.2 Intervirtual machine communication mechanisms

Unlike traditional microkernels, Xen does not implement a high-level IPC mechanism. Instead, the Xen intervirtual machine communication is built on top of the Xen memory sharing primitives. The Xen hypervisor implements intervirtual machine communication with two simple primitives: shared memory, and interrupt-like event channels (Figure 5.2).

The semantics of the memory-based IPC mechanisms is not restricted in any way by the Xen architecture; the guest virtual machines follow the following communication paradigm. A shared page contains a producer-consumer ring to implement a lock-free queue of IPC request-responses between two virtual machines. Some low-bandwidth devices, e.g., console, and Xenstore, use a shared ring to transfer the actual data. Throughput-intensive devices like disk and network use a shared ring to communicate only references to the pages with actual data.

This simple, channel-oriented IPC is designed for a low-level, throughput-intensive, long-lasting device-like communication. Compared to a traditional lightweight microkernel message IPC, these channels are relatively expensive to setup. However, the expectation is that communicating devices are initialized only once during the system boot. The asynchronous nature of this communication primitive is optimized for high-throughput, bulk data transfer. Multiple requests are batched in the ring – Xen's memory mapping hypercalls also support batch updates of guest's memory. Furthermore, shared rings minimize the amount of event channel notifications, and thus minimize the number of intervirtual machine interrupts. In a ring receive loop, the receiver is expected to check if the ring contains more request. This way, an expensive asynchronous notification via an event channel is required only when the sender needs to wake up the receiver.

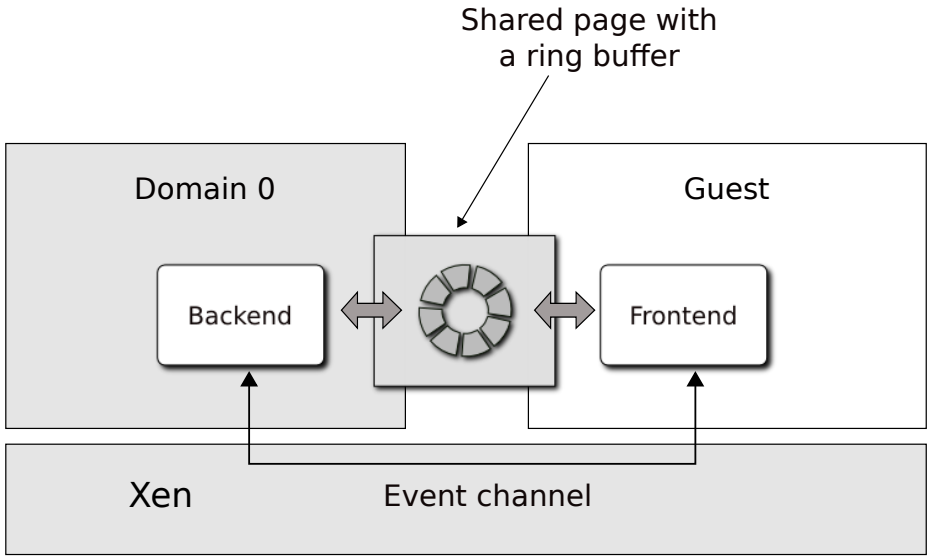

Figure 5.2. Shared rings and event channels.

5.2.1 Shared rings

A Xen shared ring is implemented as a page shared between two virtual machines. A logical ring consists of a shared buffer and three sets of pointers: two sets are private for producer and consumer, and one is shared between them in the shared ring page (Figure 5.3). The ring buffer has a size equal to the power of 2 and occupies less than a single 4K memory page.

The ring communication is organized in the following way: one end of the communication channel produces requests and consumes responses (typically a frontend device); the other end consumes requests and produces responses (typically a backend device). Initially, the ring is empty and all pointers point to the start of the ring buffer. Then, the frontend as a request producer adds requests to the ring. Internally, the frontend maintains a private request producer pivot pointer (`req_prod_pvt`), which is incremented every time when the frontend adds a new request to the ring.

To push all added requests at once, and notify the backend, the frontend advances the shared request producer pointer (`req_prod`), by making it equal to the private request producer pivot pointer (`req_prod_pvt`).

To check if there are any new requests in the ring, the backend compares the shared request producer pointer (`req_prod`) with its private copy of the request consumer pointer

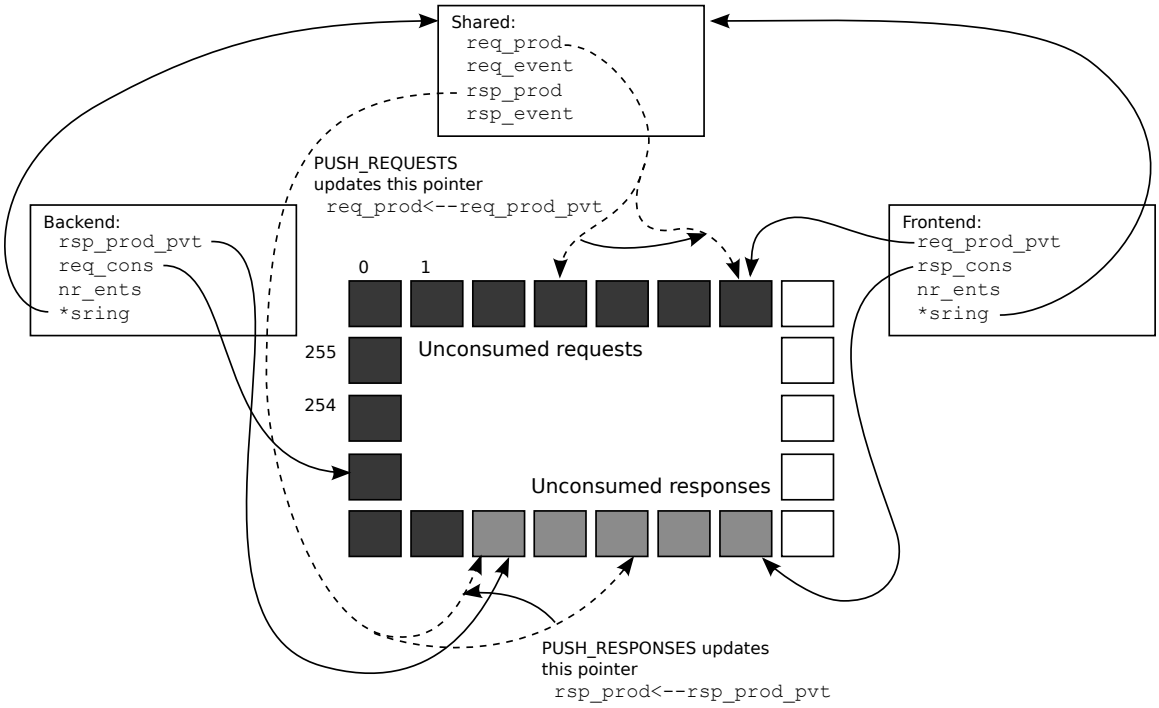

Figure 5.3. Low-lever ring details.

(`req_cons`). The `req_cons` pointer points to the last processed request. One at a time, the backend fetches requests from the ring. For each request, it creates a response which typically acknowledges that the request is received. To provide the frontend with an ability to match requests and responses, the backend saves the request identifier into the response. Similarly to the frontend, the backend maintains a private pointer to the last consumed response (`rsp_cons`).

5.3 Discussion

The Xen virtual machine platform exposes an interface which is built to model an interface of a hardware CPU. To replay execution of the guest virtual machines, XenTT implements interposition interface for both the paravirtualized CPU, and the split device driver communication interfaces. In the later sections, this dissertation discusses how XenTT benefits from the low-level nature of the Xen CPU interface, and from the general device communication model.

CHAPTER 6

IMPLEMENTING THE BUILDING BLOCKS

This section describes implementation of the building blocks required for constructing a deterministic replay engine according to the principles presented in Chapter 4. XenTT – a replay engine for the Xen virtual machine monitor – consists of four main building blocks and a high-bandwidth communication channel across them (Figure 6.1):

- **Event interposition:** The event interposition layer implements logging and replay of the low-level virtual machine interface exported by the Xen hypervisor. Interposition primitives are designed to introduce a minimal overhead on the critical execution path of the system. A lightweight logging operation requires one to read the hardware state of the system and put a logging record in a lock-free, shared-memory buffer for asynchronous processing by the user-level logging daemon.

- **Logging and replay daemons:** User-level logging and replay daemons process the log of recorded events, committing it to a stable storage.

- **Device daemon:** To log and replay communication of virtual devices, XenTT relies on the fact that all Xen devices use a uniform shared memory interface for connecting guest and host device drivers. The device daemon implements a *determinizing proxy*.

- **Replay coordination:** Finally, replay coordination mechanisms ensure controlled execution of the guest system between a pair of nondeterministic events. These mechanisms include execution scheduling, branch counting logic, single-step execution, replay of synchronous and asynchronous events, and CPU branch tracing.

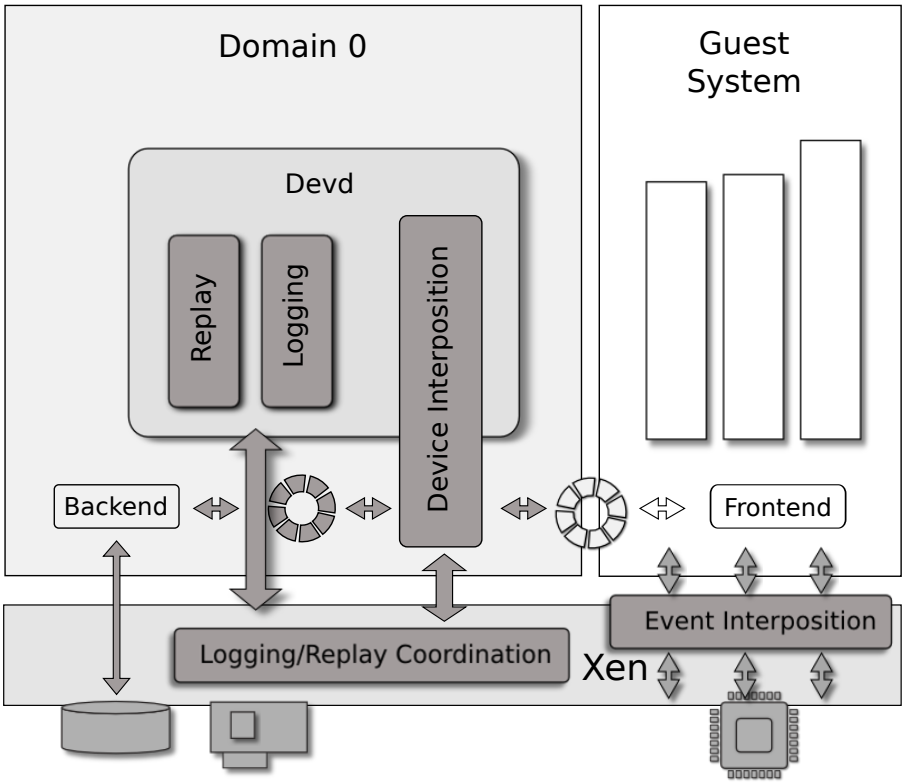

Figure 6.1. General architecture of Xen

6.1 Lightweight interposition and logging

Building a lightweight interposition layer is an exercise in developing a fast inter-process communication mechanism. The logging code resides on the critical processing paths of a guest virtual machine: inside interrupt and exception handlers, in the I/O paths of the device drivers, and on the exit path from the guest to the hypervisor. A typical length for these critical paths in Xen is around 4,000 CPU cycles on a modern 2.4 GHz processor. To be performance-transparent, XenTT's interposition code can introduce only 10–20% overhead (400–800 cycles) on these paths.

The main principle for implementing a fast interposition layer is to offload all tracing, processing, and saving of the trace data from the critical path. XenTT implements this principle by utilizing a three-stage logging pipeline: tracing functions, ring channels, and a logging daemon (Figure 6.2).

XenTT relies on a range of fast communication techniques for implementing its lightweight

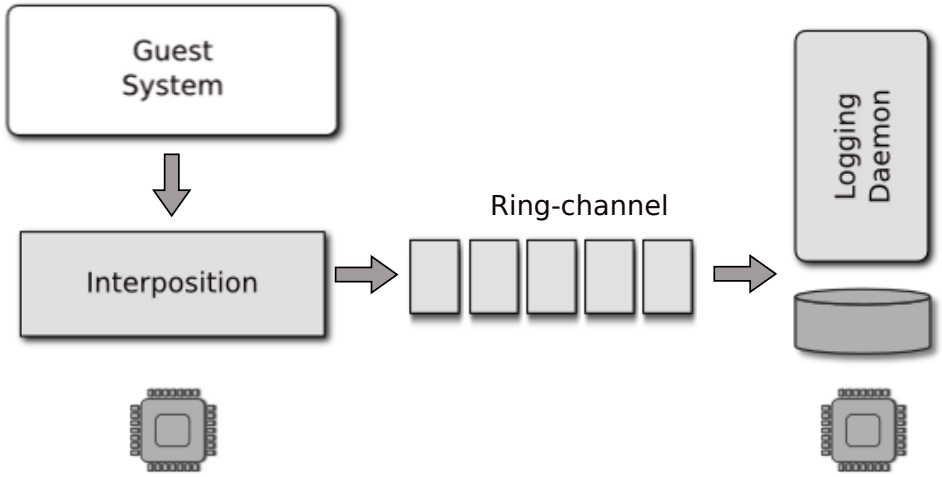

Figure 6.2. Lightweight interposition pipeline.

interposition. XenTT has no memory allocations, no data copying, and no locks on the critical path. At the heart of XenTT's interposition layer are two primitives: (1) a general marshaling mechanism and (2) lock-free, high-bandwidth communication channels. Together, these primitives provide a mechanism for a zero-allocation, zero-copy, and lock-free interposition and recording path from the hypervisor to a persistent store.

6.1.1 Event marshaling

Logging code records information about a nondeterministic event and resumes processing on the critical path. XenTT avoids memory allocation and copying on the critical path by unifying memory management and event transfer in a single communication mechanism. XenTT relies on a set of general marshaling primitives to allocate the memory for a new event record straight into the communication channel and then serialize the event's data into that memory.

XenTT represents an event as a structure with a fixed-size header, several 32-bit registers, and a length field that describes the size of the data section (Figure 6.3). If the event-specific data fit in registers, the event occupies one record in a ring-channel buffer. Otherwise, the marshaling code sequentially allocates records from the ring-channel buffer.

XenTT's marshaling code maintains the invariant that each event spawns one or more event-channel records. This is critical for implementing zero-allocation. In the ring chan-

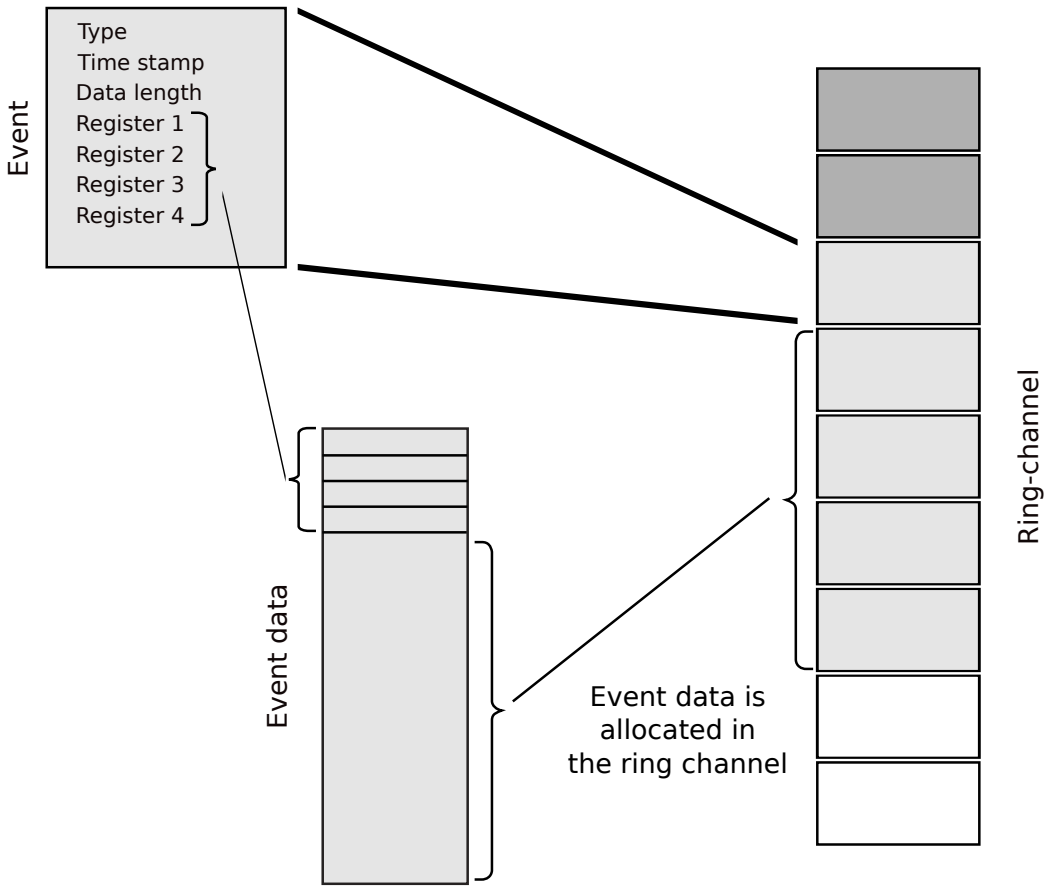

Figure 6.3. Event allocation in the ring channel.

nel, the pointer to the next record in the ring always points to the next available record, which can be allocated by just incrementing this pointer.

6.1.2 Lock-free, nonblocking communication channels

To remove events from the critical processing path, XenTT relays them to a logging daemon, which asynchronously flushes them to a disk. XenTT implements a fast, bandwidth-oriented, and lock-free mechanism for this relay: a *ring channel*.

A ring channel provides a general producer-consumer ring, implemented as a continuous region of virtual memory shared between the sender and receiver. The channel memory contains a header, describing the shape of the channel, and a buffer for payloads. A ring channel is both lock-free and nonblocking; allocation, send, and receive operations are done with a single update of the producer or consumer pointers. To avoid blocking, a ring

channel provides flow control and tries to notify the receiver via an out-of-band mechanism when new records are available in the channel. For channels in which delay does not matter, ring channels notify the receiver only if the channel becomes dangerously full.

6.2 Atomicity of recording event and timestamps

To ensure atomicity of recording a nondeterministic event and the exact position of the event in the instruction stream of the system, XenTT preempts and suspends execution of the guest system.

6.2.1 Atomicity of local events

To observe and record all nondeterministic events in the CPU virtual machine interface, the XenTT interposition layer ensures that all nondeterministic events are privileged, e.g., result in a hardware exception which exits to the hypervisor. Since XenTT implements the recording layer along the Xen virtual machine interface, most nondeterministic events, local to the physical CPU on which the guest system is running, are already privileged, due to the fact that the Xen hypervisor needs to interpose on them to implement virtualization of hardware.

For example, events like privileged instructions, hypercalls, and exceptions do not incur a cost of an additional exit, since the guest system already exited to the hypervisor through one of the exception vectors. One exception from the above rule is the read timestamp counter instruction (`rdtsc`), which is unprivileged in the default Xen architecture, but needs to be made privileged by the XenTT, as XenTT needs to observe and record the time value reported to the guest.

6.2.2 Atomicity of cross-CPU events

To ensure atomicity of recording asynchronous updates to the state of the system originating on a different physical CPU, XenTT needs to preempt execution of the guest system. The following two design choices are possible.

One option is to record an event on the same physical CPU on which an event originates. It is possible to pause the domain, record the event in the log on the local physical CPU, and resume execution of the guest. The Xen pause function (`domain_pause`) preempts

execution of the domain, and changes its scheduling flags, preventing it from running again. Unfortunately, the domain pause is a relatively heavyweight operation due to locking and scheduling queue manipulations. Pausing domain is only a reasonable choice if several nondeterministic events are recorded at once.

Empirically, it was found that a more efficient alternative for recording cross-CPU events is to migrate recording of an asynchronous event between two physical CPUs – from the physical CPU, on which it originates, to the CPU, on which the guest system is running. XenTT migrates execution of the recording function between the CPUs by using the `on_selected_cpus` primitive in the Xen hypervisor. `on_selected_cpus` implements a way to invoke a function on a specific CPU. `on_selected_cpus` request invocation on another CPU with an interprocessor interrupt (IPI). IPI preempts execution of the guest system, and invokes the requested function in the context of the IPI interrupt handler. `on_selected_cpus` allows XenTT to avoid any domain locking on the recording path. Recording functions only need to control preemption.

6.2.3 Branch counter caching

An additional overhead involved in recording of nondeterministic events is introduced by frequent accesses to the relatively slow hardware branch counter register. To minimize this cost, XenTT implements a counter caching logic. The hardware counter is accessed only once for each exit from the guest into the hypervisor.

6.3 Pluggable interposition functions

XenTT's interposition functions are designed to follow a single general pattern. This pattern implements an interposition logic, which makes sure that events of any scheduling type, e.g., in-place, optional, or asynchronous, are recorded and replayed correctly.

The interposition code for the user traps is shown in Listing 6.1. Lines 4 – 5 declare the new nondeterministic event of the type `TTD_GUEST_TRAP`, and the in-place flag `TTD_EVENT_FLG_IN_PLACE`. If a domain is running during replay (Line 7), the event is recorded with a general marshaling function `ttd_trace_reg2`, which takes two data registers as arguments. Line 9 asks the execution scheduler to replay all current events.

If an event has a side-effect on the state of the system, e.g., update of the guest's

```
1  int trace_guest_trap(struct vcpu *v, int trapnr,
2                       struct cpu_user_regs *regs, int use_error_code)
3  {
4      ttd_event_t event = {.event = TTD_GUEST_TRAP,
5                           .flags = TTD_EVENT_FLG_IN_PLACE};
6
7      if( ttd_is_replayed_domain(v->domain) ) {
8          ttd_trace_reg2(event, v, NULL, 0, trapnr, use_error_code);
9          return ttd_replay_current_events(v, regs, &event);
10     };
11
12     /* Add event emulation and preemption control here */
13
14     if( !ttd_is_timetraveling_domain(v->domain) )
15         return 0;
16
17     ttd_trace_reg2(event, v, NULL, 0, trapnr, use_error_code);
18     return 0;
19 }
```

Listing 6.1. Example of the interposition code.

registers, or a copy-user operation, it is performed in Line 12. Note, that event emulation must be done atomically with recording of the event to the log; therefore, proper preemption locks must be acquired in Line 12.

If a domain is not recorded for replay (Line 14), the interposition code can exit right away after emulating the event. If, however, it is a recorded domain, the event is logged to a permanent event log with the same marshaling function `ttd_trace_reg2` (Line 17).

6.3.1 Replay of in-place events

For recording of the in-place events, the interposition code creates an event with the `TTD_EVENT_FLG_IN_PLACE` flag set. During replay, this in-place event must be replayed at exactly the same point inside the hypervisor's code, at which it is interposed. Therefore, the interposition code (Line 9) passes the event to the event scheduler, which is responsible for replaying current events, as a request to replay this specific event at this point of execution. If the event scheduler fails to replay a requested event, it signals that execution of the guest system diverged from the original run.

If the interposition code interposes on an asynchronous event, the goal of the interposition template is to block all accesses to the state of the system. For example, during replay, the interposition code blocks a number of asynchronous updates to the wall-clock time values in the `shared_info` page. For an asynchronous event, Line 9 passes NULL as the requested event. If the interposition code is invoked at the same point at which the event occurred during the original run, the event and its effect is replayed by the replay code inside the `ttd_replay_current_events` function, otherwise it gets blocked.

6.4 Logging daemon

A logging daemon is responsible for taking events from the ring channel and saving them to a persistent store. Its role is to keep the free space in a ring channel under a threshold, above which a flow-control notification – or even worse blocking – of the sender is required.

The logging daemon benefits from the simple, linear organization of the ring channel. The ring-channel content is flushed as a single write to filesystem. Only at this time does the event data incur its first data copy (into the filesystem buffer cache). Because the logging daemon runs on a separate CPU, this copy is not on the critical path, as long as the filesystem and storage layers are able to keep up with the flow of events coming from the ring channel.

6.5 Instruction counting logic

During a replay, XenTT forces the guest system to take its original execution path. Replay of asynchronous events requires a knowledge of the exact position of the event in the system's instruction stream. During replay, the system is forced to take its original execution path, by injecting a nondeterministic event at the exact same place in the instruction stream. For example, replay of an interrupt requires the following changes to the system's state: i) preempt execution of the system; ii) save its current execution context; iii) create an interrupt invocation stack; iv) redirect the flow of control to the interrupt handler.

A position of event in the system's instruction stream is determined by a number of instructions executed since the system's start. Lack of hardware support and high overheads of implementing instruction counting in software complicate realization of this simple

idea. Several decades of research led to the development of multiple instruction counting algorithms, which can be implemented in software [124], or in hardware [32] (see Chapter 3 for details and history of development of the branch counting techniques). In a special case of a system with purely synchronous signals, the notion of time can be reduced to a logical clock, since in this case, only the order of events between communicating processes is nondeterministic.

On a modern hardware, a practical instruction counting algorithm relies on support from hardware branch counting interface. The performance monitoring interface is a primary mechanism designed to assist performance analysis of the compiler and low-level system optimizations [51, 49]. Recent versions of Intel and AMD CPUs provide a capability to monitor a number of metrics reporting the state and performance of internal CPU components (see Appendix B of the IA-32 Intel Architecture Optimization Reference Manual [50] for a complete description of the performance monitoring interface of the Intel architecture).

The advantage of the hardware-assisted instruction counting algorithm is performance and transparency. Hardware counters introduce only minimal overhead to the execution of the system, but at the same time can count instructions through the entire stack of the guest virtual machine.

The main disadvantage of the hardware-based counting is its fragility. Hardware performance counters are only statistically correct, and provide no guarantee about accuracy of the counters on per-instruction basis, and can over- or under-count instructions in practice. This work uses a number of techniques to make sure that the counting is accurate.

6.5.1 Positioning events in the instruction stream

The hardware branch counting interface in Xen is primarily designed to provide integration with the OProfile statistical sampling profiler [114]. Xen provides a limited interface to read and write hardware branch counters, but does not implement virtualization of branch counters on a per-guest virtual machine basis, does not provide mechanisms for configuring and handling counter overflows, and does not provide interface for regestering overflow handlers – handlers for the nonmaskable interrupts generated by the hardware performance

counting interface.

XenTT extends Xen with the limited support for virtualizing branch couters. The main goal is to count a number of branches or instructions inside the time traveling virtual machines, and carefully filter branch counting events from outside of the logged guests.

To achieve this goal, XenTT implements an abstraction of a logical branch counter. XenTT extends the `vcpu` – Xen virtual CPU structure with a field, which represents the number of instructions executed by the virtual machine. Two steps are required to implement this logical counter in Xen. First, XenTT needs to count only instructions executed by the guest virtual machines, not by the hypervisor. Second, XenTT needs to accurately attribute instructions to individual virtual machines.

The Intel implementation of the hardware performance counting provides a way to specify whether the counter is active in `ring0`, or `ring1-ring3` contexts. XenTT configures the counter to count performance events only above the `ring0`, since all guest virtual machine code runs in the user rings.

To attribute performance events to individual virtual machines, XenTT implements a notion of a counter context switch. Every time the hypervisor context switches between guest virtual machines, XenTT saves the value of the physical branch counter for the switched out virtual machine, to make sure it resumes counting at exactly right point. For the guest which is context switched in, XenTT records the current value of the physical CPU counter to make sure it knows the position at which the guest resumes its execution.

With the above changes, the value of the logical counter L_{now} at any moment of execution is provided as the sum of the logical counter at the context switch in time $L_{t_{ctx\_in}}$ and increment of the corresponding real hardware counter from the context switch time $R_{t_{ctx\_in}}$, to the point at which the measurement is taken R_{now}:

$$L_{now} = L_{t_{ctx\_in}} + (R_{now} - R_{t_{ctx\_in}}) \tag{6.1}$$

6.5.2 Fixing nondeterminism of branch counters

On the Intel architecture, two hardware performance counters can be utilized to implement an accurate instruction counting algorithm: branch instruction retired counter (`BR_INST_RETIRED.ALL_BRANCHES`), and instruction retired counter (`INST_RETI-`

RED.ALL) [51]. The first counter – branch counter – counts all taken branches executed by the CPU, at the moment when a branch instruction becomes retired. The instruction counter works in a similar way but counts all instructions at the moment when they become retired.

The branch retired counter was among the first hardware performance counters added to various CPUs. It became a traditional mechanism for implementing the precise instruction counting algorithm [27]. The intuition behind this approach is an obvious observation that a value of the instruction pointer register alone is not sufficient to uniquely identify a point in the program's instruction stream alone. In the example of a simple loop, the same instruction pointer gets executed many times. However, if the instruction pointer is extended with the number of all taken backward branches, the tuple {instruction pointer, branch counter} is sufficient to identify the exact position in the instruction stream.

On the x86 systems, the above tuple must be extended with the value of the ECX register. The reason for that is that string copy instructions can be preempted with the interrupts in the middle of the copying loop. The last iteration is of the loop is remains in the ECX register.

Until recently, most deterministic replay systems used the retired branch counter for implementing the precise positioning of events. Recent versions of the Intel CPUs extend the performance monitoring interface with a counter capable of counting all instructions retired by the CPU. It may seem that the instruction counter is a perfect hardware mechanism, which immediately solves the problem of accurate instruction counting. In practice, this is not true. Two problems complicate implementation of an accurate branch counting: delay of the counter overflow interrupt, and nondeterminism.

6.5.2.1 Delay of the counter overflow interrupt

The hardware performance monitoring interface of the Intel CPUs provides support for preempting execution of the system when a certain number of performance events is reached. In other words, it is possible to configure a performance counter to signal a nonmaskable interrupt when the counter overflows.

XenTT uses the branch counter interrupts to preempt execution of the guest system during replay in a controlled manner. XenTT configures the branch counter to overflow

and triggers the interrupt at the point at which it needs to inject an asynchronous nondeterministic event in the system's instruction stream.

A nonmaskable counter overflow interrupt can be delayed for many cycles. Therefore, if the interrupt from the counter gets delayed, XenTT might miss a point in execution at which the event occurred during the original execution, and fail to force the system to repeat its original run.

A practical approach to address the interrupt delay problem is to configure the replay engine to preempt the execution of the system long enough in advance to account for the delay of the interrupt. After the interrupt is received, the system is single-stepped until the proper point in the instruction stream is reached.

In practice, most of the times the branch counter interrupt is delayed by just several instructions. However, on a long execution, it is possible to see the interrupt to be delayed for more than a hundred of instructions. XenTT therefore configures the replay engine to preempt execution of the system 120 instructions prior the point at which it needs to replay an asynchronous event.

6.5.2.2 Counter nondeterminism

Both branch and instruction counters become nondeterministic in face of interrupts and exceptions. XenTT faces this problem during replay, when the system is single-stepped for long periods of time in order to compensate for the effect of the delay of the counter overflow interrupt. Table 6.1 lists several examples of nondeterministic behavior of the instruction retired counter on the Intel 2.4 GHz 64-bit Quad Core Xeon E5530 Nehalem processor, which is used for development and testing of the XenTT replay engine.

To address nondeterministic behavior, this dissertation developed two mechanisms aimed to correct the instruction counting algorithm: instruction counting in software, and correcting the current value of the hardware counter.

Software counting is a way of fixing anomalous behavior of a branch counter by counting retired instructions, and branches in software. XenTT implements software counting only when it executes the system in a single-stepped fashion. This work extends Xen with an instruction decoding functionality. During the single-step execution, XenTT uses the decoding engine to parse the instruction stream, and correct for incorrect hardware counter

Table 6.1. Examples of an anomalous behavior for the instruction retired counter.

Instruction stream	Counter behavior
ret; jmp;	After ret counter increments by two, but does not increment after jmp
rep <foo>	Branch counter can be lost if the instruction is preempted with an interrupt

behavior. XenTT implements code for correcting behavior of the retired branch, and retired instruction counters.

The software counting is incapable of fixing the counting anomaly if nondeterministic behavior is triggered while the system is not in the single-step execution mode. An instruction sequence which might trigger nondeterministic behavior of the counter can be preempted with an exception either during the original run, or during replay. In both cases, it is impossible to observe and correct such anomaly.

To preserve determinism of instruction counting, XenTT tries to guess the correct value of the counter based on the value of instruction pointer register. If the counter is only two instructions apart from the recorded value of the counter, at which the event happened during the original run, and at the same time the values of the EIP registers between original and replay runs are the same, XenTT changes the value of the hardware counter to match the one recorded during the original run.

During development of XenTT, the implementation of accurate instruction counting was tested on several hardware architectures. Both instruction and branch counters, and several techniques to ensure reliable deterministic behavior of the counters were tried. Table 6.2 summarizes the results for different hardware architectures.

XenTT achieves determinism in the behavior of the hardware performance counters on two server-class processors: Xeon Nehalem, and Xeon Sandy Bridge. A surprising finding was that branch counters are nondeterministic on a laptop-class Core i7–2760QM processor, despite the fact it has the same family and model numbers as the server-class Xeon.

Table 6.2. Counter determinism on various processors.

Processor name (family, model)	Performance counter	Achieved behavior
Xeon Prescott (0xf, 0x4)	BR_INST_RETIRED	nondeterministic
Xeon Nehalem (0x6, 0x1a)	BR_INST_RETIRED	nondeterministic
	INST_RETIRED	deterministic
Xeon Sandy Bridge (0x6, 0x2a)	INST_RETIRED	deterministic
Core i7 Sandy Bridge (0x6, 0x2a)	INST_RETIRED	nondeterministic

6.6 Execution scheduling

When a recorded virtual machine execution is re-executed, XenTT relies on a custom execution-scheduling engine to dynamically inject the recorded stream of nondeterministic events back into the replaying guest. The *execution-scheduling engine* is the part of the replay engine that ensures deterministic execution of the guest during any replay run. The engine executes the guest in a controlled manner from one nondeterministic event to the next. XenTT distinguishes four types of events: asynchronous, in-place, optional, and nonreplayable.

A proper design of the event scheduling engine and a proper choice of the event scheduling types ensures extensibility of the XenTT replay with the possibility to record additional information about the execution without breaking the core of replay responsible for maintaining determinism. Specifically, XenTT extends the core deterministic engine with the ability to record information about time, which is required for providing a faithful performance model during the replay runs. This work further relies on extensibility of the logging code to add tracing of additional debugging events, which help to analyze violations of determinism in the long system runs.

The XenTT execution scheduler implements support for the asynchsonous and in-place events, which were discussed in Chapter 4. In addition to these events, XenTT provides support for another two event types: optional, and nonreplayable.

Optional events implement best-effort service: The XenTT replay engine will replay an optional event if the engine observes that the guest is at a position in the instruction stream at which the event occurred in the original run; otherwise, it will discard the event. XenTT relies on the optional event type to carry performance information about the guest's original

run. During replay, XenTT uses optional events to implement a control channel between the replay daemon and the execution scheduling engine. The replay daemon injects control events into the replay log to enable and disable BTS tracing of the guest, and the hypervisor, control verbosity levels for various components of the replay engine, and so on.

Nonreplayable events carry arbitrary information from the hypervisor, and guest's execution environment into the permanent log. XenTT uses nonreplayable events to implement debugging primitives: `printf` into the event log, tracing of real-time performance information during both the original and replay runs.

XenTT uses event types for (1) reordering events during replay and (2) choosing a minimal subset of events required for determinism of execution during replay. For example, in-place events only need to be recorded if they may affect later execution of the guest. In-place events that copy data out of the guest do not *need* to be recorded to ensure deterministic replay, but XenTT allows such events to be recorded for debugging or offline-analysis purposes.

6.6.1 Event scheduling

Four types of nondeterministic events define the logic of the XenTT's execution scheduling algorithm: *in-place*, *asynchronous*, *optional*, and *nonreplayable*. This set of event scheduling types provides a general mechanism for supporting the logic required to ensure correctness of replay, record and replay performance information, and implement a general debug and tracing logic.

Section 6.6 provides a detailed description of event scheduling types; this chapter describes the logic of the execution scheduling algorithm, which is required to implement replay of the four scheduling types.

6.6.1.1 In-place events

Following its original execution path, a system deterministically reissues all synchronous requests during replay. Replay tools interpose on these requests and return values the system received during the original run. Despite the fact that during replay, the system deterministically follows its original execution path, the external nondeterministic environment does almost nothing. During replay, all external events are read from the log.

The execution path of the external environment is limited to accessing requests in the log. Compared to the original run, this small execution path introduces less interference with the hardware state of the CPU, and the replayed system can run faster.

6.6.1.2 Asynchronous events

During replay, XenTT relies on a combination of hardware branch counters and the single-step execution to inject nondeterministic events into the execution of the system at the exact position at which they occurred during the original run. XenTT configures branch counters to overflow and raise a nonmaskable overflow interrupt a hundred of branches before the original event takes place. This is done to address a hardware delay in receiving the interrupt.

After execution of the system is preempted with the overflow interrupt, XenTT continues execution of the system in a single-step debugging mode. XenTT relies on the hardware support for preempting the system with the CPU debug exception after each instruction.

Reaching the target place in the execution, XenTT replays the asynchronous event and continues execution by scheduling execution of the system to the next nondeterministic event.

6.6.1.3 Optional and nonreplayable events

XenTT tries to replay optional events if the position of an optional event in the instruction stream of the system falls on one of the system's exits into the hypervisor. If, however, such exit does not occur, an optional event is dropped.

XenTT always drops a nonreplayable event. The main purpose of a nonreplayable event is to collect debugging information during the system run. Nonreplayable events make sure that such debugging information can be traced easily in a reliable and extensible way.

6.6.1.4 Scheduling execution of the guest during replay

Figure 6.4 illustrates the scheduling decision taken by the XenTT's execution scheduling code. At the moment t_{now}, the guest system exits into the hypervisor through one of the exception mechanisms. Similar to the interposition code described in Section 6.3, the

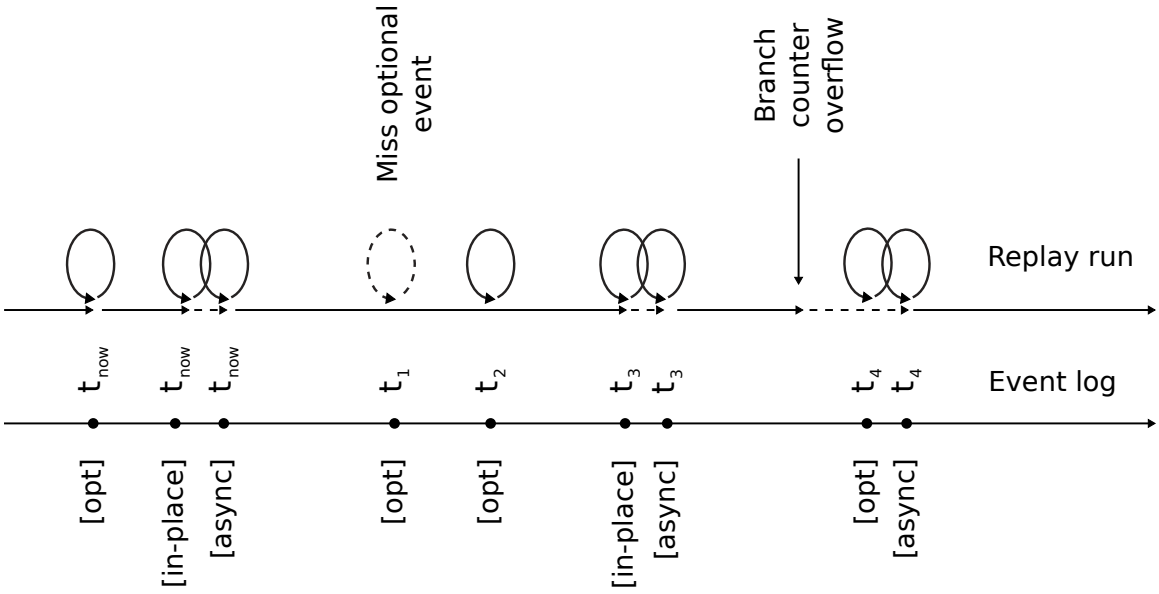

Figure 6.4. Example of the execution scheduling during replay.

interposition code for this exit invokes the `ttd_replay_current_events` function, which tries to replay all events which happen at this point of the guest's execution.

In the front of the event log, the replay code finds an optional event, which was recorded at time t_{now} during the original event. The replay function replays this optional event, and tries to schedule the execution of the guest.

Since the next event is in-place, XenTT lets the system run, basically allowing the hypervisor to continue execution of the exception handler until it encounters the interpositon code, which asks for replay of an in-place event. When the system asks for the replay of the in-place event, the replay function replays it. The replay function does not return until all events which can be replayed at this point of execution are replayed. Since the next event is an asynchronous event, which also occurs at time t_{now}, the replay function replays it right away.

To schedule the next execution interval of the guest system, the execution scheduling code searches through the log for the next "schedulable" event. Schedulable events must have a type in-place, or asynchronous. Therefore, the execution scheduler chooses the next event to be the in-place event at time t_3. Since this event is in-place, the scheduler lets the system run until the system asks for replay of this event. During this run, some optional

events can be missed, and later dropped (the event at time t_1), but some will be replayed (event at time t_2).

At time t_3, the replay code replays both in-place and asynchronous events. Searching for the next schedulable event, the execution scheduler picks an asynchronous event at time t_4. To replay an asynchronous event, the event scheduler needs to configure the branch counter to overflow and single-step execution of the system until the proper point in the instruction stream is reached. Both optional and asynchronous events are replayed together at time t_4.

6.6.1.5 Marking optional events as in-place

The replay engine benefits from recording as many in-place events as possible. First, in-place events do not require a slow single-stepping – the system can just run to a point at which it exits into the hypervisor. Second, in-place events help to correct nondeterministic behavior of the hardware branch counters (see Section 6.5.2 for more details).

XenTT tries to mark as many events as possible to be in-place. If the guest system exits into the hypervisor via a deterministic exit, e.g., a hypercall vector, privileged instruction, or one of the deterministic exceptions, XenTT marks the normally optional `TTD_UPDATE_BRCTR`, which is used to record performance information on every exit from the guest system into the hypervisor (see Section 8.7.4 for additional details), as in-place.

6.6.1.6 Single-stepping and EFLAGS

XenTT relies on the trace flag (TF) in the EFLAGS register to force single-stepped execution of the guest system. Obviously, the TF flag can change execution of the system during replay, if it gets "leaked" into the guest. For example, the TF flag changes execution of the Linux kernel on the system call entry path. XenTT's replay logic prevents the TF flag set by the replay algorithm from leaking into the system.

Several x86 instructions can let the guest system learn that the TF flag is set. `pushf`, and `int` instructions save the entire EFLAGS register on the stack of the guest system. To clear the TF flag from the saved EFLAG register, XenTT implements an x86 instruction parser, which is used during the single-stepped execution to find `pushf`, and `int` instructions, and clear the TF flag on the stack.

6.6.1.7 Replay on exit to guest

It is reasonable to assume that the timestamp of the guest system changes only while it is running. In practice, this assumption is not true. A logic of the instruction emulation implemented inside the Xen hypervisor can change the instruction pointer of the guest system, moving its timestamp forward. XenTT supports such cases by checking if more events are ready to be replayed right before exiting into the guest system.

CHAPTER 7

DETERMINISM IN XEN

A Xen virtual machine is exposed to two major sources of nondeterminism: a hypervisor interface, and virtual device drivers. This chapter explains the details of implementing an interposition layer for Xen. First, this chapter discusses the implementation of the XenTT interposition layer from the very first steps of the guest virtual machine life cycle: creation of the guest's memory image, execution and emulation of the first privileged instructions, exceptions, hypercalls, and interrupts. Then, the chapter proceeds with describing an interposition of the four device drivers, which are required for a typical guest system to run: console, xenstore, diks, and network.

To understand the design and implementation of a deterministic replay engine for the Xen virtual machine monitor, it is important to understand what constitutes the state of the guest system:

- **Memory:** The majority of the guest's state is contained in the memory of the guest virtual machine. The guest system interfaces with the hypervisor through several shared pages (see Chapter 5 and Chapter 7 for additional details), which model the hardware configuration registers of a physical CPU, and contain information about pending and masked interrupts, wall clock time, and a list of the physical to machine translation. The majority of nondeterministic updates from the Xen hypervisor are memory updates to these shared memory pages. Additionally, Xen hypervisor uses a copy-user function for copying return values from the hypercalls in the preallocated locations inside the guest memory. The low-level view of the guest's state allows XenTT to model the state of the system as a set of memory pages, and model the nondeterministic events as updates to these pages.

- **Virtualized CPU and memory management unit:** A part of the guest's state is represented by a virtual CPU inside the hypervisor. Specifically, this part represents the state of the virtualized getneral CPU registers, and state of the virtualized memory management unit. The hypervisor updates the register state of the system when it redirects control flow of the guest system to an interrupt handler, required to process a pending event channel notification. The memory management unit is updated through a set of hypercalls, either by the guest system itself, or by the device driver domains which rely on the page remapping techniques to transfer data from the backend devices to the guest system, and *vice versa*.

- **Backend devices:** Backend devices are part of the run-time execution environment of the guest system. During both the original and replay runs, backend devices follow the bootstrap protocol initiated by the domain creation mechanism, and the guest system. XenTT makes sure that both bootstrap and communication protocols are deterministic, and will recreate all memory exchange operations during the replay runs.

- **Hypervisor:** The hypervisor itself does not contain any state directly observable by the guest system besides the virtual CPU, and virtual memory management unit. The domain, and virtual CPU information structures contain flags which can affect the execution of the guest system, e.g., terminate it immediately. In practice, such events are not triggered by the Xen VMM. However, such nondeterminism can be introduced by the privileged users. XenTT does not try to block or interpose such nondeterminism.

7.1 Domain creation and initial state

The initial state of the domain consists of the domain configuration file, command line arguments to the domain creation commands, and state of the guest's disk devices. Device state of the domain and the XenStore information is always recreated from scratch by the domain creation protocol.

Domain creation protocol, which is responsible for configuring backend device driver state for the guest domain, is deterministic, as analyzed below. To ensure determinism of

the disk state, both original and replays runs of the system start off a fresh snapshot of the guest's disk. XenTT uses LVM – a block versioning store, and `dd` – a block copy command, to create snapshots of the guest's disk devices.

7.1.1 Domain creation protocol

Most of the work required for constructing a new guest virtual machine is performed by a set of user-level tools running inside Domain 0. The Xen hypervisor is only responsible for allocating required data structures at the hypervisor level. Domain creation protocol consists of multiple steps. The goal of this section is to analyze this protocol for deterministic behavior:

- **Trigger creation:** A user-level application `xm` provides a console interface to start a new virtual machine. `xm` uses functionality of the Xend daemon. `xm` passes a request to the Xend along with the path to the virtual machine configuration file, and optional command line parameters.

- **Parse domain configuration file:** Xend – a user-level daemon running inside Domain 0 is the central component which orchestrates the creation of a new virtual machine. Xend parses the virtual machine configuration file, which describes the CPU, memory, and device requirements for the new virtual machine. Xend parses the configuration file and passes control to the Xen hypervisor, triggering initialization of the data structures for the new domain.

- **Create domain in Xen:** Xen uses the `domain_create` function to allocate the domain structures inside the hypervisor, and initialize all the subsystems, which implement the domain: memory, shared info page, grant tables, event channels, architectural part, and scheduling.

- **Init domain's memory image:** After Xen returns from the hypercall, the memory subsystem for the new virtual machine is ready, and therefore, Xend can initialize it with the kernel of the new guest. Xend uses domain builders – a set helping applications which abstract implementation of the low-level architectural details for

every guest system, and implement a general interface for initializing the memory of a guest system with the guest kernel, its initrd file system, modules, etc.

Note that the Linux domain builder allocates two pages from the guest's memory image for the console and xenstore shared ring buffers.

- **Init devices:** Xend starts connecting device drivers for the new guest virtual machine by introducing it to the Xenstore. Xenstore is a special device in the Xen architecture; it is connected separately from the rest of the devices. `xenstored` – Xenstore daemon process – receives the connection parameters for the new virtual machine: event channel port number, and the machine frame number for the shared ring buffer.

  After Xenstore is connected, Xend triggers creation and connection of all all devices configured for the guest virtual machine. Xend uses a clever concept of device controllers which hide complexity of individual devices behind a simple hierarchy of interfaces.

  Device controllers which are created for every device generate two entries in the Xenstore database. When written to the database, these entries trigger creation of the corresponding devices inside the backend and frontend virtual machines.

  To implement this elegant idea, Xen architecture supports a concept of a device bus. Each guest kernel has a Xenstore device driver, which, like any other bus device driver, is responsible for discovering new devices on the bus. Device discovery is triggered when a new entry is written to the corresponding part of the Xenstore database. If a new device is added to the Xenstore device bus, the Xenstore driver tries to locate a corresponding device driver in the guest kernel. After that, Xenstore driver starts a traditional device driver initialization (probing) protocol.

- **Wait for devices and unpause:**

  After device information is written to Xenstore, Xend exits back to `xm`. At this point, domain is created; however, its devices are not connected. `xm` initiates a "wait-for-device" protocol, which makes sure that all backend devices are created before

domain starts. When backend devices are ready, `xm` can proceed and unpause domain starting its execution. At this point, the guest kernel starts booting.

At every stage of the domain creation protocol, the components involved in creation of the guest system depend on a set of explicitly passed configuration parameters. Domain creation tools did not show any nondeterminism. It is possible that under a high load, the Xenstore database will fail the transactions from the backend domains, but in practice, this is an unlikely scenario which is so rare that XenTT does not handle it.

7.2 Xen virtual machine interface

A guest virtual machine receives a large fraction of nondeterministic events through a paravirtualized interface of the Xen hypervisor. This section enumerates the sources of nondeterminism in this interface and describes how they are logged by XenTT.

7.2.1 Start info

Start info page is a page shared between the guest system and the hypervisor at boot time. Start info is used to pass the boot configuration information, like the version of the hypervisor, number of physical pages allocated to guest, start info domain flags, machine address of the shared info page, machine frame numbers, and event channels for connecting the XenStore and console backend drivers, guest's kernel command line arguments. XenTT leverages determinism of the domain creation protocol, which recreates values in the start info page during reboot.

7.2.2 Shared info

Shared info page is an integral part of the domain's initial state. Shared info gets updated by the hypervisor even before the guest virtual machine starts running. Shared info page gets updated by the hypervisor for a number of reasons: event delivery masking and unmasking, time version updates, updates to the shared info arch fields, updates to the wall clock time, etc.

XenTT implements a general interface for logging and replaying updates to the shared info page. Every update to the shared info page is encoded as a triple:

```
{offset, update data, update size}
```

The triple encodes the offset of the update from the beginning of the shared info page, size of the update, and new data to be written in the shared info page.

7.2.3 Grant table operations

The grant table interface implements a mechanism for sharing and transferring pages between domains. A domain advertises its desire to make a page accessible by another domain by setting a proper grant permission in its grant table. The grant tables store information for memory access permissions and in-flight sharing of pages between domains. Grant tables are typically updated asynchronously by the backend drivers. A grant table update has a form of a compare and exchange, or a clear flag operation.

For the compare and exchange operation, XenTT records a grant table operation as an index pointer into the grant table array, old value, new value, and the result of the compare and exchange operation. For the flag clear operation, XenTT records an index pointer into the grant table array, and the bit number, which has to be cleared.

7.2.4 Event channels

An event channel is a one-bit communication primitive used to send immediate notifications between virtual machines. The logic of the event delivery is implemented as several checks and updates of the `shared_info` page inside the Xen `evtchn_set_pending` function. The hypervisor first checks that a particular event was not already signaled to the domain. It then checks that the event is not masked by the guest. If the event can be delivered, the hypervisor updates the `shared_info` page, and makes sure that a guest's virtual CPU is preempted to receive this event immediately.

To deliver the event, the Xen hypervisor preempts execution of the guest system. Since the event channel pending bit is set in the shared page, the low-level code responsible for exiting to guest virtual machines creates a special interrupt stack to force execution of the interrupt handler. Although the event delivery protocol requires several updates to the `shared_info` page, and injection of an interrupt frame into the guest, its execution is deterministic. XenTT records and replays event notifications by simply invoking the event delivery function (`evtchn_set_pending`). To interpose on event channel notifications, XenTT implements a source code wrapper around the `evtchn_set_pending` function.

7.2.5 Copy user interface

The copy-user interface is used to return data from the hypervisor to the guest. XenTT wraps the copy-user function and records all asynchronous copy-user events. Recording of in-place copy-user events is optional, since they will be reinvoked as part of another action.

7.2.6 Privileged instructions

The Xen hypervisor supports privileged CPU instruction emulation (e.g., `in`, `rdtsc`). XenTT interposes on this emulation to detect instructions that return nondeterministic results.

7.2.7 In-place exceptions and interrupts

Most exits of the guest system into the hypervisor via one of the exception vectors are in-place events: exception vectors 0-16, hypercall entry vector (int 0x81). These exceptions are deterministically re-executed by the guest system, if determinism of all other events is preserved.

Some exceptions are artefacts of the Xen virtualization layer, they are unobservable by the guest system, and are therefore nondeterministic. Examples of such nondeterministic exceptions are: (1) page faults – some page faults are triggered as a result of the shadow paging translation layer, and not as a result of the guest operation, such page faults are transparent to the guest; (2) coprocessor not available (exception 7); (3) general protection violation (exception 13).

XenTT does not record the deterministic in-place exceptions to reduce interposition overhead. XenTT, however, provides a configuration option to record all exits from the guest system for the purpose of detecting violations of determinism as early as possible.

7.2.8 Interrupt 0x80 – guest system call

Interrupt 0x80 is an interrupt which is traditionally used to implement the system call interface in the Linux kernel. Xen hypervisor allows direct execution of the interrupt, i.e., the guest user-level can transition straight from the Ring 3 to Ring 1, bypassing the hypervisor. XenTT does not record information about the 0x80 interrupt.

7.2.9 Hypercalls

Hypercalls implement an interface similar to a system call interface, but between a virtual machine and the hypervisor. Hypercall invocations are deterministic in a way that if determinism of execution is preserved, the guest always re-issues the same hypercalls during replay. Therefore, hypercalls themselves do not need to be logged.

7.2.10 Hypercall continuations

Hypercall continuations are a mechanism which allows Xen to suspend execution of a long hypercall, create a hypercall continuation, and resume the hypercall later. Many of the Xen hypercalls accept a potentially rather long list of arguments, which are processed in a loop. From inside the argument processing loop, the hypercall code cooperatively checks if local events need delivery, or if a soft IRQ is pending, and preempts its invocation exiting immediately. The continuation mechanism guarantees that the Xen microkernel will re-invoke the same hypercall with the value of saved continuation after processing a pending event but before returning to guest.

Hypercall continuations are not deterministic between the original and replay runs, which can break determinism of execution. To simplify XenTT architecture, hypercall continuations are disabled for the time-traveling guests.

7.2.11 EFLAGS register

To single-step the guest during replay, XenTT uses the trace flag (TF) in the EFLAGS register of the guest. To preserve determinism of the guest, XenTT virtualizes the EFLAGS register. During replay, when the guest is single-stepped, XenTT parses the guest's instruction stream and detects instructions that try to save the EFLAGS register.

7.2.12 Memory virtualization

A virtual machine monitor provides the illusion of continuous physical memory to the guest. To support execution of unmodified guests, full virtualization has to implement this abstraction transparently to the guest [177]. Trying to optimize performance of the guest system, paravirtualization exports the layout of an actual machine memory to the guest. Particularly, a paravirtualized hypervisor exports a list of available machine pages,

which a guest can treat as its physical memory. The paravirtualized guest kernel creates an abstraction of a continuous physical memory for the upper layers of the system.

A set of physical pages available to the guest system can vary between the original and replay runs. If exposed to the guest system, a difference in the memory layout violates determinism of execution during replay. Fortunately, similar to full virtualization, Xen supports a special mode of memory management, which provides full virtualization of the physical memory: *shadow paging*.

Shadow paging allows the guest system to use its physical memory as if it runs on real hardware. Translation between the virtualized physical memory and the real physical memory of the machine is performed by the hypervisor and is completely transparent to the guest. For each guest system, the hypervisor creates a shadow page table, which is used by the hardware CPU instead of a guest's page table. Xen intercepts all updates to the guest's page table and propagates them into the shadow page table performing a required translation from the physical to machine memory. See [39] for a general description of shadow paging, [22] for the state of the latest Xen implementation, and Section 2.3 for the future of virtual memory virtualization in Xen.

To ensure determinism of memory management, XenTT runs the guest system in a shadow paging mode.

7.2.13 Time virtualization

To keep track of time, a nonvirtualized operating system relies on counting periodic timer interrupts. Due to the concurrent execution of multiple virtual machines and contention for the CPU, this approach is impossible in a virtualized environment. First, if the guest system is not running at the moment of interrupt delivery, the interrupt can be delayed or lost. Second, a guest system may not run continuously between interrupts. Finally, the correct operation of the guest kernel and user-level applications may rely on the presence of a high-precision time source.

To address these issues, Xen exports a wall-clock time, system time since boot, and run-time state statistics to the guest system through a shared memory region that Xen updates periodically. Additionally, the guest can account time by requesting periodic timer interrupts and by accessing the hardware time-stamp counter register (TSC). To obtain

the most recent time values, the guest system interpolates time values with nanosecond precision by reading a hardware timestamp register, which is used to compute the time passed since the last memory update.

To ensure determinism of time values, XenTT logs updates of these values in the shared info page, and implements emulation of the `rdtsc` instruction, which accesses the timestamp counter. XenTT logs updates to the guests run-time state statistics. These statistics reflect the amount of time a guest system spends in one of four states: running, runnable but not scheduled due to CPU contention, blocked by some other system activity, or idle. Finally, XenTT logs execution of periodic, polling, and single-shot timers.

7.3 Device drivers

The major source of nondeterminism in any system is communication with external devices. A guest virtual machine is not allowed to communicate with physical hardware directly.[1] Instead, all device communication is redirected to the virtual machine monitor, which multiplexes real hardware between multiple guest systems.

A guest system accesses its virtual devices via a backend-frontend split device pair (see Section 5.2 and [39, 99] for the detailed description of the Xen split device driver mechanism). To notify each other about new I/O requests, backend and frontend device drivers rely on a fast, lock-free, producer-consumer ring, which they share in a memory page. The backend and frontend devices add new requests to the ring by simply advancing the pointers in the shared ring, and sometimes notifying the other end via an event channel.

For the high-bandwidth I/O devices, the shared ring contains only pointers to the memory pages with the actual I/O payload. The ring essentially holds a queue of I/O requests. Each I/O request contains a machine frame number of a page with the actual I/O payload. A typical I/O transfer relies on the memory mapping mechanism, although other ways of communicating I/O data are possible, e.g., memory copy in and out of a large shared I/O buffer, hypervisor-supported memory copy operation, page flipping, etc. For example, low bandwidth devices like console and Xenstore communicate the data payload straight

[1] Xen provides direct device access capability for PCI devices. XenTT disables this feature to interpose on all nondeterministic device events.

through the shared ring page. The author's earlier work Fido [30] allows split devices to share the entire address space of their domains, to make ensure a complete zero-copy data transfer between domains.

Xen uses the following I/O transfer protocol between the frontend and backend domains. The frontend device driver grants the backend access rights to all pages containing the actual I/O payload. The frontend domain creates a batch of I/O requests in the shared ring buffer, and notifies the backend about available requests by updating the ring producer pointer. The backend reads the I/O requests from the shared ring, extracts the machine frame number of the page with the I/O payload, and asks the hypervisor to do the mapping or a memory copy of the page into its own address space. The hypervisor verifies that the frontend actually granted the access rights to the backend domain, and then performs the memory mapping operation.

7.3.1 Logging and replaying device drivers

There are two challenges in logging device driver communication in Xen. First, the overhead of logging every update to the memory shared between virtual machines is prohibitive. Instead, XenTT leverages the semantics of the shared ring, and logs only one event – pointer update. Thus, it is possible in theory that a guest virtual machine accesses the data in the shared memory before the pointer update, but in practice, device drivers do not do this.

The second challenge is a result of the fact that a shared ring is updated by a device driver inside a device driver domain kernel. XenTT has to record the exact state of the guest system at the moment of update. This state is only available in the Xen microkernel. To avoid expensive multistage communication needed to read guest's state, the author of this work suggests a novel technique, which ensures atomicity of a memory update and accessing the guest's state. Instead of updating a pointer in the ring, the device driver domain sends a special "active" message to the Xen microkernel. The message describes the update. The microkernel performs the update and atomically records the state of the guest system.

7.3.1.1 Shared ring interposition

To preserve determinism of replay, XenTT must ensure determinism of updates to the shared ring buffers. To control all shared rings updates from the backend devices, XenTT implements a special device driver interposition component – `Devd`, which is inserted between each pair of the backend and frontend device drivers to mediate their communication (Figure 7.1).

`Devd` is implemented as a user-level application, which runs inside the device driver domain, i.e., Domain 0 in a typical Xen setup. `Devd` follows the design patterns of a typical device driver framework. `Devd` implements a concept of the device bus. By monitoring the Xenstore database, `Devd`'s bus driver discovers new device drivers in the time-traveling guests. For each newly discovered device, `Devd` walks through the list of registered drivers, and tries to find a match for the device.

Instead of connecting to the event channels, and shared rings of the frontend domain, backend devices connect to the shared rings created by `Devd`. `Devd` remains transparent to the communication protocol between backend and frontend devices. Both backend and frontend devices update their ring pointers as they do in case of noninterposed execution. Running transparently to backend and frontend devices, `Devd` reflects all updates between the two rings it mediates. In Figure 7.1, `Devd` mediates the update of the `req_prod` pointer.

`Devd` relies on a minor change in the connection protocol of the backend drivers to detect the fact that domain is recorded or replayed, and read the connection parameters from the `Devd` subtree in the Xenstore database instead of the ones used for the unmodified Xen domains.

7.3.1.2 Determinism of ring operations

During the original run, `Devd` records a nondeterministic event for each update of the ring shared with the guest. `Devd` relies a set of marshaling functions similar to the ones used by the logging code inside the Xen hypervisor interposition layer. To ensure atomicity of the event recording with respect to the guest state, `Devd` relies on a technique of active messages. Instead of suspending the guest system, reading its time stamp, performing a ring

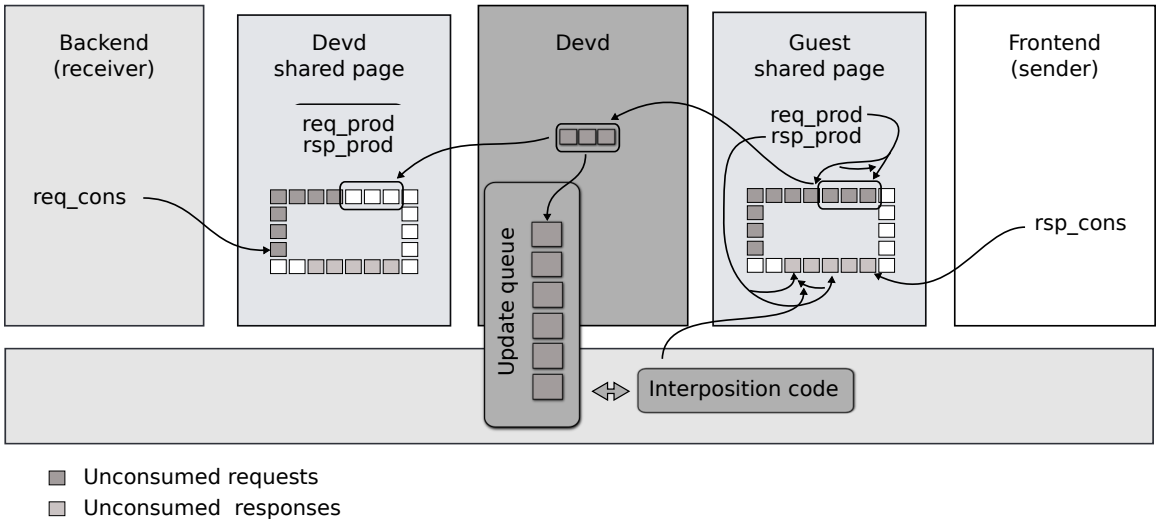

Figure 7.1. Details of ring interposition.

update, and resuming the guest, `Devd` sends a special update message to the hypervisor, which encodes the update in form of:

```
{offset in the ring page, update data, update size}
```

The XenTT logging layer receives the update request, and performs an update in a way similar to recording an asynchronous event, e.g., it sends an IPI interrupt to preempt execution of the guest, perform the update, and record the guest's time stamp atomically.

During replay, `Devd` controls ring pointer updates and synchronizes them with the XenTT execution scheduler. At the point of execution, at which a ring pointer update needs to be replayed, the replay code suspends execution of the system, and sends a request to `Devd` asking it to replay the pointer update. `Devd` waits while the backend driver processes all requests up to the one requested by the replay code. Then `Devd` updates the ring pointers and notifies the replay code that update is done. The execution scheduler resumes the execution of the guest then.

7.4 Determinizing proxies for Xen devices

`Devd` implements a determinizing proxies for the four most critical Xen devices: console, Xenstore, disk, and network.

7.4.1 Console

Figure 7.2 shows the architecture and the Devd interposition of the console device. In the Xen architecture, the frontend console device driver runs as a kernel module inside the guest's kernel. The guest's console device is similar to a serial device in an unmodified server system. The console device implements a channel to send and receive text messages to and from the guest system.

The console frontend establishes a shared page, which is split into two shared rings. Each ring serves as a producer-consumer ring for incoming and outcoming text. Since the console's bandwidth is typically low, the frontend device sends console's text data straight through the shared rings.

Inside the Domain 0, a user-level daemon, Consoled, receives the data from the shared rings, and exports them as a tty interface, which can be used from the Domain 0 domain.

Devd consumes two rings exported by the frontend domain, and exports a fresh pair of rings, which it creates in a memory mapped file. Devd advertises this fresh pair in the Xenstore so the Consoled daemon can connect to those rings. The Consoled daemon is modified to read the connection information from the Xenstore, and use the memory mapped files instead of the Xen memory sharing primitives. Devd mirrors all communication between the rings while ensuring the determinism of ring updates.

To recreate all console input deterministically during replay, Devd logs it into a file. During replay, Devd replays the inputs from the file, and blocks all accidental input coming from the Consoled. During replay, Devd passes all the console output to the Consoled. This way, a XenTT operator can watch the console of the system during replay.

7.4.2 Xenstore

Similar to console, the Xenstore frontend device (Xenbus) uses a shared page to connect to the user-level Xenstore daemon running inside Domain 0 (Figure 7.3). Xenbus uses two rings allocated straight into the shared page to send and receive Xenstore commands. Xenbus uses plain text format for transferring Xenstore data.

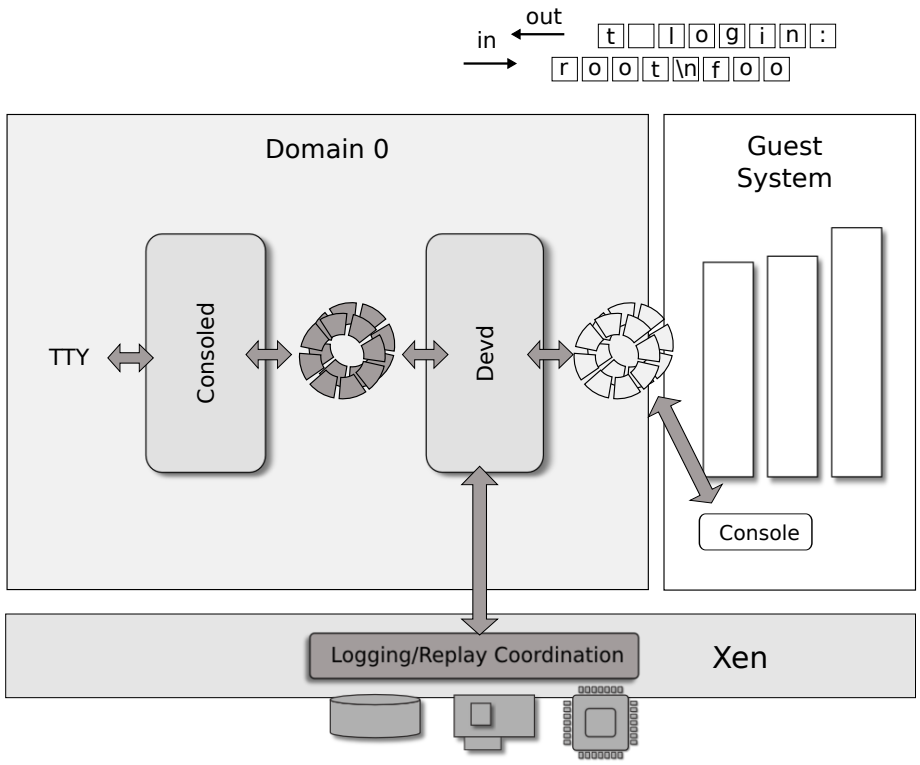

Figure 7.2. Architecture and interposition of the console device.

`Devd` consumes the rings exported by the frontend `Xenbus` driver, and advertises a fresh pair of rings, which it allocates in a memory mapped file, to the backend. During the original run, `Devd` logs all Xenstore input in a file. During replay, this input is replayed to the guest system.

7.4.2.1 Determinism of Xenstore transactions

Xenstore is a registry database fostering the exchange of device and domain configuration information; it implements a publish-subscribe notification interface across virtual machines. To support atomic updates, Xenstore implements a transactional interface for updating its state. Determinism of the device bootstrap protocol required deterministic Xenstore transactions. XenTT implements this support by making sure that transactions from a replaying virtual machine always commit.

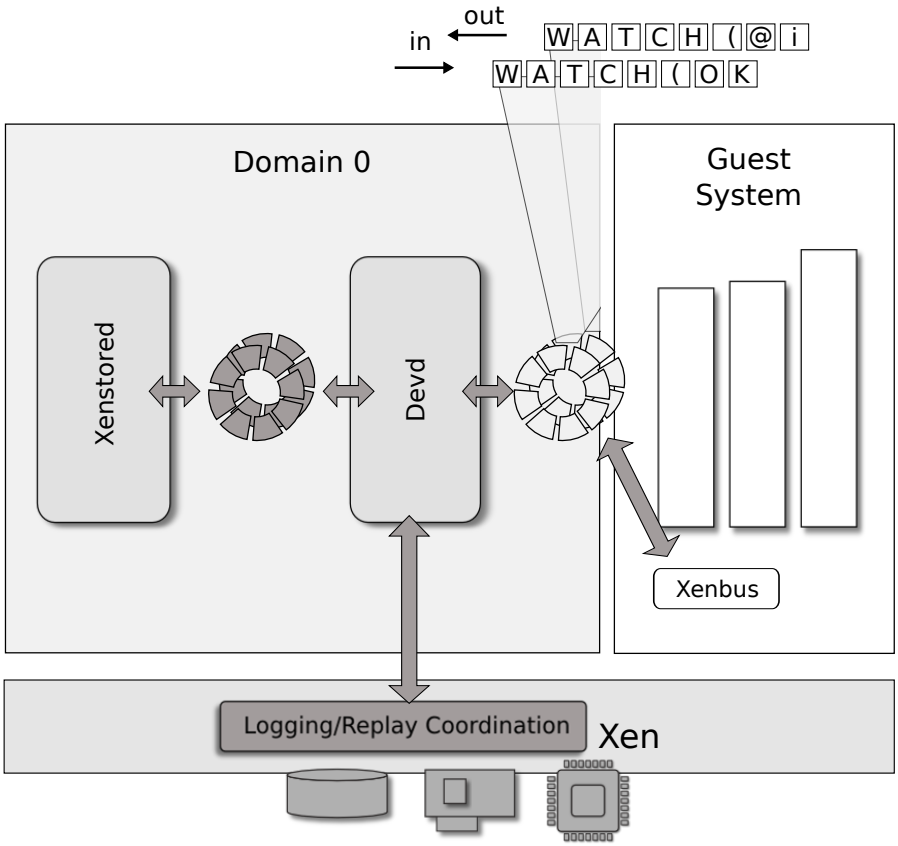

Figure 7.3. Architecture and interposition of the XenStore device.

7.4.3 Disk

Xen block device is implemented with two kernel modules – backend and frontend device drivers running in the Domain 0, and the guest kernels (Figure 7.4). The block frontend uses a shared page to implement a producer-consumer ring for the I/O requests. Each I/O request is described by a request data structure in the shared ring buffer. The request data structure stores a set of references to memory pages with actual I/O payload. Before putting request into the ring, the frontend device grants the backend domain the access rights to map the pages with payload for reading or writing. To perform the actual data transfer, the backend domain temporarily maps the payload pages into its address space.

An interesting detail of the block protocol is that the block split devices use only one ring for both outgoing requests from the frontend, and the incoming responses from the

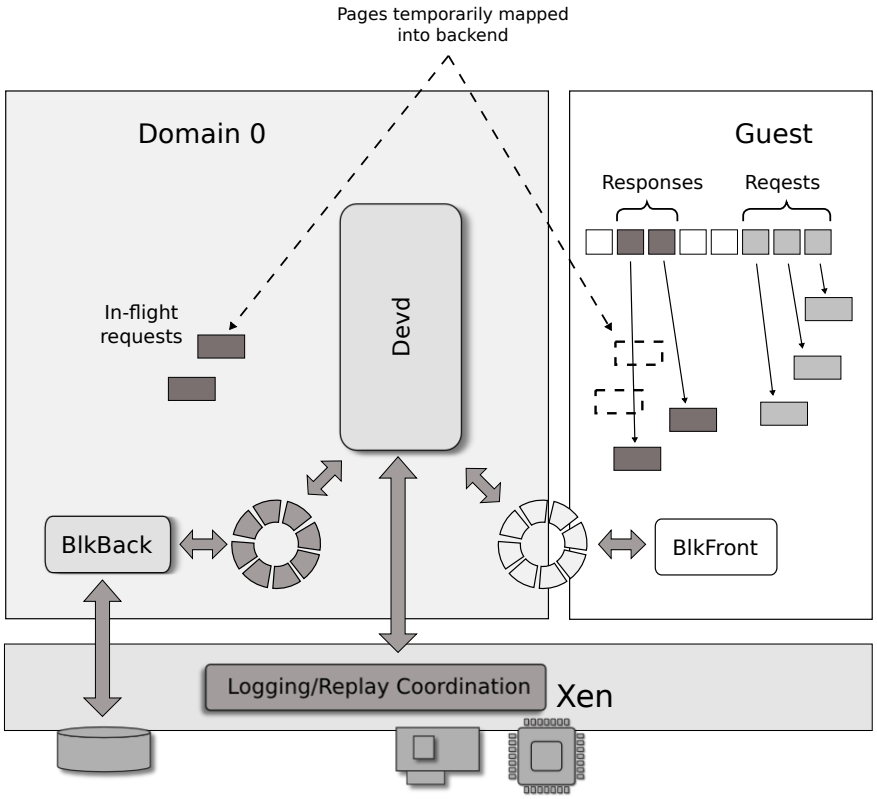

Figure 7.4. Architecture and interposition of the block device.

backend. In practice, this design choice implies that the number of outstanding in-flight requests is limited by the size of the ring, which in the current Xen architecture is limited to one memory page, or 32 I/O requests. Each request allocates one entry in the shared ring; this entry gets released only when the backend device receives a response from the I/O subsystem of the backend kernel.

Similar to the console, and Xenstore devices, `Devd` advertises a fresh ring to the backend device. Since the block backend device runs in the context of the Domain 0 Linux kernel, `Devd` needs a memory sharing mechanism to create a shared page between the user-level `Devd` daemon and the backend kernel module. XenTT implements a Linux kernel module –`ttd-relay`, which is used by the block backend driver to create a shared page. `ttd-relay` is a simple but general mechanism designed to establish memory mapping between the Linux kernel and user-level code. `ttd-relay` establishes the memory sharing by overloading a memory map (`mmap`) operation for a file in the Linux

debug file system. When a user-level code tries to open and map this file, `ttd-relay` allocates a memory page, which is later advertised to the block backend driver.

Similar to the Console and Xenstore devices, the block interposition code in `Devd` monitors and records all updates to the guest's shared ring. The I/O payload data are exchanged via the Xen memory mapping primitives. Memory mapping operations between the guest and Domain 0 are recorded and replayed as normal nondeterministic events in the Xen virtual machine interface by the XenTT interposition code.

7.4.3.1 Handling out of order responses

Although the payload of the disk responses is always deterministic, during replay, the backend device driver can return responses in the order different from the original run. `Devd` logs the order of responses, which the guest system observes during the original run. During replay, `Devd` caches and re-orders responses from the backend to make sure that the guest system sees them in the same order. `Devd` logs the order of the request in a file, and uses the request identifier as a way to uniquely identify requests.

7.4.4 Network

Similar to the disk device, the Xen network device is implemented with two kernel modules – backend and frontend device drivers running in the Domain 0, and the guest kernels (Figure 7.5). Split network devices use a pair of shared rings to implement two independent communication paths – transmit (TX) and receive (RX).

To send an outgoing packet, the frontend device puts a reference to the memory page with the send payload into the TX ring, and grants a backend domain rights to map this page. The backend driver maps the page with the data and passes it to the upper levels of the Domain 0 network stack. In a typical Xen setup, backend relies on the Linux software bridge or the NAT routing code to redirect packets from the guest domain to a driver of an actual physical network device. Only when the packet is sent out to the physical network, the backend receives a packet destroy upcall from which it can unmap the payload page, and return it to the guest domain.

The receive operation is also initiated by the frontend device. Frontend allocates a set of memory pages to receive network data, and sends them to the backend device via

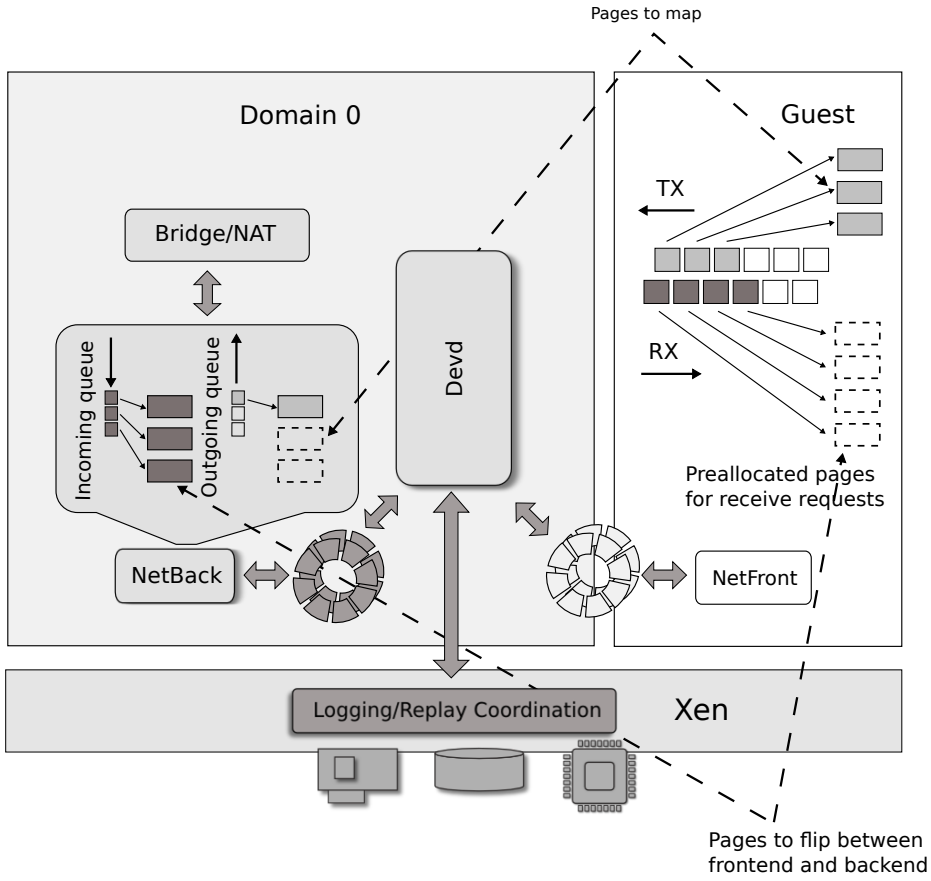

Figure 7.5. Architecture and interposition of the network device.

the RX ring. Since the physical network device uses the direct memory access (DMA) to receive packet payload in the memory of the Domain 0, the packets received for the guest system are already in the Domain 0 memory. To avoid copying packed data into the guest's memory, the backend device uses a special memory operation – page flipping, to exchange its own page with the network data for one of the pages allocated for the packet receive by the frontend domain. It should be noted that the Xen network architecture supports memory copying for the network packets of the small size. For a small packet, the copy operation is more efficient than a series of page table manipulations required for the page flip.

`Devd`'s interposition code monitors and logs all updates to the shared rings exported by the guest. Similar to the block device, during replay, the network backend can return responses for the send packets in the order different from the original run. `Devd` logs and replays the correct order of responses.

7.4.5 Low-overhead payload logging

To ensure determinism of the guest system during replay, XenTT needs to record and replay all network traffic received by the guest system during the original run. In practice, on a high-bandwidth network connection, XenTT must support recording of a 100MB/s stream of network data while introducing only minimal interference to the performance of the network.

XenTT uses a special logging architecture aimed to minimize recording overheads for the high-bandwidth network connections (Figure 7.6). First, XenTT implements a Domain 0 kernel module, which can be used by the backend drivers to log data payload for high-bandwidth I/O streams straight into a file without leaving the Domain 0 kernel.

Second, XenTT makes sure that I/O and CPU bottlenecks are removed from the critical path. XenTT allocates Domain 0 with two physical CPUs and two disks used for logging. On a high-throughput network workload, the network stream of 104MB/s can nearly saturate a sustained bandwidth of a modern hard drive; e.g., the hard disks used in the experimental setup have sustained throughput of 130MB/s. The stream of nondeterministic events recorded at the Xen virtual CPU boundary can also reach 30-60MB/s. To ensure noninterference of these two log streams, XenTT separates them into two physical disks.

A single physical CPU becomes another bottleneck. If two data streams are processed together, the overheads of memory allocation and data copying on the Linux file write path can make these streams to interfere. XenTT makes sure that Domain 0 runs with two physical CPUs.

7.5 Tools for scaling development

7.5.1 Page guarding

XenTT makes the following assumption about the way in which the Xen hypervisor updates the memory state of the guest system. XenTT assumes that the hypervisor will use only two interfaces to update the memory state of the guest system: i) pages which are explicitly shared between the guest and the hypervisor through the `share_xen_page_with_guest` function call; ii) memory copy updates through the `copy_to_user` function.

The Xen hypervisor shares several pages with the guest system, which serve as a low-overhead communication interface between the guest and the hypervisor. `shared_info`

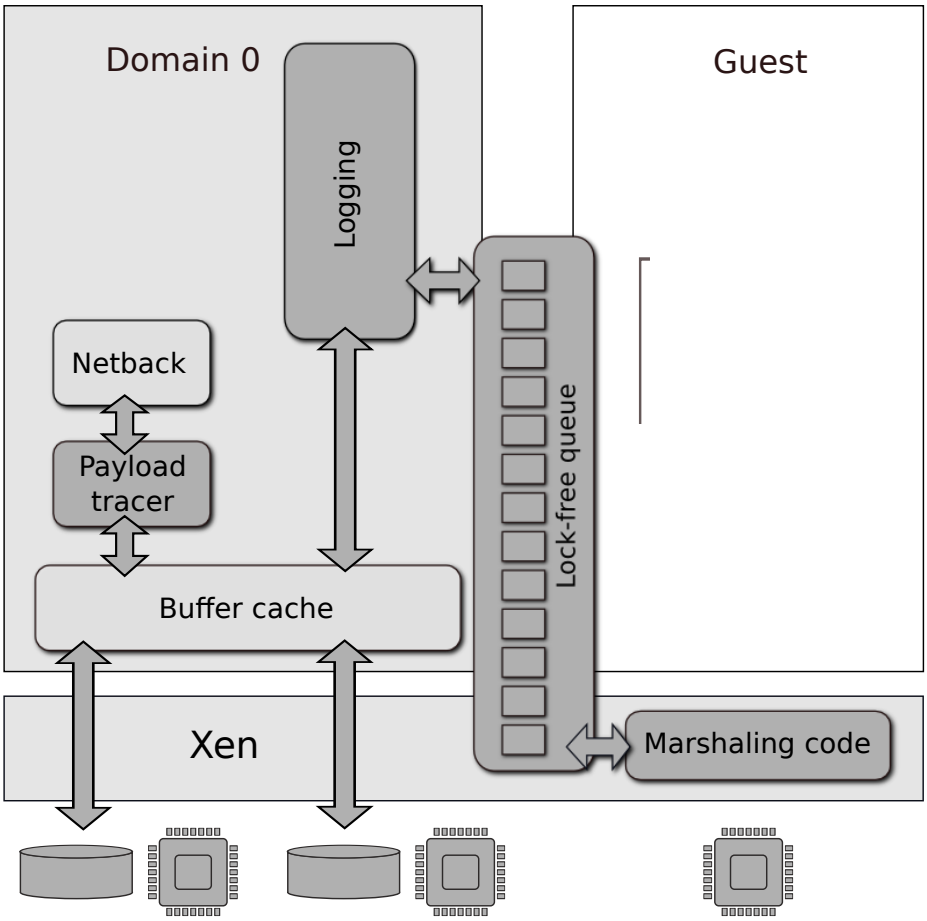

Figure 7.6. Low-overhead logging infrastructure

page encodes information about pending interrupts, interrupt masking, time, and memory grant operations. This information is an important source of nondeterminism, and is updated throughout the entire hypervisor code.

To detect updates to the shared memory pages, XenTT extends the hypervisor with support for automatic page protecting (*guarding*), and unprotecting for every transition into and out of the hypervisor (Figure 7.7). XenTT extends the `share_xen_page_with_guest` function call to track all pages shared between the hypervisor and the guest system. XenTT then instruments all the entry and exit points into and out of the Xen hypervisor. When the guest system drops into the hypervisor, XenTT guards the pages. This way, if the hypervisor tries to do an update of the shared state, it is detected with the page fault.

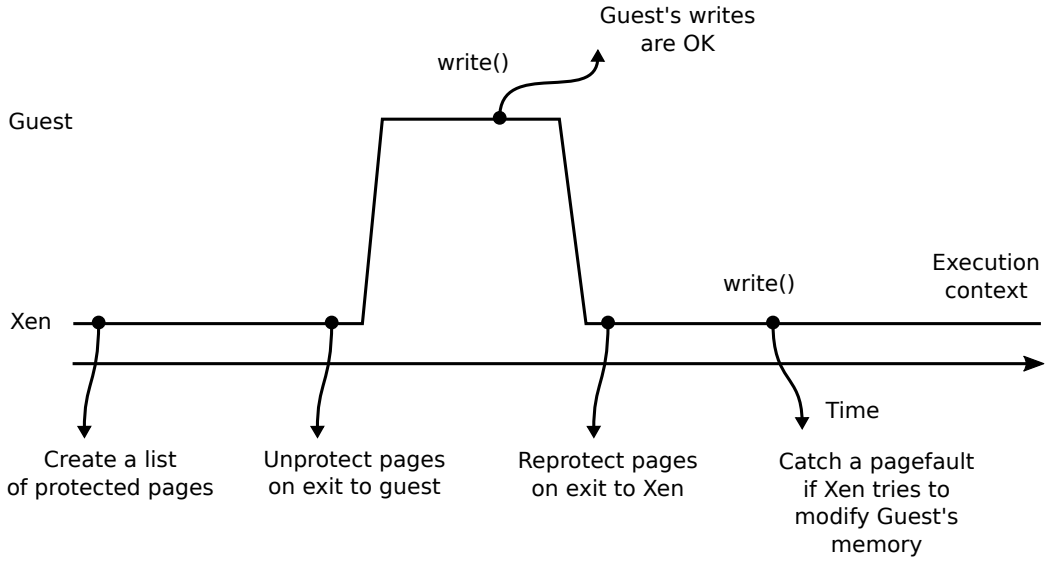

Figure 7.7. A timeline of the page guarding mechanism.

Of course, the guest system might update its shared state too. Since such updates are deterministic, XenTT unprotects the shared pages before exiting to the guest.

Several points in the hypervisor entry/exit code require instrumentation with the guard/unguard functions. A guest system transitions into the hypervisor through one of the following entry points: exception handlers, interrupt or SMP interrupt handlers, hypercalls, and hypercall continuations. The hypervisor has a single exit point into the guest. All of the above entry and exit points require instrumentation in low-level parts of the hypervisor implemented in assembler (`entry.S`). XenTT, however, can leverage the fact that it implements the logging interposition code in this point, and therefore does not require additional instrumentation.

7.5.2 Branch Tracing Store

Branch Tracing Store is a hardware mechanism provided by the Intel CPUs. BTS enables recording of all branches taken by the CPU in a memory buffer. BTS interface allows to configure a region of memory to either serve as a circular buffer, or as a linear array, in which CPU traces all taken branches. When an array representation is used, BTS can be configured to send a nonmaskable interrupt when the array reaches a certain length.

XenTT extends the Xen hypervisor with support for the Branch Tracing Store. XenTT

adds the functionality, which is required to configure the BTS store, and to handle non-maskable interrupts. If BTS tracing is enabled, XenTT flushes the content of the BTS buffer every time it traces a nondeterministic event. This way, the analysis is able to see what code the system was executing between nondeterministic updates. If no events are recorded for a long period of time, XenTT receives a nonmaskable overflow interrupt, and flush the content of the BTS buffer from the interrupt handler. To save content of the BTS buffer, XenTT relies on the general logging functions, which are designed to be a general mechanism for relaying a stream of data from the Xen hypervisor up to the logging daemon running inside Domain 0.

7.5.3 Processing and comparing execution traces

BTS saves the trace in the following form:

branch: {`from address, to address`}

To resolve machine addresses into readable symbol names, i.e., function names and source code line numbers, a special tool is built. The GDB debugger is used as a library to parse and resolve the application's symbol information into symbol names. The way to utilize GDB as a library is to start the GDB process from inside of the trace reading application (`ttd-read-log`), and use the Machine Interface library (`libmigdb`) to provide a library interface to the GDB server. The Machine Interface library is modified to support a specific request, which is required to print the trace. Finally, standard text comparison tools like `vimdiff` are used to compare the execution traces. They work quite well in practice.

7.5.4 Using BTS as a low-level debugging mechanism

Despite the fact that XenTT primarily uses BTS to record branches executed by a time traveling system, this work implements support for using the BTS for tracing the execution of Domain 0 and the Xen hypervisor. The BTS trace is especially effective for aiding debugging of a faults for which neither a stack trace nor a meaningful value of the instruction pointer are available. For example, the author of this dissertation has

successfully analyzed an error in the implementation of the interposition boundary, when the interposition code was invoking a NULL pointer from an interrupt context inside Xen.

CHAPTER 8

ANALYSIS INTERFACE

This dissertation argues for a new approach to dynamic systems analysis. A replay analysis paradigm offers a precise algorithmic approach for analysis of complex software systems. Once recorded, the comprehension of an anomalous execution is aided by an automated search algorithm, which is free to leverage a variety of techniques aimed at the exploration of a complete execution history of the system and its state.

Several changes introduced by the deterministic replay are important for understanding of how the interaction between the analysis algorithm and the execution state of the system changes compared to the traditional dynamic analysis. First, the analysis algorithm is no longer restricted to base its reasoning on a small subset of the system state. The entire execution history and the complete state of the system is available for the analysis logic. Second, the analysis algorithm is not limited in its computational complexity by a requirement to keep the run-time overhead low. Running off-line, the analysis algorithm is free to explore the hypothesis space, and reconstruct the finest details of the system's execution. Third, the analysis algorithm can iteratively go backward and forward in time. Determinism of execution guarantees that individual passes will operate on the identical state.

The goal of this work is to provide a general analysis interface capable of supporting a wide range of dynamic systems analyses, which can operate over the instance of replayed execution, and access the entire execution history of the system, access complete state of the system at every point of execution, access both high-level and low level semantics of the system, and support iterative analysis over multiple replay runs. This dissertation develops the replay analysis environment as a low-level mechanism aimed at extracting the information from the machine-level trace of replayed execution.

8.1 Static versus dynamic representation of execution

A fundamental decision one has to make when designing an interface between the analysis algorithm and the execution of the system is whether the analysis will operate against a live instance of the system's execution, or over a static trace. The ability to record a complete trace of the execution turns traditional dynamic analysis into the discipline of the *execution mining* [113] – an analysis algorithm views the recorded execution as a static trace, against which it builds a set of queries. A query language similar to SQL can be constructed to ask questions about the control flow and state of the program over such static trace [122, 79].

A number of dynamic analyses can be implemented by reasoning over the trace of – transitions in the system's state. For example, a control flow integrity analysis, which is described later in Section 10.2.2.2, builds its reasoning based on a trace of the call/return pairs.

Despite the fact that execution mining is a powerful paradigm, the static representation of the execution as a trace is not ideal for analysis of large software systems, and long execution traces. Live re-execution of the system during replay provides an opportunity to leverage components of the highly optimized virtualization stack to use them effectively as a fast information extracting and query generation engine. Several reasons argue for use of live re-execution for replay analysis:

- **Replay is fast:** Most dynamic analyses are structured to iteratively ask several concrete and relatively small questions about the system. The answer of the analysis is a composition of these small questions. It is more performance and space efficient to re-execute the system and regenerate the trace required for the specific questions than to store the entire state of the system as a static trace with the query processing mechanism.

- **System state is real:** Deterministic replay recreates the state of the system at every step of its execution in the system's memory and external devices. Operating against the real instance of execution, an analysis algorithm is free to access any part of this state, reconstruct the semantic information out of the low-level state, and make

a decision. For example, an analysis algorithm may want to reconstruct the kernel process list to identify a processes hidden by a rootkit mechanism. It is possible to provide a layer emulating state accesses for the trace-based approach, but this requires a number of complex mechanisms.

- **Execution is real:** Like any other virtual machine, the guest system is running live during replay. Deterministic replay engine tightly controls the execution of the system: the system will receive only the inputs, which it received during the original run, delivered at exactly the same positions in the instruction stream at which they happened in the original run. But otherwise, a replayed guest is a normal virtual machine. This opens several interesting opportunities for analyses, which are nearly impossible to build with static traces. For example, deterministic replay provides an option to re-run execution on the real hardware and collect hardware-level metrics like a number of TLB and cache misses. Another attractive opportunity is the ability to change the execution of the system with the goal to analyze the effects of the change. The technique of replay with modifications can be used for concolic testing of large software systems [38], automatic malware unpacking [156, 101, 100], and delta debugging [192]. Further, it is possible to use dynamic memory protection mechanisms to significantly improve performance of the data-flow analysis [90].

XenTT relies on these features to provide a practical platform for functional and performance analyses of whole systems. XenTT recreates the execution and state of a guest virtual machine precisely by appropriately replaying the inputs to the guest. As detailed in Section 8.7, XenTT creates a realistic performance model by replaying the system on real hardware and recreating the hardware conditions of the original run. The performance of a replayed execution is generally not identical to that of the original run, but changes in a predictable manner and is sufficiently precise for many whole-system analyses.

8.2 XenTT analysis interface

While designing the analysis interface for the XenTT project, this work combines insights from both the static execution mining systems [112], and dynamic replay re-

execution engines [43, 44, 20]. XenTT runs analysis algorithm against an instance of the live execution of the system, which is re-created by the replay engine (Figure 8.1).

The analysis algorithm runs outside of the guest system. The analysis observes execution of the system, and accesses its state through a Virtual Machine Introspection (VMI) interface. Since replay runs live, the entire state of the guest system is available for analysis at every point of execution. The VMI interface is responsible for resolving multiple layers of virtual addresses and providing a transparent access to the virtual memory inside the guest system.

To simplify development of the analysis algorithms, the raw access to the guest's memory is complemented with the symbol names interface, i.e., the ability to access the state of the system through high-level symbol names which were used in the source code of the system during its development. XenTT implements a powerful symbol resolution library which parses a debug symbol information of the application or a kernel running inside the guest system, and provides an analysis algorithm with a convenient state access interface.

To extend replay analysis with an option to reason about performance properties of the system, XenTT extends a traditional replay protocol to capture enough information about the original execution, so it can recreate a nonfunctional property of the system – *performance*. The goal of this work is to provide analysis with a realistic performance model, which can be queried on different performance properties at any moment of execution.

XenTT structures the analysis algorithms as a set of probes, which extract a trace of relevant events from the execution of the system during replay. The analysis algorithm builds its decision logic based on the observations of the trace of events generated by the probes, i.e., control flow transitions and data accesses inside the system. At the moment, XenTT supports development of the analysis algorithms in the C language. However, the XenTT interface is language-, and system-independent.

Figure 8.2 describes details of the XenTT analysis stack. The analysis algorithm runs as a user-level process inside the privileged Domain 0 domain. The analysis code is linked against the VMI and symbol resolution libraries. In addition to providing virtual address resolution inside the guest system, the VMI layer is responsible for installing breakpoints, providing run-time environment for the probe handlers, and mapping pages from the guest

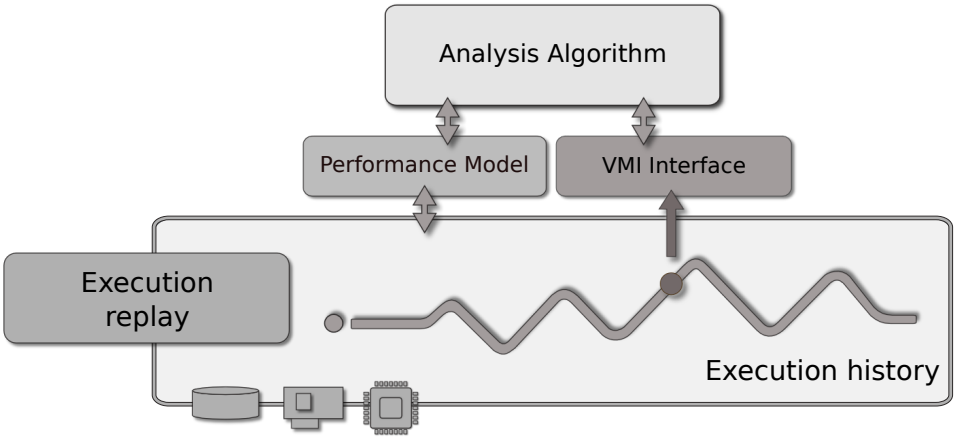

Figure 8.1. Analysis interface

virtual machine into the address space of the analysis process. The sysmbol resolution library, `dwdebug`, provides resolution of the symbolic names into virtual addresses inside the guest system.

8.3 Monitoring and accessing the state of the system

The XenTT analysis interface provides two basic primitives: the probe mechanism aimed at extracting relevant traces, and virtual machine introspection framework designed to provide a convenient interface for accessing the state of the system. A variety of trace generating, indexing, caching, and querying mechanisms can be built on top of these two basic primitives.

8.3.1 Probes

Probes provide an analysis algorithm with a way to observe transitions in the state of the system. An analysis algorithm registers a set of probes to watch for events of interest. Each probe has a handler associated with it, which is invoked by a library run-time when the system tries to execute or access a probed state.

XenTT follows a practical choice, and supports two kinds of probes: breakpoints and watchpoints. Breakpoints and watchpoints implement semantics of the traditional debugging data and code breakpoints. A code breakpoint mechanism provides an option to monitor when the system tries to execute a specific part of code. Similarly, a watchpoint is

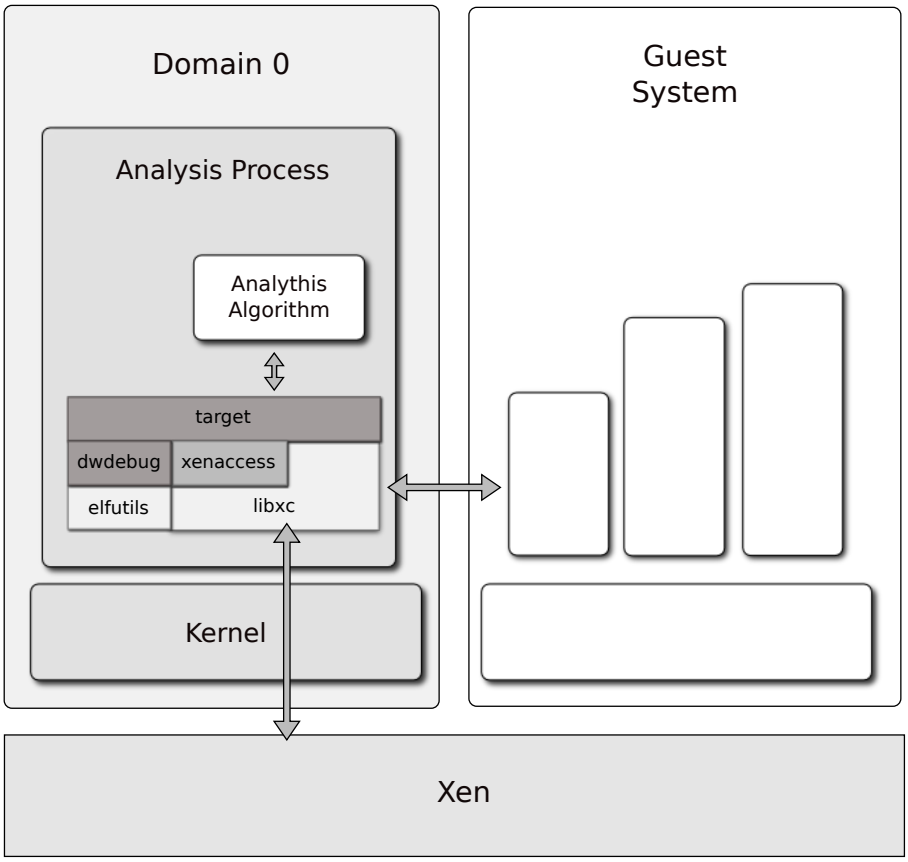

Figure 8.2. Overview of the analysis architecture

triggered when a specific memory location is accessed.

Both breakpoints and watchpoints have reasonably low overhead, and little implementation complexity if compared to finer ways of tracking the state of the system.

8.3.2 Native symbol names

Native symbol names are aimed to provide a convenient programming interface for development of the analysis algorithms. The goal is to create an illusion that analysis algorithm runs as part of the system, which it analyzes, and can access its state in a nearly transparent way.

A variety of representations of the system's state is possible. One side of the spectrum is a raw bit-level representation – an analysis algorithm receives no support for extracting semantic information from the system's state. The bit-level representation is often necessary for analyzing an adversary code with no clear semantics, e.g., a polymorphic malware, or

a return oriented code. In this case, the analysis algorithm is responsible for reconstructing a high-level semantics from the raw memory and execution trace [38, 160].

On the opposite side of the state representation spectrum is a complete fusion of the analysis algorithm with the system it analyzes. Compile- and run-time techniques can be used to compile the analysis algorithm against the system's source code, and provide a complete illusion that analysis runs inside the system [152]. Source-level integration provides the analysis algorithm with an opportunity to re-use implementation of the system to access its own state, e.g., rely on the system-defined list walking functions to iterate through the list data structures from inside the analysis algorithm.

Native support for symbol names belong to the middle of the state representation spectrum. The idea is to let the analysis algorithm transparently use the symbol names which were used in the source code of the system. Two mechanisms are required to implement this transparency: virtual machine introspection, and symbol name resolution.

XenTT implements a virtual machine introspection and symbol resolution layer, which resolves symbolic names used in the code of the system into memory and register locations during an actual system run. XenTT relies on the debug-symbol information to reconstruct functions and data-structures from the binary state of the guest system. XenTT implements a powerful debug-symbol, and virtual machine introspection library, which provides the analysis algorithm with the ability to access a binary state of the guest through a familiar symbol names. To simplify development of new analysis algorithms, the core library provides a set of general primitives, which support common exploration patterns, e.g., breakpoints, watchpoints, and control-flow integrity. This work envisions that an extensible set of analysis libraries will abstract many aspects of the low-level system and application-specific state.

8.4 Principles of virtual machine introspection

Virtual machine introspection and symbol resolution interfaces are designed to provide the analysis algorithm, which runs outside of the guest system, with the transparent access to the system's state. Several layers of translation mechanisms are required to access a variable within an application or a kernel running inside the guest virtual machine (Figure 8.3):

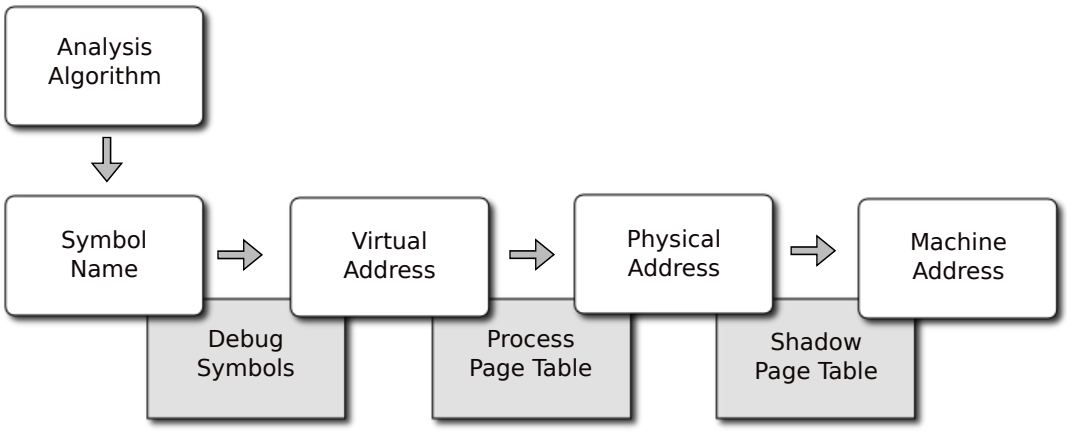

Figure 8.3. Multistage process of symbol resolution.

- **From symbol names to application memory:** The analysis algorithm enters the VMI resolution pipeline with a string name of a variable it tries to access, e.g., `jiffies`, `sys_open`. To access the requested variable, XenTT needs to identify its memory or register location storing the variable at a specific moment of execution, inside the virtual address of the process running inside the guest system.

  The XenTT symbol resolution layer runs in front of the VMI library. Given a symbolic name of a variable, the symbol resolution library returns the virtual address of the variable in the address space of the guest process, or kernel. The symbol resolution library performs this step by utilizing the debug symbol information. XenTT relies on the presence of the debug information file, which is typically generated by the generated by the compiler as a DWARF file. At the moment, XenTT supports DWARF format generated by the GCC compiler.

- **From application memory to guest physical memory:** When the virtual address of a symbol is known, it needs to resolve it into guest's physical address. To perform this step, XenTT relies on the `xenaccess` library, which is capable of locating and walking the page table data structures inside the guest system.

- **From guest physical memory to machine memory:** The last step of translation is to resolve a physical address in the guest's memory into a unique machine address.

XenTT utilizes the Xen memory translation mechanism to do this step. An output of the translation pipeline is a unique frame number – page number of a page which contains the required data.

After the machine frame number of is known, the VMI layer uses the mechanisms provided by the Xen virtual machine to map this page into the address space of the analysis process. The VMI library reads the value of the variable at the corresponding offset withing the page, and returns it to the analysis algorithm.

The VMI library is responsible for resolving multiple layers of virtual memory addressing inside the guest system. The VMI library takes a virtual address inside an address space of a process or a kernel running inside the guest system, and returns a machine frame number, which can be mapped inside the address space of the analysis process.

XenTT supports symbol and address space resolution for the variables inside the guest kernel. The Flux research team continues working on extending XenTT with support for accessing guest's user-level processes.

8.5 Architecture of the XenTT analysis stack

The XenTT analysis stack consists of five libraries, and some functionality provided by the Xen hypervisor (Figure 8.4).

8.5.1 Targets

To implement a virtual machine introspection stack, which can operate on various binaries, XenTT introduces a notion of the *target*. A target implements an abstraction layer, aimed to hide functions specific for accessing the state of various analysis targets. XenTT supports two targets: a normal Linux process, and the Linux guest virtual machine. The target library makes sure that analysis code can run against any of the supported targets, and of course against live and replayed virtual machines. The `target` library provides functions for attaching to a specific process or virtual machine target, and registering probes.

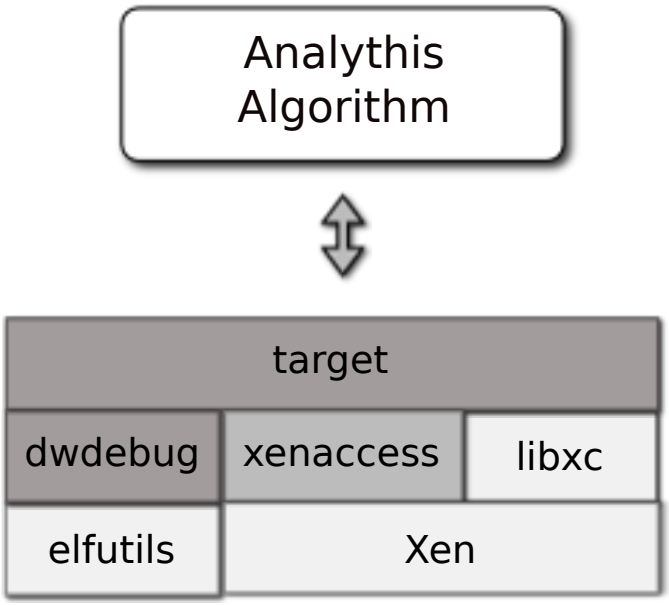

Figure 8.4. Architecture of the XenTT analysis stack.

8.5.2 Symbol resolution: dwdebug and elfutils

Two libraries of the XenTT stack are responsible for resolution of the symbol names: `dwdebug` and `elfutils`. XenTT relies on the functionality of the `elfutils` library to parse the DWARF debugging data format [66]. DWARF files are generated by the compiler along with the binary of the program. The DWARF format contains description of where the specific functions, and variables are located inside the program binary.

The `elfutils` library provides a low-level access to the debugging and symbol information. To export an fast lookup of the symbol names, XenTT implements the `dwdebug` library.

8.5.3 Resolving the virtual, physical, and machine addresses

The `xenaccess` library and the Xen memory management system are responsible for resolving virtual addresses inside the guest system into guest physical and machine addresses. The `xenaccess` library implements support for resolving guest virtual into guest physical addresses. `xenaccess` is able to locate a page table data structure in the address space of the guest system. It then walks the page table to resolve the virtual address into physical.

The Xen shadow memory interface is responsible to provide translation between guest physical and system machine addresses.

8.5.4 Reading and writing guest's memory and registers

The `libxc` library provides a user-level interface to the functionality of the Xen hypervisor. XenTT relies on the `libxc` library to implement memory mapping of the guest pages into the address space of the analysis process, accessing and changing state of the register file of the guest system, and registering for the notifications on the debugging interface.

8.6 Probes

Probe handlers run in a run-time environment provided by the XenTT VMI library. The purpose of the VMI run-time is to convert low-level debug exceptions from the Xen hypervisor into high-level upcalls to probe handlers. The analysis algorithm is therefore abstracted from all low-level details involved into handling probe invocations.

The run-time environment of the VMI library provides an analysis algorithm with an illusion that it executes over the stream of events, which it subscribed to. In the future work, the author of this dissertation envisions that analysis algorithms will transparently run on either a dynamic stream of events generated by re-executing the system during replay, or by observing a stream of events generated statically by other means.

8.6.1 Xen-level probe interface

A general idea of a probe is to upcall an analysis handler when execution reaches a certain point in the code, or tries to access a watched variable. Multiple layers of the Xen virtualization stack are involved in triggering and processing of the probes. The following stages are involved into processing of a probe: i) intercept control of the guest system; ii) invoke a user-level analysis handler; iii) emulate original instruction; iv) return control to the guest. It should be emphasized that probes must remain transparent for both the guest system and determinism of the execution during replay.

8.6.1.1 Installing a probe

XenTT supports two kinds of probes: data and code breakpoints. XenTT uses both a hardware and software breakpoint mechanism to implement the code breakpoints. However, at the moment, XenTT supports only hardware debug registers to implement data breakpoints. In other words, XenTT does not support page-level memory protection mechanisms to implement data watchpoints.

To install a hardware data or code breakpoint, XenTT relies on the `libxc` library to access the guest's register file. XenTT makes sure that the Xen hypervisor correctly handles updates to the breakpoint registers, and updates the hardware registers for the guest domain on the domain context switch.

To install a software breakpoint, XenTT uses the `libxc` library to map the content of the memory page, which contains the probed instruction, into address space of the analysis algorithm. After the page is mapped, XenTT saves the original instruction, and overwrites the first byte of the probed instruction with the software debug interrupt instruction (`0xcc`).

8.6.1.2 Handling the debug exception

When the guest system reaches a probed code or memory location, e.g., tries to execute a software breakpoint instruction, the Xen hypervisor receives a debug interrupt exception. Xen suspends execution of the guest system, expecting that a debugger process will unsuspend it later. Xen signals a virtual interrupt on the debugger event channel `VIRQ_DEBUGGER`. The XenTT VMI library registers to listen for the debugger event channel notifications.

When VMI library receives a debugger event, it tries to find either a probe handler, which is registered at the address matching the current value of the instruction pointer (`EIP`) register, or tries to find a corresponding data probe handler, which is registered on a memory location saved in one of the first four debug registers (`DR0` – `DR3`), and signaled via the debug condition register (DR6).

8.6.1.3 Probe handler invocation

Since the system is suspended for the debugger by the Xen hypervisor, the probe handler can be safely invoked by the VMI library run-time. Even if the handler tries to

access the state of the system, the access is atomic with respect to the state of the system. The VMI library trusts the handler to eventually return control back to the library.

8.6.1.4 Emulation of the original instruction

If the software breakpoint mechanism was used for installing the probe handler, the original instruction on the probed location should be emulated by the VMI library. XenTT emulates the original instruction in the most straightforward way. XenTT restores the original instruction in the memory of the guest system, and lets it run in a single-step mode to make sure that XenTT gains control back after executing this one instruction. To single-step the system, XenTT sets the trace flag (`TF`) in the flag register of the guest system.

After executing the original instruction, and gaining back the control over the guest system, XenTT reinstalls the software breakpoint by writing the `0xcc` single-byte instruction code in place of the first byte of the probed instruction. The VMI library clears the trace flag, and resumes the guest system.

8.6.2 Determinism of execution in face of VMI

It is important to ensure that determinism of replay holds in face of processing debugging exceptions, and single-stepping introduced by the VMI probes. Before installing a software or hardware probe, the VMI library registers the probe points with the Xen hypervisor.

XenTT extends the Xen exception handlers to conceal the debug exceptions generated by the VMI probes from the guest system, and instead redirect them to the VMI library. To recognize a VMI probe exception, XenTT uses two techniques. For the software breakpoints, e.g., a single-byte 0xcc instruction, XenTT compares the current value of the instruction pointer with the address of the VMI probe. For the hardware breakpoints and watchpoints, XenTT compares the values of the hardware debug registers with registered VMI probe points.

8.7 Performance modeling

Replay protocols ensure that the off-line copy of execution is identical to the original run; i.e., during replay, the system repeats its original execution instruction by instruction. An obvious limitation of this approach is that during replay, any dynamic analysis is limited to reason only about functional properties of a system. To enable performance analysis of a system during replay, XenTT extends a traditional replay protocol to capture enough information about the original execution so it can recreate one of the nonfunctional properties of the system – *performance*. The goal is to provide analysis with a realistic performance model, which can be queried on different performance properties at any moment of execution.

8.7.1 Re-execution approach to performance

A variety of approaches is possible for recreating performance characteristics of the system during a replay run. The spectrum ranges from replaying a system under control of a cycle-accurate CPU simulator to controlled execution on the real hardware. Instead of replicating a complex hardware state of the CPU in a simulator, XenTT aims to recreate a faithful performance model by replaying the system on the real hardware, and recreating hardware conditions of the original run. Replay on the real hardware ensures a simpler replay infrastructure, better replay performance, and higher realism of the performance model. Several factors change the performance of a system during replay even if it runs on the identical hardware:

- Performance of modern superscalar processors greatly depends on the quality of the code they execute. A number of complex hardware optimizations is involved in optimizing execution of a continuous instruction stream. Execution preemption, interference between multiple instruction streams, execution of additional layers of virtualization, and replay code pollute the internal state of the CPU and change performance of the system across three hardware environments: physical hardware, virtualized machine interface, and virtual machine during replay. It is obvious that some microlevel performance metrics cannot be observed identically across three environments. Nevertheless, the macrolevel performance properties of the system

remain similar across three environments and the re-execution approach to performance analysis is feasible.

- Replay eliminates all periods of idle execution from the execution history of the system. First, all external I/O events are retrieved from the log file. This eliminates idle wait cycles from the execution of the replayed system. Second, replay erases all periods of inactivity when the system halts the CPU. As a result, any workload which is not limited by the lack of CPU power progresses faster [64, 187, 43].

  To account for this change, it is possible to record exact times at which the system starts periods of continuous nonpreempted execution. The recorded time values create a reference model of the system's performance. The challenge is to ensure fidelity of the model on the intervals between the reference timestamps.

- To repeat its original run, the system is executed in a controlled deterministic environment. Replay mechanisms inject nondeterministic events at the exact points they happened during the original run. Controlled execution of the system between nondeterministic events relies on the hardware instruction counting, and single-stepped execution. The system is preempted multiple times. Every preemption affects performance of a system.

XenTT recreates performance properties of the original run by re-executing the system on the same hardware across original and replay runs. XenTT's performance model is based on the following observation. At every *replay interval* – an interval of replayed execution between two consecutive nondeterministic events – the following performance behavior is expected. Starting an interval, the system quickly recovers from the effects of the exit into the hypervisor, and gains performance characteristics of the original run. Approaching the end of the interval, the system is preempted with a branch counter exception and continues in a single-step execution mode. The end of the interval is affected by branch counting and debugging exceptions. Although a realistic performance is unrecoverable at this very last part of the interval, this work argues that this is an acceptable imperfection of the XenTT's approach, since it loses performance fidelity for only a dozen or so basic instruction blocks.

8.7.2 Hardware performance counters

The performance monitoring interface is a primary mechanism designed to assist performance analysis of the compiler and low-level system optimizations [51, 49]. Hardware performance counters export a number of metrics reporting the state and performance of internal CPU optimizations (see Appendix B of the IA-32 Intel Architecture Optimization Reference Manual [50] for a complete description of performance monitoring interface of the Intel architecture):

- **Time:** CPU nonsleep and unhalted clock ticks, and number of retired instructions, number of nonspeculative and speculative retired μops. These basic time counters provide a way to compute the most widely used CPU performance metric: rate of execution (number of instructions executed per cycle).

- **Branch prediction unit:** Number of predicted and mispredicted branches, calls, returns, and conditionals.

- **Trace cache and front end:** Number of instruction TLB misses, number of page walk requests caused by TLB misses, number of trace cache flushes, misses, speculative μops, and number of cycles trace cache spends building and delivering μops.

- **Memory subsystem:** Number of data TLB misses, number of page walks caused by data TLB misses, number of μops that experienced 1st or 2nd level cache load miss, number of μops that experienced data TLB load or store miss, number of loads replayed due to a state of the memory ordering buffer, number of 2nd and 3rd level reads and read misses, and number of 2nd and 3rd level cache references resulting in shared, modified, and exclusive hit.

- **Bus:** Number of all CPU bus transactions, number of bus transactions caused by non-prefetch operations, number of front-side bus clocks that the bus is transmitting data, number of all reads and writes, non-prefetch reads, write combining, and uncacheable memory transactions, The counters above provide a way to compute prefetch ratio, and bus utilization metrics.

- **Code characterization metrics:** Number of executed floating-point, SSE, and MMX operations.

- **Pipeline clear events:** Number of cycles that the entire pipeline spends being cleared for all reasons, number of times the pipeline is cleared due to memory ordering, and self-modifying code.

Despite the fact that the performance monitoring interface is designed to reason primarily about the performance of the CPU, a number of the reported hardware events can directly assist performance analysis of low-level software mechanisms. Time and rate metrics are traditionally used by performance analyses as general indicators of performance hot-spots. Counters reporting the performance of bus, branch prediction, trace cache, and machine pipeline units are used as indicators of poor system architectural choices, which are incoherent with existing CPU optimization technologies. Memory subsystem events are critical for analysis of low-level communication, synchronization, and memory protection mechanisms.

A general approach to virtualization of hardware performance counters is re-execution of the system on the real hardware. Performance events can be divided in two classes: depending and not depending on execution history and prior state of the CPU. Performance monitoring events like time, some of the bus transaction not involving request buffering, code characterization metrics, and pipeline clear events do not depend on the prior state of the CPU and therefore are likely to be recreated accurately during replay.

Other hardware counters depend on the execution history and prior state of the various units of the CPU. During replay, these units reach a state similar to the original run; however, the length of the execution history required to recreate these condition depends on the size and semantics of unit's "memory", and frequency of updates to this memory. Semantics and size of caching and buffering units like memory caches, TLBs, and write buffers is relatively simple. Semantics and size of branch predicting and trace cache units is rather complex.

8.7.3 Virtual performance counters

XenTT provides a uniform interface to the performance of a system during replay via abstraction of virtual performance counters. Virtual counters virtualize the hardware performance counters exposed by modern CPUs as a low-level performance interface [51, 49]. In contrast to hardware counters, virtual counters account for the impact of replay mechanisms on the performance of a system, and report performance as it would be observed during the original run. Virtual counters provide a general mechanism to translate performance of a system between replay and original runs.

For most counters, XenTT assumes a simple linear model of performance. The performance counter virtualization algorithm reads a value of the hardware performance counter at the beginning and the end of a *basic replay block* – a period of replay, during which the system runs on the real hardware without being preempted. XenTT attributes all events registered by a hardware counter to the system under replay and add this value to the virtual counter.

The value of a virtual counter V at any moment of execution now is provided as the sum of the virtual counter at the beginning of the basic replay block, i.e., on exit to guest (V_{exit}), and increment of the corresponding real hardware counter from the beginning of the replay block (R_{exit}) to the point at which the measurement is taken (R_{now}):

$$V_{now} = V_{exit} + (R_{now} - R_{exit}) \tag{8.1}$$

8.7.4 Recording and replaying the performance information

In contrast to other counters, the values of which can be recomputed during replay according to the equation (8.1), the progress of the time stamp counter cannot be recreated, and needs to be recorded and replayed by the replay system. Compared to the original run, the execution of the guest system can progress faster or slower during replay, and unless recorded, this information is lost. Two main factors which affect the performance of the system are: (1) eliminated periods of I/O waits and CPU inactivity make the system faster, (2) frequent periods of single-stepping required for replay of asynchronous events slow the system down.

To recreate a faithful view of time during replay, XenTT records and replays information about performance of the system during original and replay runs. XenTT records the time information for every exit from the hypervisor into the guest system, and for every enter from the guest system to the hypervisor. Fortunately, for each pair of transitions, XenTT needs only one actual event.

On every exit into the guest, XenTT saves the current value of the time stamp counter register (TSC). Later, when the guest system enters back in the hypervisor, XenTT creates and event of type `TTD_UPDATE_BRCTR`, and piggybacks information about the time of last exit into the guest on this event (Figure 8.5).

During replay, the progress of the time stamp counter is replayed by the replay system for every exit from the hypervisor into the guest (Figure 8.6). The replay daemon parses the event stream and injects a new `TTD_UPDATE_TSC` event for every `TTD_UPDATE_BRCTR` event it sees. The `TTD_UPDATE_TSC` event is replayed every time the guest system enters the hypervisor. This recreates the value of the time stamp counter (V_{exit}), which the guest system had right before exiting from the hypervisor.

Now the value of the time stamp counter, at any moment of execution, e.g., when a VMI probe is triggered at the moment now, can be recreated during replay according to the equation 8.1.

8.8 Relation to previous work

A number of tools implement binary instrumentation [31, 144] and binary translation [130, 119, 187]. On-demand CPU emulation suggests a novel way to combine native execution with binary translation [90]. Crosscut suggests idea of heterogeneous replay [44]. Crosscut captures execution of a system by means of a virtual machine monitor [187], but replays the same system under the control of Pin [119], a binary translation framework providing support for a number of popular dynamic analyses.

Existing work extends Bocsh with instruction counting to implement deterministic replay [58]. Daikon explores the ways to compute invariants about execution [69]. Existing causal-path profilers develop methods for cross-path analysis [35].

Several systems explore language support for expressing performance expectations in

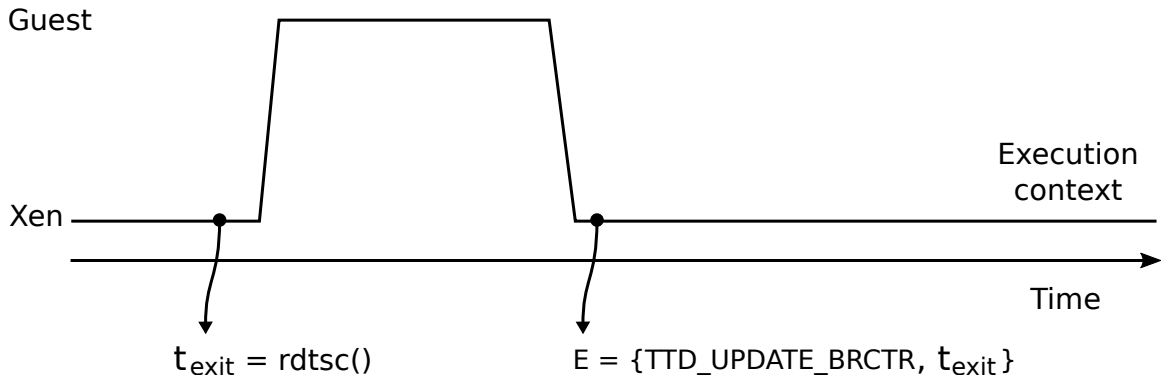

Figure 8.5. Saving and piggybacking the TSC information on exit to guest.

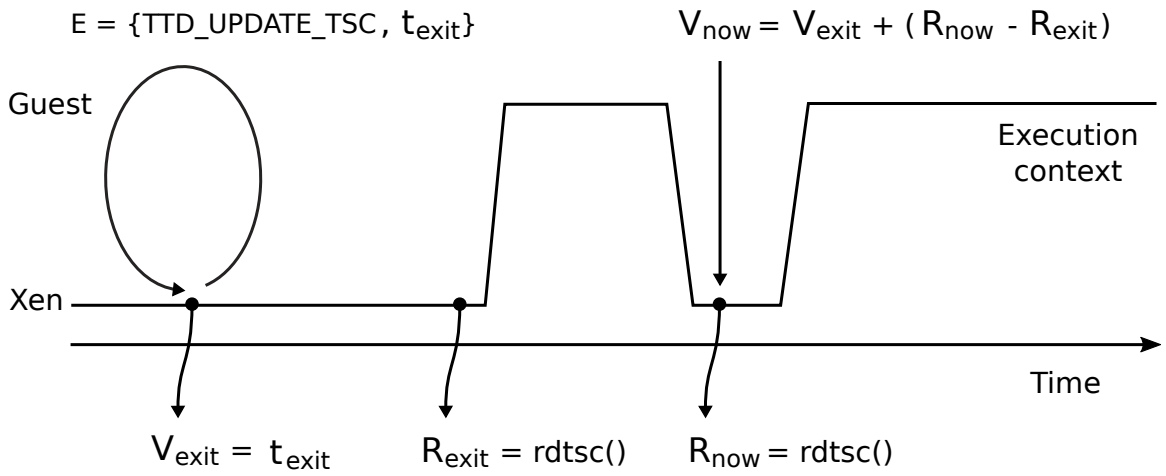

Figure 8.6. Replaying the TSC information.

the form of assertions [141, 173]. Assertions can be checked at runtime [173] or off-line against the trace of collected performance metrics [141]. Crovella and LeBlanc develop a notion of performance predicates as a part of the predicate profiling approach [53]. Using predicates, they provide a way to identify how much time a parallel program spends in one of the inefficient states, e.g., due to synchronization, load imbalance, communication, or insufficient parallelism. Crovella and LeBlanc also argue against the "measure-modify" approach to performance profiling [54].

A number of existing profilers rely on hardware instruction counters to implement light-weight, fine-grain program sampling [114, 1, 2, 7]. Cycle-accurate hardware simulators are traditionally used to provide a realistic performance model for an emulated processor

architecture [9, 121]. Hardware emulation of complex CPUs incurs significant slowdown. This work enables performance analysis at the speed comparable to the original execution by recreating performance conditions of the original run on the real hardware. Perhaps the closest to this dissertation is the PTLsim processor simulator, which relies on the Xen virtual machine monitor to switch between a cycle-accurate simulation of the x86 ISA CPU and a full-speed execution of a system on real hardware [190].

Recent work by Gupta et al. suggests dilation of time and network to evaluate performance of a system on future hardware platforms [87, 86]. Changing the speed of time for individual components, Gupta et al. make components run faster or slower in comparison to the rest of the system. Gupta et al. implement a coarse-grain performance model, which provides no way to reason about performance of individual parts of a system.

CHAPTER 9

EVALUATION

Can deterministic replay be applied for dynamic analysis of complex software systems? To answer this question, this chapter develops several evaluation scenarios, which demonstrate that XenTT can support deterministic recording of real systems on real workloads with overheads that are nonprohibitive for use in production environments.

9.1 Hardware setup

To evaluate XenTT, this work chooses a hardware platform which is representative for a production cloud environment. The experiments are performed on a Dell PowerEdge R710 server equipped with a quad-core 2.4 GHz Intel Xeon E5530 "Nehalem" processor with hyperthreading support, 12 GB of 1066 MHz DDR2 RAM, and four Broadcom NetXtreme II BCM5709 rev C 1 Gbps NICs. The machine is configured with three additional (besides the root disk) Western Digital WD1501FASS 1.5 GB SATA disks with a 64 MB buffer, 7200 RPM, and a sustained data transfer rate of 138 MB/s. This work uses a development version of the 32-bit Xen hypervisor (closest stable version to XenTT is 3.0.4) and a 32-bit Linux 2.6.18 kernel.

To minimize test variability, the system is configured with the minimal set of resources required to fulfill the test task. For all experiments, the Xen hypervisor is configured to detect and use only three CPU cores: two cores are allocated for Domain 0, and one core runs the XenTT guest virtual machine. To reduce caching and buffering effects, the memory allocation for Domain 0 and the guest virtual machine is configured to be 2GB and 1GB, respectively.

9.2 Logging and Replay Overheads

9.2.1 CPU intensive workloads

To evaluate recording overheads on a set of CPU intensive workloads, XenTT is configured to run the open-source, multiplatform Freebench benchmarking suite [73]. Freebench uses existing open source tools to implement tests, which measure a system on a variety of workloads: scientific, 3D rendering, compression, encryption, database, photo processing, audio encoding, text processing, and AI.

Figure 9.1 presents results of running Freebench benchmarks on an unmodified Linux guest virtual machine, and a XenTT guest with recording enabled. For all but one test, recording infrastructure introduced only a small overhead of 5.6%.

To evaluate performance of a recorded system on a set of systems workloads, XenTT is configured to record execution of the Phoronix benchmarking suite [142]. The Phoronix suite provides a large library of benchmarks. This evaluation uses ten benchmarks that characterize whole-system performance: `apache` (measures sustained requests/second; concurrent requests); `nginx` (measures sustained requests/second; 100 concurrent requests); `pgbench` (measures transactions/second; TPC-B-like benchmark of the PostgreS database); `phpbench` (large numbers of simple tests against the PHP interpreter); `pybench` (tests basic, low-level functions of Python). The component-specific benchmarks include `dbench` (file system calls to test disk performance); `postmark` (transactions on 500 small files (5–512 KB) simultaneously); `lame` (time to convert a WAV file to MP3); `gnupg` (encryption time with GnuPG); `stream` (RAM performance).

Figure 9.2 shows the performance of XenTT relative to the reference, non-XenTT performance, for each benchmark. LAME, GnuPG, and Stream remain within 2% of the performance of an unmodified Xen; under XenTT, `apache` serves approximately 69% as many requests per second as the reference implementation. `apache` benchmark incurs a large number of accesses to the timestamp counter register (TSC), via the `rdtsc` instruction. In an unmodified Xen, `rdtsc` is a nonprivileged instruction. XenTT, however, forces the guest system to exit on `rdtsc` to record the returned TSC value. It is not clear whether the large number of TSC events is inherent to the Apache workload or just a side-effect of the benchmark. Postmark performs a large number of small disk operations.

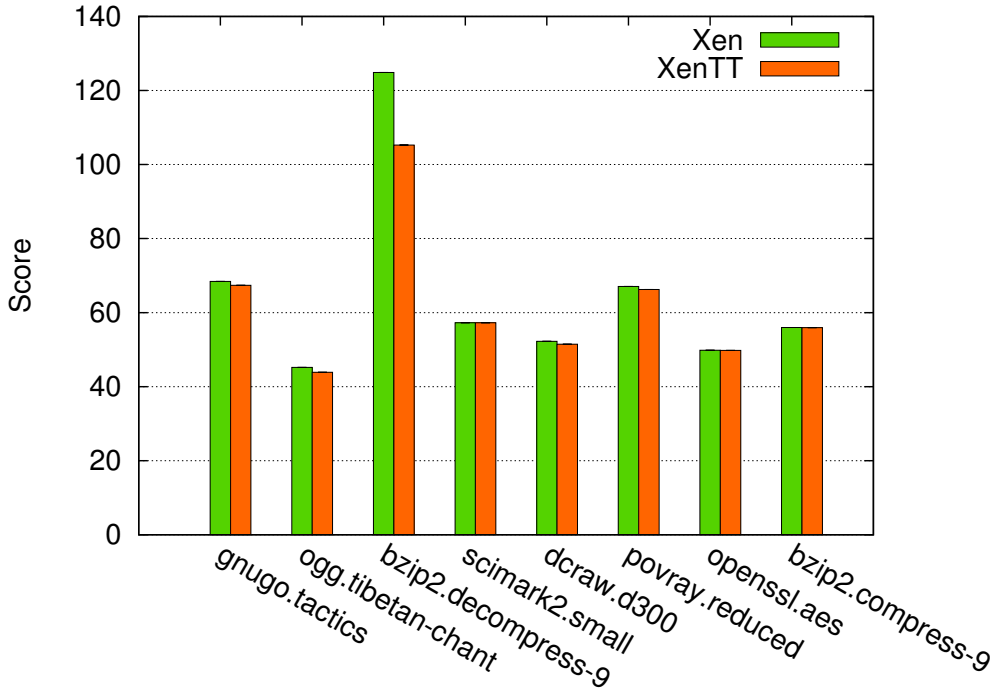

Figure 9.1. Freebench benchmarks.

Higher delays of the disk interposition code penalize its performance.

9.2.2 I/O intensive workloads: disk and network

9.2.2.1 Disk workloads

To stress sequential disk I/O, XenTT is configured to run the `dd` disk copying tool, which reads and writes a 4 GB file. `dd` uses the 1 MB block size. The tests report results averaged over three runs (Figure 9.3). The Xen disk drivers provide little buffering on the I/O path and are therefore sensitive to the delays that XenTT introduces by routing all disk requests through a user-level device interposition daemon. On a disk with the sustained data transfer rate of 138 MB/s, a vanilla Xen system achieves the write and read throughputs of 101 MB/s and 130 MB/s, respectively. The XenTT interposition layer stays within 21% and 22% of the performance of unmodified Xen.

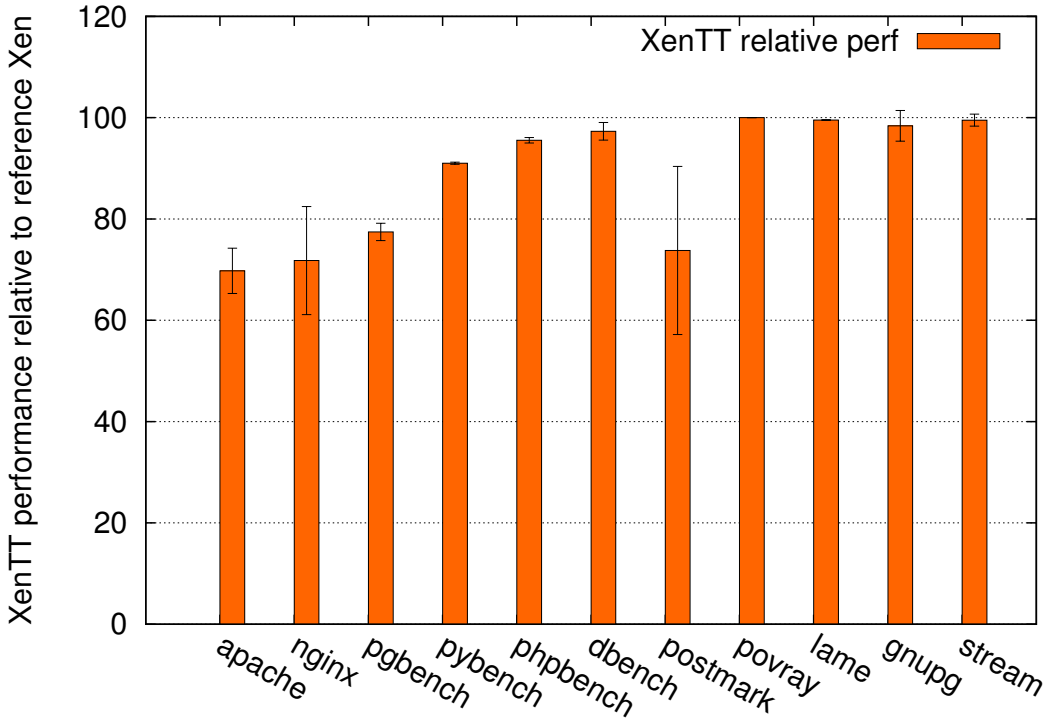

Figure 9.2. Phoronix benchmarks.

9.2.2.2 Network workloads

Overheads of network logging are evaluated by recording execution of the `netperf` network benchmark (Figure 9.4). The benchmark is configured with the TCP window size of 128 KB; run `netperf` for 60 s, and average results over three runs. Compared to disk I/O, Xen's network drivers are much more optimized to support high-bandwidth workloads and tolerate I/O delays. On send and receive operations, XenTT is able to stay within the 8% and 14% of the unmodified Xen.

To measure the network delay, XenTT records execution of the `ping` tool (Figure 9.5). The tests report overheads of interposition for both idle and loaded network path. To load the network path, the guest system runs the `netperf` TCP stream test in the background, simultaneously with the ping test. On a loaded connection, delays introduced by XenTT are within 5%. On an idle link, the interposition code adds an 80 microsecond delay.

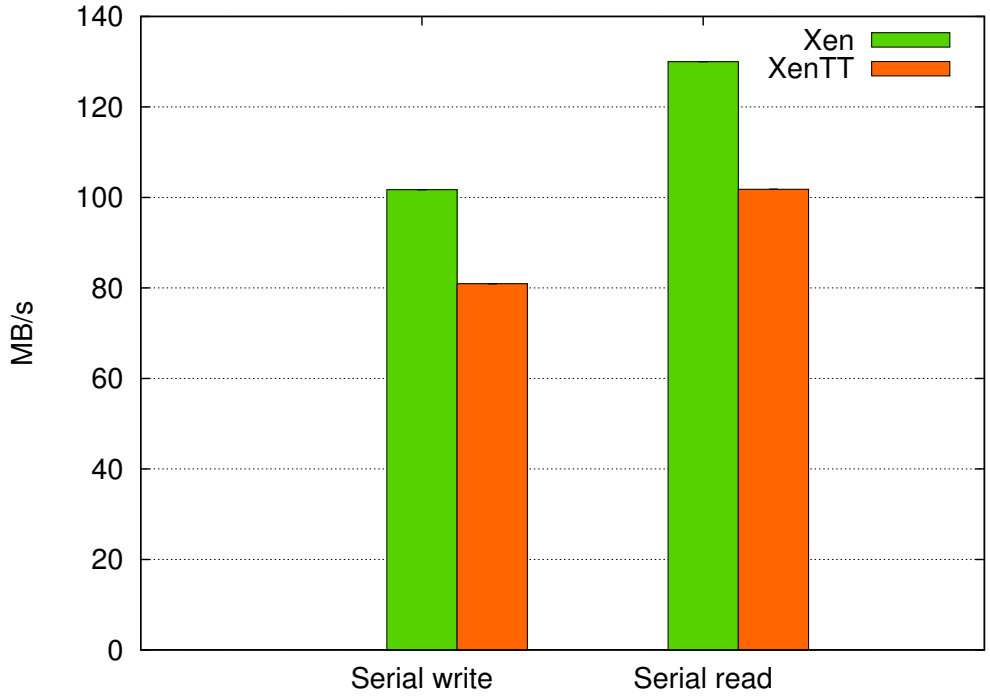

Figure 9.3. Disk throughput.

9.2.3 Discussion

Results of Phoronix benchmarks (Figure 9.2) demonstrate that overheads introduced by the XenTT recording layer are not prohibitive for applying deterministic replay analysis in a production environment. For example, the results from Figure 9.2 suggest that depending on the workload, a recording of a typical web server system, which runs the LAMP stack – a combination of the Linux operating system, a web server, a database management system, and a dynamic language interpreter – will encounter a slowdown between 5% and 31%.

The 31% slowdown is a worst case scenario if the system runs Apache web servers, and the servers are fully loaded. One might point out that the LAMP stack is a pipeline, and therefore, performance of the slowest stage defines the speed of the entire pipeline. This observation is true for the pure pipelining systems. However, modern large-scale cloud systems are designed to be elastic, i.e., scale the entire system and its parts proportionally to the load. These systems can translate the 31% slowdown of the Apache nodes into a proportional 44% increase in the number of servers required to process the Apache

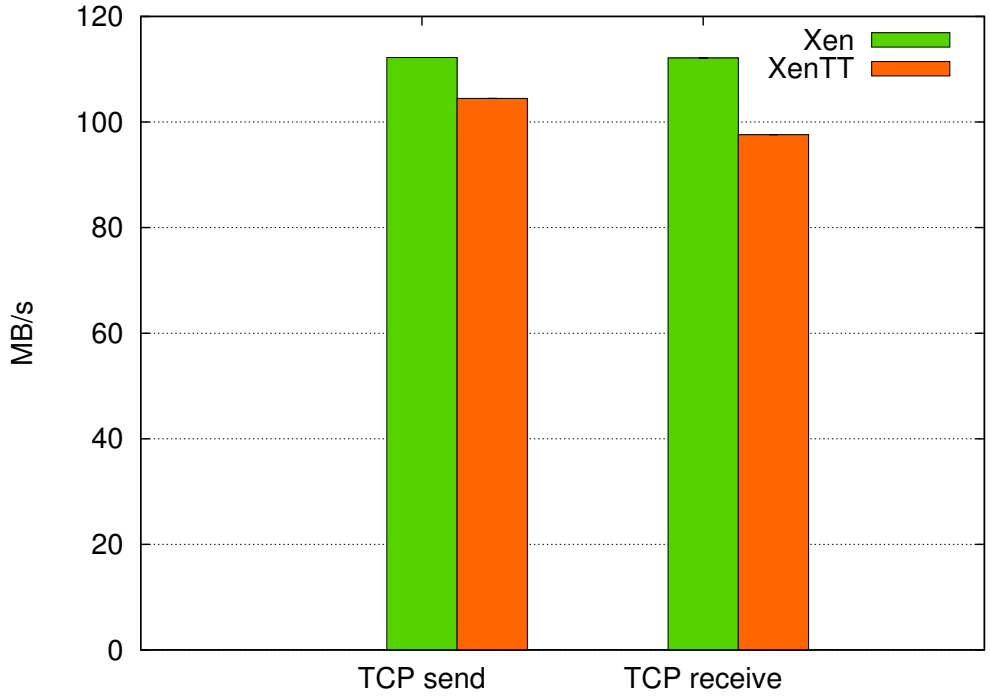

Figure 9.4. Network throughput.

workload at the speed which matches the speed of the original unrecorded system.

In a more realistic scenario, a cloud system assigns some fraction of its nodes to run each stage of the LAMP stack. The increase in the size of the system which is required to match the performance of the original unrecorded system is computed according to the following equation:

$$\textit{Increase in size} = \sum_{\textit{Workloads}} \textit{Fraction of nodes}/\textit{Performance under recording} \qquad (9.1)$$

If we assume that each stage of the LAMP stack is assigned an equal fraction of nodes; for example, one third of servers is allocated to run each of the three LAMP stages: Apache, PostgreSQL, and PHP, then, according to Equation 9.1 the slowdown introduced by the deterministic recording layer will translate into the 25% increase of the number of servers (the performance under recording numbers for the Apache, pgbench, and phpbench are taken from Figure 9.2).

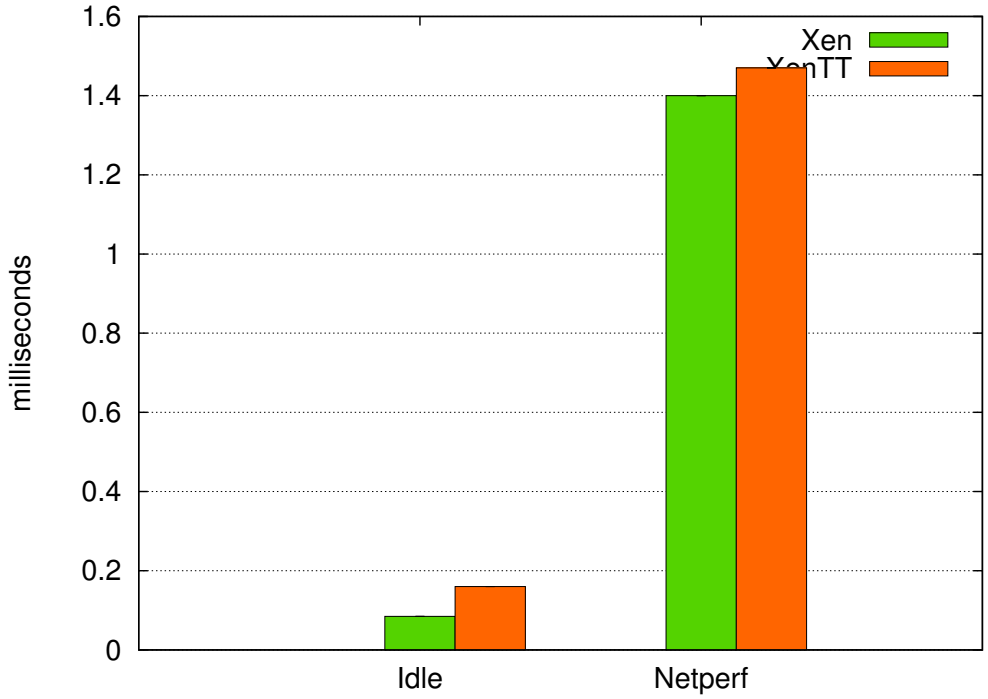

Figure 9.5. Network delay.

For the systems which run CPU-intensive workloads like media encoding, which is represented by the `lame` benchmark in Figure 9.2, or data encryption, which is represented by the `gnupg` benchmark in Figure 9.2, and `openssl` benchmark in Figure 9.1, XenTT introduces only a minimal overhead of 2%.

Modern large-scale cloud systems are designed to be elastic, i.e., to scale proportionally to the load. These systems can translate the slowdown of its components into a proportional increase in their size without architectural changes. A larger size, and thus a higher operational cost, can be potentially justified by the saving opportunities, which are provided by deterministic analysis tools. Thorough dynamic analysis enabled by deterministic replay can significantly reduce the costs of debugging complex software errors, costs of analyzing subtle performance anomalies, and the price of investigating effects of security compromises.

Similar to the Phoronix system benchmarks, disk (Figure 9.3) and network (Figure 9.4) tests demonstrate that XenTT enables practical deterministic recording of a guest system

on the I/O-intensive workloads. The slowdown of 20% for the disk and less than 15% for the network devices implies that although a production system might require a proportional increase in cost, it is unlikely to require any architectural changes in order to match performance of an unrecorded system. This higher cost can be justified by the potential to significantly reduce the development cycle of a complex, multilayer storage and networking stack through application of novel automated dynamic analysis techniques enabled by deterministic replay.

The overheads of XenTT on both CPU and I/O workloads are not high enough to affect the behavior of the recorded system. While introducing the 20% slowdown, deterministic replay mechanisms preserve the realism of the analysis.

Finally, additional costs of recording the guest system are hidden in the logging component itself. These overheads are discussed in Section 7.4.5. In summary, on most workloads, to keep up with the recorded system, the logging daemon requires a dedicated physical CPU. Depending on the workload, one dedicated CPU will be able to provide enough computational power to ensure full-speed recording of 1-4 guest systems. Similarly, to ensure full-speed recording on network intensive workloads, the recording layer requires one dedicated hard drive per network interface. Here we assume the hard disks with the serial write speeds of 130MB/s, and 1Gbps network interfaces.

9.3 Log size and growth speed

To evaluate the space required to store deterministic logs, XenTT is configured to record several representative tasks (Table 9.1). Boot of the Linux kernel incurs a large number of exits to hypervisor (XenTT log the performance information on every exit) and nondeterministic events. An idle XenTT system generates a raw log with a speed of 167MB/hr (4GB/day), or 44 MB/hr (1GB/day) if compressed with gzip. For the TCP network receive test, the sizes of both nondeterministic event log and payload log are reported. The test does not report compressed size of the payload log, since it is payload dependent.

The upper limits on the applicability of deterministic replay in a production environment are determined by the workloads, which result in a high speed of log generation.

Table 9.1 suggests that the system, which only reads and writes its disk at the highest speed, generates the compressed log at the speed of 5.5 GB/hr and 7.2 GB/hr, respectively. This implies that a 1 TB hard disk can only store 5.7 and 7.5 days of recording. The system, which sends and receives data over the network at the highest speed, will generate the compressed log at the speed of 18 GB/hr and 30 GB/hr, respectively. At such high rates, a 1 TB hard disk can only store 2.3 and 1.3 days of recording. The best case for deterministic replay is an idle system, which generates the compressed log at the speed of 1 GB/day. A 1 TB disk will store 3 months of deterministic recording.

9.4 Precision of performance model

To characterize the precision of XenTT's performance model – the performance counters, which xentt makes available via the VMI interface, the performance model is configured to measure several simple instruction sequences. Table 9.2 lists the results. The "nopX" tests consist of X `nop` instructions; the "realX" tests consist of X simple instructions (i.e., `pushl`, `inc`, `addl`, etc). To improve fidelity of the measurement, the possibility of page faults while loading the test code pages is removed. In the table, the "Native" cycle counts are obtained by directly instrumenting the test using the `rdtsc` instruction directly; the "VMI/Perf Model" values are obtained from the XenTT performance model via VMI at the beginning and end of the test sequence. The "Error" values are the ratio of VMI to Native cycle counts.

The performance model is noticeably less precise for very small instruction sequences; however, this is to be expected given the fact that a combined entry and exit path to the hypervisor costs approximately 800 cycles, and the performance model must account for these and other expensive events as it virtualizes the TSC. However, its precision becomes much better over even a relatively small number of cycles (i.e., 10K sequences of `nop` or real instructions). In conclusion, the performance model provides reasonable precision for development of accurate performance analysis algorithms.

Table 9.1. Log size for various workloads.

Activity	Raw / Compressed Log Size	
Linux boot	903 MB	145 MB
Idle machine (12 hours)	2 GB	529 MB
	(167 MB/hr)	(44 MB/hr)
TCP receive (4 GB)		
Event log	1.8 GB	342 MB
Payload log	4.4 GB	(dependent)
TCP send (4 GB)	1.1 GB	211 MB
Disk write (4 GB file)	600 MB	145 MB
Disk read (4 GB file)	414 MB	62 MB

Table 9.2. Precision of the performance model.

Test	Native	VMI/Perf Model	Error
nop1	120	212	1.77x
nop10	24	100	4.17x
nop100	68	128	1.88x
nop1K	352	420	1.19x
nop10K	3,568	3,808	1.07x
nop100K	34,924	35,976	1.03x
nop1M	349,068	350,300	1.00x
real10	28	116	4.14x
real100	80	168	2.10x
real1K	516	632	1.22x
real10K	5,224	5,260	1.01x
real100K	51,700	51,888	1.00x
real1M	516,756	517,680	1.00x

CHAPTER 10

CASE STUDIES

This dissertation makes an argument that the replay analysis paradigm should become a de facto part of the engineering cycle of any complex software system. The case studies in this section are aimed to demonstrate the power and flexibility of this new paradigm, and ultimately convince the reader that it is a viable and practical way to understand and analyze behavior of complex software. This section demonstrates two applications of the deterministic analysis approach: (1) a cross-layer Linux kernel performance analysis, and (2) backtracking of a multistage security exploit.

The first case study demonstrates the feasibility to use XenTT for performance analysis of a complex software system – a network attached storage stack implemented with the Linux kernel. The second case study demonstrates the power of deterministic replay for performing a backward analysis of the system. Multiple replay runs are used to implement backtracking of a security exploit, i.e., discovering the origin and path of a multistage security attack – a difficult question to answer conclusively based on a single execution or examination of postattack system state.

10.1 Performance analysis of the network attached storage

One of the original motivating examples for developing the technique of replay analysis was the author's experience of measuring, tuning, and analyzing the performance of a Linux-based network-attached storage stack [30]. Network-attached storage (NAS) is a representative example of a complex multicomponent system [68, 129, 94]. The performance of such system depends on the coordinated operation of a number of highly tuned kernel components and several physical devices. The complexity of cross-component interactions provides an example to motivate the approach taken by this dissertation.

This study analyzes the incoming NFS request processing path within the Linux kernel using replay, and contrasts this effort with prior experience without replay.

10.1.1 The NFS request path

Before reaching the physical disk, every file system request flows through several layers of storage and network stacks on the server machine (Figure 10.1, write request path is shown with a grey line). The request goes through the layers of the network device driver, network protocol layers, Sun RPC, NFS server, local filesystem, buffer cache, disk device drivers, and finally physical disks.

To minimize the impacts of I/O delays, most layers communicate via asynchronous queues. Most parts of the request-processing path are designed to optimize data movement with zero-copy communication. However, some interfaces require a data copy due to design restrictions. The delays of individual requests and overall request throughput of the system critically depend on the availability of space in the communication queues to buffer requests between stages, the speed of data movement between devices, memory and caches of individual CPUs, the delays of cross-layer communication and synchronization primitives, and efficiency of stage scheduling.

Under a full load, the system pipelines request processing, reducing a per-request execution time from a full path to the length of the longest pipeline stage and overheads of cross-stage communication. The performance of the pipeline is measured by the number of requests processed per period of time and depends on the ability of a system to fully load individual stages.

A simple error in the configuration of individual component can severely impact performance of the entire system. A stall of a single stage blocks subsequent stages and eventually the entire processing path.

10.1.2 Prior analysis without replay

Over the course of the work on optimizing performance of the network attached storage [30], the author of this dissertation detected and eliminated a number of performance bottlenecks in different processing stages. A large bandwidth-delay product and the small

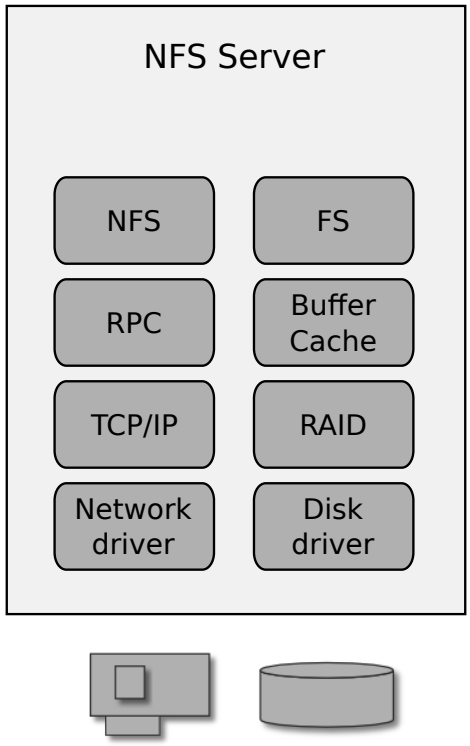

Figure 10.1. Request processing path of a network-attached storage system.

size of the network buffers were reasons preventing the network link from reaching its theoretical bandwidth.

An insufficient number of NFS server threads was responsible for blocking the NFS client after it quickly fills the window of in-flight NFS requests. A lack of a buffer space was preventing the RAID5 algorithm from collecting a complete stripe on write and resulted in an additional read-before-write request. The misconfigured Linux `pdflush` daemon[1] was limiting the performance of this system by failing to provide enough requests to the NFS client layer. Synchronization errors in the network device drivers were causing transient lockups of the network interface card.

A thorough dynamic analysis enabled the author to improve performance of a system by a factor of four, reaching a performance limit determined by the speed of a network link between client and server machines [30]. Based on traditional performance monitoring

[1] `pdflush` promotes writes of filesystem data from the client's buffer cache to the disk.

tools, the analysis required many hours of work, numerous benchmark runs, and thorough understanding of the low-level behavior of the system. To generate a global view on the system's performance, the author collected a number of performance metrics at different layers of a system. This required several network, filesystem, and block-level benchmarks and a number of performance-monitoring tools (listed in parenthesis next to every part of the NFS system: network sockets, buffer caches (`procfs`), TCP/IP protocol, NFS, and RPC (`tcpdump`, `ethereal`), block device level (`iostat`), memory management (`slabtop`), and the system overall (OProfile). To recreate a request-processing path, the author manually analyzed the source code of the system. Some corner cases of the system's behavior involving asynchronous execution on soft pagefaults required a separate run-time clarification with OProfile.

Lacking proper analysis tools, the author failed to quantify the performance overheads of cross-stage communication, and data movement between layers of the network attached storage stack, as it requires source-code instrumentation of a system. The author also failed to provide an explanation for some performance anomalies related to lock contention between the ext3 file system and the Linux md5 software RAID.

10.1.3 Analysis algorithm

The goal of this analysis algorithm is to explain behavior of the system on the NSF write path. First, the analysis dynamically verifies that the actual run-time behavior of the system corresponds to a typical hight-level understanding of the Linux network and storage stack, derived from the manual source code examination. Second, the analysis aims to identify the stages of the processing pipeline which take the most time. The goal of the analysis algorithm is to measure how much time each individual request spends at each stage of the processing path of the network attached storage.

Two steps are required for achieving the above goals: monitor the system on the request processing path, and track processing of individual requests.

10.1.3.1 Identifying the request processing path

To identify all the functions involved in the processing of network and disk requests, a quick source code analysis of the Linux kernel was performed. This manual source analysis

is augmented with the information from the BTS traces, which are collected iteratively over multiple replay runs (see Section 7.5.2 for a detailed explanation of the BTS interface).

The bottom part of Figure 10.2 shows the request processing path for the NFS write request. A request goes through seven components of the Linux kernel: frontend network device driver, TCP/IP stack, NFS Sun RPC, NFS, VFS, and file system write interfaces, buffer cache, pdflush daemon, and finally the block front device driver.

Using XenTT VMI, the analysis algorithm insert probes on and in 17 functions on the path (Table 10.1). When the request processing path is known, the analysis uses VMI probe mechanism to monitor individual requests.

10.1.3.2 Keeping track of individual requests

To measure processing times for individual requests, the analysis tracks each request through the probed functions. This task requires tracking identity of each request across the stages. At every processing stage, the analysis algorithm uses a memory address of a data structure, which describes the request as a unique identifier for the request. In a single address space of the Linux kernel, each data structure has a unique address.

Figure 10.3 describes the data structures used by the analysis algorithm to keep track of individual requests, and processing times at every stage of the processing path. Every time a request moves between components of the Linux kernel, e.g., from the TCP/IP stack to the Sun RPC layer, the analysis changes the *identity* of request, e.g., from the address of the `sk_buf` to the address of the `svc_rqst` – the two data structures which represent it at two adjacent layers of the network or storage stack.

The analysis algorithm keeps a hash of key-value pairs, which allows it to look up a particular request by the address of its kernel data structure. When request is moved between stages, the algorithm removes from the hash the address of the data structure from the previous stage, and adds the address of the data structure which describes the request at the new stage. The analysis algorithm adds a new stage to the request's description. To keep track of time spent at each stage, the analysis algorithm relies on the VMI interface and the XenTT performance model.

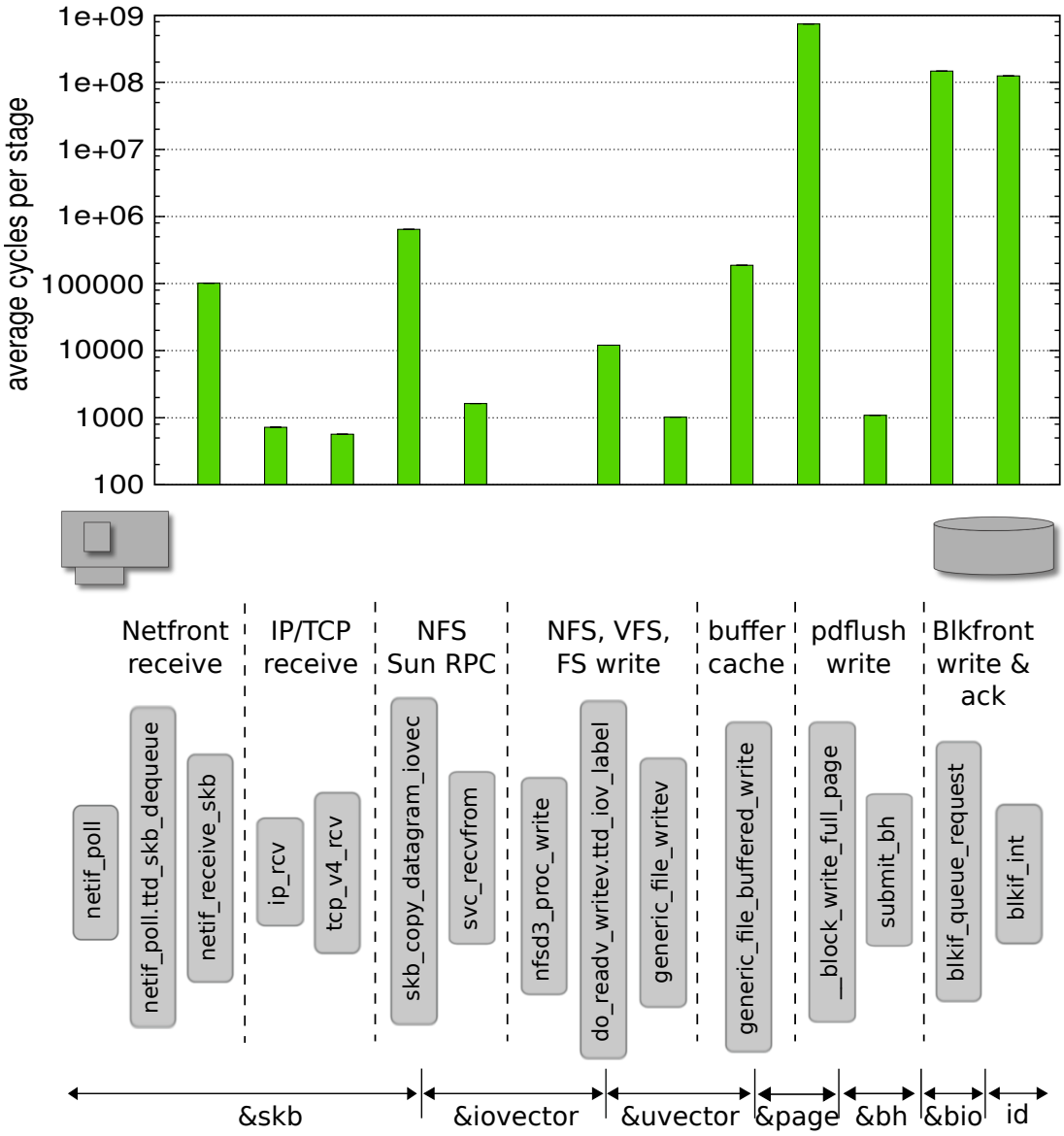

Figure 10.2. NFS request processing, bottom to top: data structure types that identify the request; probed functions; major path sections; graph of average cycles spent in each layer.

10.1.4 Performance on the request processing path

The top of Figure 10.2 shows the time which requests spend on average at each stage of the NFS request pipeline. The NFS experiment is configured with two d710 Emulab nodes connected through a 1Gbps network connection. The client uses the `dd` disk copying tool to write a 1GB file in an NFS mounted file system. The NFS mount is exported by a guest virtual machine, the execution of which is recorded with XenTT. The XenTT node is

Table 10.1. VMI probe points required to monitor the processing path of the NFS write requests.

Probe point
netif_poll
netif_poll.ttd_skb_dequeue
netif_receive_skb
ip_rcv
tcp_v4_rcv
skb_copy_datagram_iovec
svc_recvfrom
nfsd3_proc_write
do_readv_writev.ttd_iov_label
generic_file_writev
generic_file_buffered_write
__block_write_full_page
submit_bh
blkif_queue_request
blkif_int

configured identical to the configuration described in Chapter 9.

The `dd` application achieves the write speed of around 69MB/s. Requests spend the most time – up to 309 ms (742M cycles) in the buffer cache (the buffer cache stage ends with the invocation of the `__block_write_full_page` function). Getting through the elevator scheduler (the stage ending with the `blkif_queue_request` function) can either take no time at all or can take hundreds millions of cycles. On average, the analysis measures the time spent in the elevator scheduler to be 147M cycles, or 61ms. The round trip from the frontend device to the backend device and back averages at 124M cycles, or 51ms (the stage ending with the `blkif_int`). The time required to perform a receive operation by the IP layer (`netif_receive_skb`) takes either no time, or up to 301K cycles (the average is 101K cycles or 42 μs). Such variability is explained by the position of the request in the receive queue. The `skb_copy_datagram_iovec` copy operation which copies the packet data from the `sk_buf` to the `iovec` data structure may take from 100K-3M cycles. It averages at 643K cycles, or 267 μs.

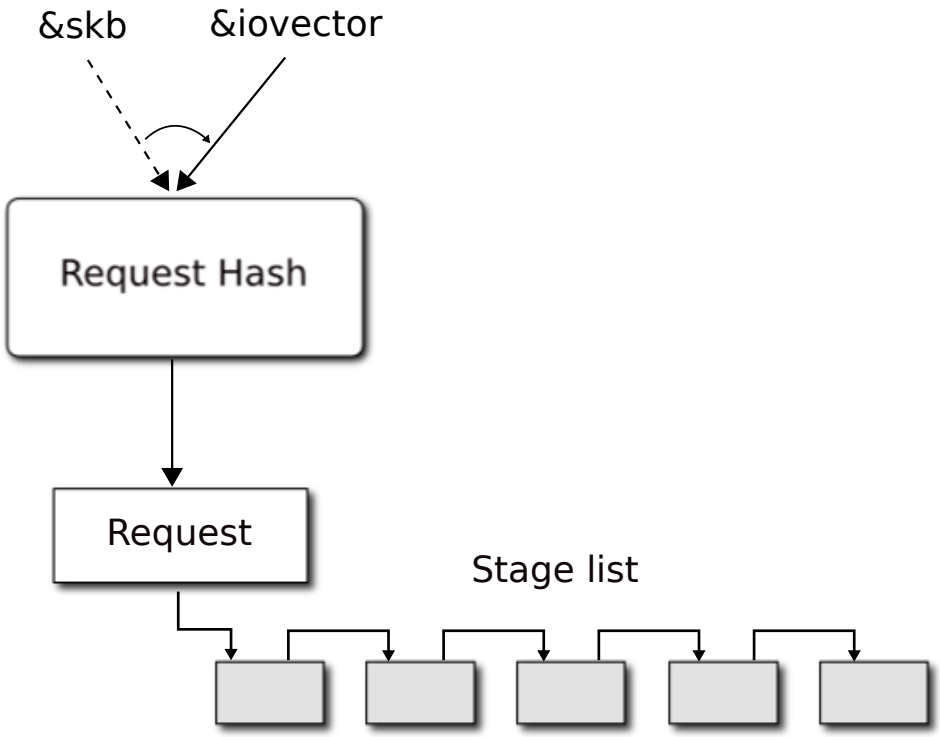

Figure 10.3. Datastructures describing requests at every stage of the processing path.

10.1.5 Discussion

The replay-based performance analysis is an enjoyable and thrilling experience. Several aspects of this approach were important for changing the way we analyze complex software systems today. First, deterministic replay provides the ability to look at the movement of individual requests along the processing path, and observe the internals of several rather complex subsystems of the Linux kernel: network stack, file system, and disk. The ability to re-replay the system and verify that the static model of the system's behavior corresponds to the actual run-time behavior for processing of particularly "slow" requests is a quick and efficient way to understand dynamic properties of a complex system.

Second, the development of the NFS performance analysis was fast (3 days), and convenient. The NFS path tracking code is a general mechanism for understanding request processing paths in various subsystems of the Linux kernel. Adding a new path, e.g., an NFS read request path, should take less than a day. The fact that the probe programming, request tracking, and performance measurement collection is done from the outside of the

complex system is a major enabler for simplifying development of rather complex analysis.

Third, determinism of replay enabled the author of this dissertation to develop the analysis algorithm iteratively. The author did not guess the request processing path correctly from the first time. However, changing the analysis code and running the replay is much faster and easier than re-running the entire NFS file copy experiment.

10.1.6 Relation to existing work

Periodic program sampling is a standard technique for execution profiling and performance analysis [81]. To identify how execution time is spent across parts of a program, the program is periodically preempted to sample the value of the instruction pointer register. The period of time from the last sampling is attributed to the part of a program addressed by the current value of the instruction pointer. Over a long run, this approach creates a statistically correct profile of a running program. Trying to preserve some information about the control flow inside the application, recent profilers provide the capability to inspect and save the program's stack trace on every sample [114, 81]. The accuracy of a profile information depends on the length of execution and period of sampling. To ensure fine-grain sampling intervals, recent profilers rely on hardware performance counters [114]. Found in most modern CPUs, hardware performance counters provide a way to preempt execution after the CPU runs through a defined number of certain low-level architectural events [51, 49], e.g., CPU cycles, taken or mispredicted branches, cache misses, and TLB misses. Performance counters provide flexibility to profile an application with respect to different hardware metrics.

Trying to bridge the semantic gap between analysis and execution, modern profilers provide support for injecting lightweight analysis probes straight into the binary image of a running system [31, 144]. Binary instrumentation is used to insert and remove probes at run time, thus enabling profiling of long-running systems without interrupting their execution. Run-time probes enable iterative profiling, i.e., the ability to add more probes and collect specific data after the initial pass of analysis. Targeting production systems, profilers support a safe domain-specific scripting language for implementing probe code [31, 144]. Language safety mechanisms are used to prevent analysis code from accidentally destroy-

ing the state of a system.

Causal path analysis is an effective way to reason about the behavior of complex multi-component systems [3, 14, 35, 146]. In a request-driven system, causal paths represent information about delays and resources consumed by individual requests. Path-based analyses extend the system with logging primitives, which collect enough information to reconstruct the request path [14, 35, 146]. Pinpoint [35] and Pip [146] track requests by extending them with a unique identifier. Magpie eliminates the need for a unique request identifier by reconstructing individual request paths through a concept of temporal join, and application-specific path schema [14]. Aguilera et al. developed statistical methods to infer program paths in a distributed system from passive observations of network traffic [3].

Multiple teams approach the analysis of complex systems by studying their performance on a critical path [188, 70, 13, 150]. Fields et al. developed a dependence graph model for an out-of-order CPU pipeline [70]. Barford and Crovella developed a critical-path model for TCP [13]. Yang et al. provided a methodology to construct a critical-path graph for MPI applications [188]. Saidi et al. provided a method for automatic generation of a dependence graph for the Linux kernel based on implicit state machines [150].

10.2 Backtracking kernel exploits

The power of the replay analysis is in its ability to provide a complete explanation for the abnormal behavior in a seemingly normal code. Both traditional debugging and analysis of adversary attacks can benefit from automatic means of explaining execution of the system. As the software systems grow in complexity with every generation, it becomes harder and harder to explain the run-time behavior of a complex multilayer software, e.g., understand how a system gets into a particular state. The level of complexity and sophistication of the exploit scenarios is also growing rapidly. A nearly common deployment of protection mechanisms: stack guards, address space randomization, sandboxing makes it quite common to see multistate, and multiexploit attacks, which deploy techniques to bypass these protection mechanisms.

Automatic explanation of execution is a promising research area in which deterministic analysis can fully leverage the power of deterministic replay.

Deterministic replay provides an analysis mechanism with the ability to move its logic backwards in the execution history. It is possible to apply the principles of backward slicing [170] to derive a set of actions in the past, which cause the system to get into an abnormal state int the present. Determinism of replay provides the analysis algorithm with the ability to ask a question: when and how a particular variable, which affects the state of the system now, was changed in the past. To answer this question, an analysis algorithm relies on a new replay run. During this run, analysis observes the accesses to a specific variable.

The number of edges in the backward slice of the anomalous execution is small compared to the number of all possible variables affecting the state of the system. This is the key observation which enables deterministic analysis to scale and explain execution histories of multistage debugging anomalies, and attacks.

The analysis example presented in this section implements the first step towards the vision of automatic explanation of execution. XenTT implements a subset of routines required to generate and verify hypotheses about the causes of a faulty behavior. The analysis moves backwards in the execution history along the edges generated by these hypotheses

10.2.1 Backtracking

A general backtracking algorithm consists of the following steps:

- **Hypothesis generation:** Generate the hypothesis for the possible cause, which forces the system to get into a bad state.

  In a typical scenario, there are two possibilities for how the system gets into a certain state: data or control dependency. A data dependency forces the system to take a certain control flow path, e.g., invoke a certain function pointer, or take a certain branch of the control flow construct. A control flow dependency allows the algorithm to move backwards across multiple function invocation to answer the question of how a certain flow of execution was created. Every time the analysis moves backwards along the control-flow edge, it checks the integrity of the control flow along this edge, to see if the flow of control was hijacked.

For the control dependencies, XenTT uses the context tracking, trace recording, and control flow integrity libraries to identify and verify integrity of the control flow.

In a fully automatic backtracking algorithm, the data dependencies are generated with a program slicing algorithm [170]. Without a working implementation of a backward slicing algorithm, XenTT relies on the human to generate a hypothesis manually by looking at the trace of execution, and at the source, and possibly, assembly code of the system.

- **Hypothesis exploration:** Start a replay session to run an analysis aimed to verify if hypothesis is correct, and if so, search for the point in the execution history, at which a critical transition happens – a data, or a control flow update, which causes the system to choose an execution path leading to a fault.

  This step of the backtracking analysis relies on an analysis algorithm, which is able to verify the hypothesis. The analysis algorithm has to check if the hypothesis is true, e.g., the control flow is violated, and identify the position in the execution history where the critical transition happens.

- **Choosing new state:** If the hypothesis is correct, the backtracking algorithm moves backwards in the execution history of the system choosing a new state, from which it continues to the hypothesis generation step.

10.2.2 Auxiliary analysis mechanisms

Several helping mechanisms provide a general framework for developing any backtracking analysis algorithm. Context tracking analysis is responsible for tracking the execution flow of a particular thread while it can transition across kernel, user, and interrupt contexts. The control flow integrity (CFI) analysis is a general mechanism to answer the question whether the control flow of the system is compromised at a particular interval of its execution.

10.2.2.1 Context tracking

Many analysis algorithms include the knowledge of an execution context in which the system is running at a particular point of its execution, in their logic, e.g., in kernel,

executing a system call, in the interrupt handler, etc. To simplify development of such analyses, and provide a general analysis interface, the XenTT VMI library implements a context tracking analysis.

The context tracking analysis monitors how the execution within the guest system moves between context, e.g., user to kernel, kernel to interrupt, interrupt to interrupt, etc. At every moment of execution, the context tracking analysis can inform the calling analysis about the current context.

The context tracking analysis is often used as a context filter. For example, if an analysis wants to record the trace of the execution of a specific function, e.g., verify the control flow integrity of a system call, it can use the context tracking analysis to filter out all the interfering contexts, which preempt the execution of the target context. Another potential use of the context tracking analysis is detection of hidden execution inside the guest system.

Table 10.2 lists the probe points, which are used in the guest Linux kernel to track changes of the context. Four groups of probes track transitions between: traps, IRQs, system calls, and process switches.

10.2.2.2 Control flow integrity

Similar to the context tracking analysis, control flow integrity (CFI) analysis is another component of the backward reasoning framework. CFI verifies that control flow of the execution corresponds to some model of an acceptable control flow transitions. In other words, CFI allows the analysis algorithm to identify the point at which the control of the system was hijacked either by a malicious exploit, or as a result of a simple null pointer dereference error.

XenTT implements verification of a simple CFI model: *all returns should match their call cites*. This model is sufficient for the kernel code, which uses the simple execution model, and in which nearly all calls return during the normal kernel execution.

In general, deterministic replay provides two ways to implement the CFI analysis. It is possible to record a trace of all call/return (or in a more general case, all control flow instructions) and build an analysis reasoning about such a trace. In this system, it is possible to use the branch tracing store to record a trace of all taken branches. Alternatively, it

Table 10.2. VMI probe points required to monitor context switches inside the guest Linux kernel.

Probe point
Traps
do_divide_error
do_debug
do_nmi
do_int3
do_overflow
do_bounds
do_invalid_op
do_coprocessor_segment_overrun
do_invalid_TSS
do_segment_not_present
do_stack_segment
do_general_protection
do_page_fault
do_coprocessor_error
do_alignment_check
do_simd_coprocessor_error
IRQs
do_IRQ
System calls
system_call
system_call.return
Process switches
schedule.switch_tasks

is possible to implement a dynamic CFI analysis by instrumenting the guest system at run-time and observing the control flow through the VMI probe mechanism.

XenTT implements the dynamic CFI analysis. Figure 10.4 presents a high-level logic behind the CFI analysis algorithm. The main challenge for a dynamic CFI algorithm is to remain in control of the execution flow within the guest system. In some way, the dynamic CFI algorithm is similar to a binary translation engine used in the traditional virtual machine monitors.

The analysis starts the CFI algorithm by pointing it to an entry point of a function, from which it starts tracking the control flow. The CFI algorithm tries to use a static disassembler

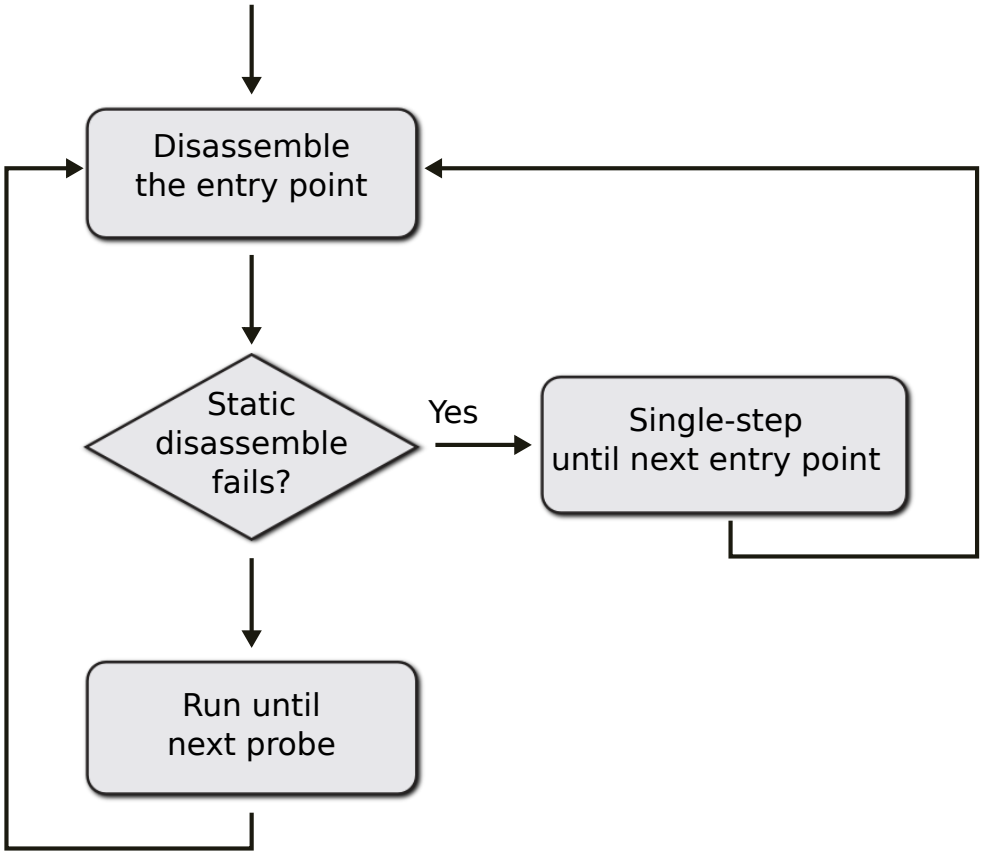

Figure 10.4. A flow-chart diagram for the CFI algorithm.

to identify the `call` instructions inside the function body. If the static disassembling succeeds, the CFI algorithm instruments all the call sites inside the function, and lets the system run until one of the call instructions gets executed and control returns back to the CFI algorithm.

If the static disassembling fails, the CFI algorithm has no other option but to single-step the system until it reaches the next call site, at which point the CFI algorithm will try to switch back to the static disassembling.

10.2.3 Background of the `linux-sendpage` exploit

The `linux-sendpage` attack exploits the CVE-2009-2692 [136] vulnerability to effect a local root privilege escalation on vulnerable Linux kernels (2.6.0–2.6.30.4; 2.4.4–2.4.37.4). A failure to correctly initialize the protocol operations structure for the Bluetooth network driver makes the kernel vulnerable to "executing" a `NULL` pointer. The exploit

code calls `mmaps` a writeable, executable memory page at address `0x0`, and writes to it code that sets the UID field in the kernel's data structure for the user process to 0 (root permission). It then calls the `sendfile` syscall on a Bluetooth socket, which eventually executes the `sendpage` field (incorrectly set to `NULL`) in the Bluetooth driver – resulting in a jump to the user-injected code. The analysis records a virtual machine executing the exploit, and writing the password file – the observed symptom used to seed this analysis.

10.2.4 Exploit analysis

The method presented in this section uses replay to validate a hypothesis that some event *E* occurred. Once a hypothesis is validated, it may imply a new set of hypotheses that are possible causes of *E*, which is tested with another replay. With a knowledge of kernel-level events and privileges and VMI utilities that validate hypotheses, the analysis algorithm assembles a causal attack chain.

The analysis consists of five replays (passes). In Pass 1, the analysis hypothesizes that the password file was opened because a process wrote to it (other hypotheses include raw disk writes or offline file modification). Using VMI, the analysis inserts a probe on the `sys_open` system call that watches for write-mode opens of the password file, and returns the process hierarchy.

Pass 2 hypothesizes that a process in the hierarchy detected in Pass 1 escalated its privilege to root. It uses a watchpoint to detect writes to the `UID` field in a process's control block in the kernel; when found, it returns the process which escalated its privileges.

Pass 3 hypothesizes that the privilege escalation observed in Pass 2 occurred while the kernel was executing, and tries to determine the final system call executed prior to escalation. In this analysis, it finds the call to `sys_sendfile`.

Pass 4 hypothesizes that the escalation was due to a kernel control flow violated (other hypotheses are certainly possible), and performs a dynamic control flow integrity (CFI) analysis starting from the location discovered in Pass 3, using VMI to probe CALL and RET instructions, and ensuring correspondence and that only known kernel code runs. It detects a violation when the kernel jumps to address `0x0`. Figure 10.5 shows the stack trace and final jump.

```
sys_sendfile called (brctr = 401828811)
    do_sendfile called (ip=0xc015c060)
        fget_light called (0xc015d470)
        fget_light returned (0xc015d4b4)
        rw_verify_area called (0xc015bfb0)
        rw_verify_area returned (0xc015c017)
        fget_light called (0xc015d470)
        fget_light returned (0xc015d4b4)
        rw_verify_area called (0xc015bfb0)
        rw_verify_area returned (0xc015c017)
        shmem_file_sendfile called (0xc0156bb0)
            do_shmem_file_read called (0xc01569c0)
                shmem_getpage called (0xc01561a0)
                    find_lock_page called (0xc013a360)
                      radix_tree_lookup called (0xc022c900)
                      radix_tree_lookup returned (0xc022c928)
                    find_lock_page returned (0xc013a3fc)
                    shmem_recalc_inode called (0xc0154f40)
                    shmem_recalc_inode returned (0xc0154f83)
                    shmem_swp_alloc called (0xc0155fd0)
                      shmem_swp_entry called (0xc0155050)
                        kmap_atomic called (0xc01142b0)
                          __kmap_atomic called (0xc0114090)
                            page_address called (0xc0145250)
                            page_address returned (0xc014529d)
                          __kmap_atomic returned (0xc0114252)
                        kmap_atomic returned (0xc01142d4)
                      shmem_swp_entry returned (0xc0155103)
                    shmem_swp_alloc returned (0xc015613b)
                    kunmap_atomic called (0xc0114330)
                    kunmap_atomic returned (0xc0114371)
                    find_get_page called (0xc013a230)
                      radix_tree_lookup called (0xc022c900)
                      radix_tree_lookup returned (0xc022c928)
                    find_get_page returned (0xc013a27c)
                shmem_getpage returned (0xc01564c9)
                file_send_actor called (0xc0139b50)
                    sock_sendpage called (0xc02702d0)
                      UNKNOWN FUNCTION (0x00000000) CALLED
                          (BRCTR = 401829543)
```

Figure 10.5. Pass 4 CFI analysis of sendpage exploit.

Finally, Pass 5 hypothesizes that a user process wrote code to the address Pass 4 discovered; it uses VMI to probe the page fault handler to determine if and when this address was written, and in what calling context – i.e., a kernel thread or user process; what instruction attempted the write.

10.2.5 Discussion

This analysis obtains low-level, conclusive attack information. Although some of the stages of this analysis rely on functional knowledge of the Linux kernel (process control block structure, syscalls), other stages are more generic (the CFI analysis). It is impossible to manually implement stages that cover a complete set of kernel functionality – but it is possible to implement and model enough, that, when combined with generic stages (such as CFI, memory analyses, etc.) – the analysis can backtrack many different kinds of intrusions.

As part of the future work, the author plans to automate backtracking – a hypothesis engine will build a forest of analyses to iteratively backtrack a system event and determine its causal chain. Finally, since a full recording of the system's execution is available for the analysis algorithm, it would also be possible to analyze an attack whose stages were widely separated in time – i.e., an initial exploit that created a fake account, but never used it until much later.

CHAPTER 11

CONCLUSIONS

This dissertation presents a deterministic replay framework and analysis environment designed to create a foundation for the replay-based analysis of complex software systems. Several goals guided design of this project: (1) support of the real systems used in industry and academia on a daily basis; (2) support for realistic workloads; (3) minimal path for starting analysis of an already running system; (4) analysis of the entire software stack. The goal of this work is to show that deterministic replay environments can address the challenges of being practically useful, and thus suitable as "standard equipment" in deployed, production systems. To this end, this dissertation describes the design principles behind the deterministic replay engine implemented for a full-featured, industry standard virtual machine monitor, shows that it is possible to provide logging and replay with an acceptable performance overhead, and describes a development path for the interfaces that support functional and performance analyses over real systems. The ubiquitous availability of whole-system deterministic replay can change the ways in which people approach systems analysis – change for the better. This dissertation shows that such replay platforms are practically feasible and demonstrates useful techniques for their implementation.

11.1 Design choices and lessons learned

The work on this dissertation required four man-years. Three man-years were spent designing and implementing the logging and replay layer for the industry-standard, full-feature virtual machine monitor Xen. One man-year was spent developing the virtual machine introspection mechanisms, and examples of deterministic analysis. This section discusses several design choices which were critical for achieving the dissertation's original goals.

11.1.1 Paravirtualization and deterministic replay

The engineering challenge of developing deterministic replay lies in the complexity of designing and implementing a complete logging and replay interposition boundary. This is trivial in case nondeterminism is restricted to a high-level messaging interface. Complexity of practical replay tools comes from a many-hour analysis required to design a replay interface for a system with a highly-optimized low-level interface involving communication via shared memory, access to low-level CPU and I/O interfaces, and asynchronous signals.

Implementations of deterministic replay for both fully virtualized and paravirtualized virtual machines seek to minimize their complexity by capturing nondeterminism at the minimal interposition boundary. A replay engine for a fully virtualized hypervisor is typically designed to capture nondeterminism from nondeterministic CPU instructions (`rdtsc`), asynchronous interrupts, nondeterministic in/out instructions, accesses to the memory mapped I/O regions (MMIO), and asynchronous DMA accesses, which transfer data straight into memory of the guest system [37]. Similar, the implementation of the deterministic replay engine for a paravirtualized virtual machine monitor records nondeterminism from nondeterministic CPU instructions, paravirtualized interrupt handlers, a set of shared memory pages which emulate hardware registers of the CPU, hypervisor call interface, and shared memory I/O interface of the paravirtualized device drivers.

11.1.2 Does paravirtualization simplify deterministic replay?

The problem with paravirtualization is that in the Xen architecture, paravirtualization is not aimed to develop a more abstract hardware interface. Paravirtualization in Xen serves a pragmatic goal to replace expensive emulation of the sensitive CPU instructions with explicit hypervisor calls. To make sure that paravirtualized interface is adopted by the mainline operating system kernels, paravirtualization stays close to the hardware CPU interface, and keeps the amount of changes to the guest kernel minimal.

Both full virtualization and paravirtualization result in inherently similar interposition boundaries with the similar sources of nondeterminism. Paravirtualization helps to simplify development of the deterministic replay layer, but the difference between two virtualization techniques is not critical. The main reason to choose the paravirtualized interface over the

full virtualization is the ability to remove a relatively complex software component – a hardware emulator – from the development path. The absence of the hardware emulator helps in the following ways: (1) explicit hypervisor calls simplify reasoning about determinism of interactions between the guest system and the hypervisor, i.e., hypercall results are deterministic due to determinism of the hypervisor; (2) a high-level paravirtualized device I/O interface allows implementation of a general device driver interposition framework, which uniformly works for almost any paravirtualized device.

11.1.3 Can we achieve more with better paravirtualization?

Ideally, a virtual machine interface should completely encapsulate execution of the quest system. Deterministic replay should be able to leverage this encapsulation to record only "true" nondeterminism entering the guest system from external sources of nondeterminism. In practice, a paravirtualized interface still requires some number of unnecessary exits to the hypervisor and some amount of emulation. First, paravirtualization virtualizes guest exceptions (interrupt vectors 0-15) in software. Second, privileged instructions require an exit into the hypervisor and emulation. Third, paravirtualization relies on the software virtualization of the guest's page tables (automatic translation of the physical map). All of the above events are deterministically issued by the guest system, and do not require logging. However, the cost of exits to the hypervisor alone is still high. A paravirtualized interface can avoid this overhead for both time traveling and nontime traveling systems by extending paravirtualization to support the hardware assisted virtualization of the CPU. Fortunately, this is the direction in which Xen is already moving. The Xen paravirtualized interface will be extended to rely on the hardware support for the virtualization of the CPU provided by the Intel and AMD CPUs.

11.1.3.1 Self logging and self replay

A less traditional extension to a paravirtualized interface can improve performance of a logged system on workloads which execute a large number of nonsensitive, nondeterministic CPU instructions, e.g., `rdtsc`. Execution of a nondeterministic instruction requires an exit into the hypervisor. This exit is not required for a non-time-traveling system, but is unavoidable for time-traveling guests to record the values of nondeterministic

instructions observed by the guest. The exit to the hypervisor can be avoided if the guest system is paravirtualized to support *self logging* – the ability to record observed values from nondeterministic sources.

Despite looking unusual, self logging is quite simple. The guest system can use tracing functions, similar to pluggable interposition functions used in the XenTT recording layer, and relay recorded information to a logging daemon, which runs outside of the guest, through one of traditional I/O devices, e.g., network, disk, etc. The replay daemon can reconstruct the value observed by the system during the original run, and replay it to the guest.

Replay of original values can be implemented in two ways. An obvious approach is to make the sensitive instruction to be privileged during replay, and trigger an exit into the hypervisor. Inside the hypervisor, a replay layer can replay the values from the original run. However, a more interesting approach is to implement an idea of *self replay*. Self replay extends a paravirtualized system with an idea complimentary to self logging – instead of accessing an internal source of nondeterminism, during replay, a self replaying system reads nondeterministic values, which it observed during the original run, from a deterministic "replay" device. Of course, the guest system executes two different code paths during the original run and replay. However, the path taken during the replay can be made deterministic, and of a known instruction length. A replay system can be implemented to compensate for the difference between two execution paths.

11.1.3.2 Nonparavirtualized device drivers and SMP quests

The practical advantage of self logging and self replay is the possibility to avoid expensive exits into the hypervisor on a critical path of the guest system. Several realistic use case scenarios can justify development of self logging and self replay mechanisms. First, self logging can provide an efficient way to record and replay execution of nonparavirtualized device drivers, e.g., device drivers, which communicate with the real hardware, via the I/O instructions and the DMA engine. A straightforward implementation of an interposition layer between the driver and a hardware device is to force an exception on every access to the memory and I/O space of the device, and the guest system. Frequent exceptions on

the I/O path result in a prohibitive interposition overhead. Self logging and self replay can eliminate the need for the exits by extending the device driver of the guest system to record all inputs from the hardware device.

An asynchronous communication scenario between the guest system and a physical device is inherently similar to communication across multiple CPU cores of an SMP guest. Self logging can eliminate the need to implement a fault-based CREW protocol across the cores, which is normally required to record execution of SMP guests [65]. If all memory accesses across the cores are instrumented to record values observed by each core, self logging significantly reduces overheads of recording the SMP system. Effectively, self logging implements the idea of *value determinism*. Techniques similar to presented in ReEmu [37] can be used to implement a lightweight CREW protocol. It is possible to design a protocol without lock operations on the read path, and requires only a single spin-lock per memory page on the write path. ReEmu reports significant performance improvements compared to the fault-based CREW algorithm in ReVirt.

11.1.3.3 Better abstractions for virtual machine interface

A discussion of the paravirtualization and its impact on the complexity of the replay layer remains incomplete without an attempt to answer the question of whether it is possible to create a more abstract, cleaner, and smaller interface between the guest operating system and the hypervisor. An existing virtual machine interface closely follows the structure of the hardware interface exposed by the physical processor. An advantage of this approach is the fact that the guest system requires only minimal changes to be paravirtualized. The similarity between virtual and hardware interfaces provides a possibility to dynamically switch between the real and virtual interfaces at run-time during boot. The Linux PVOPS project is an example implementation of a minimal indirection layer, which allows a kernel to boot on both real and virtual hardware.

A disadvantage of the existing virtual machine interface is its rather complex, low-level organization. The hypervisor can update the state of the guest through a combination of multiple loosely structured mechanisms: fields in the special pages shared between the guest system and the hypervisor, hypercall results, straight copy operations to the guest's

address space, invocations of the interrupt and exception handlers, and direct updates to guest's registers. Often, delivery of a single operation, e.g., an interrupt, requires updates to several parts of the guest's state. The diversity of the virtual machine interface complicates the replay engine, which needs to interpose on every possible update path, and make sure that updates are performed in a particular order. Implementation of a replay engine requires low-level extensions to the hypervisor. Although it is possible to implement deterministic replay for such interface, it is hard.

A virtual machine interface can be simplified if all interactions between the virtual machine and the hypervisor are encapsulated in a single abstraction of a message IPC. For many decades, microkernel systems demonstrate feasibility to (1) implement the messaging interface efficiently; (2) port Linux guests to run on top of the messaging IPC. Availability of a single guest update mechanism can significantly simplify the structure of the deterministic replay. Furthermore, a minimal support for IPC redirection can enable implementation of the deterministic replay outside of the hypervisor.

11.1.3.4 Better abstractions for scalable determinism

The ability to scale deterministic replay on a multiprocessor system will likely depend on a set of deep architectural changes to the traditional operating system stack. A monolithic operating system kernel, which runs on every processor of an SMP system, introduces a large amount of incidental nondeterminism to the traditional system. The in-kernel sharing unfairly penalizes majority of otherwise nearly deterministic applications. To eliminate the incidental sharing, a traditional operating system needs to be redesigned to run as a collection of services, which run in isolation on individual processors, and update the global kernel state via explicit communication protocols. Alternatively, some processors might not run the kernel at all, and rely on the hypervisor for process switching of local processes. In this scenario, traditional kernel services can be provided via cross-CPU system calls.

As was discussed above, an efficient recording of parallel applications will likely require an explicit support from communication mechanisms. In-kernel services, and common user-level libraries will require modifications to manage globally shared objects via

explicit memory update primitives, or message-based IPC. Finally, to reduce the amount of determinism to a manageable level, some applications might require execution under a deterministic scheduler.

11.1.4 Experiences with Xen

Demonstrating exceptional performance and stability Xen turned out to be a good choice for this dissertation. Like any other operating system software, the Xen virtual machine monitor requires a high price of learning the basics of its architecture. After this price is paid, Xen provides a programmable environment with clean and carefully designed interfaces.

Another advantage of choosing Xen is that XenTT and Xen share an important design principle – minimal interference with the guest system on its critical path. Xen implements this principle really well. This helps to simplify the design of the XenTT's interposition layer. Furthermore, the split device driver model turned to be a convenient and general layer for interposing on high-bandwidth device I/O.

11.1.5 Experiences with deterministic replay analyses

Deterministic replay analysis is a practical, efficient paradigm with the potential to change the traditional debugging cycle for complex software systems. Working with the XenTT's VMI libraries, the author was able to efficiently prototype complex dynamic analyses, and what is even more important, promptly answer questions about internals of the complex, multilayer, multicomponent stack of the Linux system. The XenTT's VMI interface – probes and symbol names – provides convenient programming abstractions for implementing dynamic systems analysis. VMI and BTS interfaces provide a convenient way of exploring the execution history.

11.1.6 Performance of the device interposition layer

XenTT demonstrates an excellent performance on CPU and sequential I/O workloads. Careful design choices for the Xen-level interposition layer, and selection of the minimal subset of nondeterministic events were the two most important reasons responsible for the high performance of the interposition layer.

XenTT's performance on the random I/O workloads is lower than expected. The possible reason for that is that the `Devd` device interposition layer is implemented as a user-level component. This design choice has two advantages: (1) the user-level development is simpler compared to kernel-level programming; (2) being a user-level application, `Devd` is more portable compared to an in-kernel implementation. Unfortunately, the user-level execution adds latency in terms of communication signaling, and execution scheduling. It is not clear if these issues can be addressed in the Linux operating system. It is possible that `Devd` will have to be re-implemented `Devd` as a kernel module.

11.1.7 Nondeterminism of branch counters

A large portion of the XenTT's development time was spent on fixing nondeterminism of the branch counting interface of the Intel CPUs. The lack of the accurate instruction or branch counting is upsetting – despite the fact that it is a critical component of the deterministic replay architecture, it is not the core contribution of this work. The author of this dissertation hopes that a wider spread of the deterministic replay tools will result in a reliable instruction counting interface becoming an integral part of every processor.

11.2 Implementation status and possible extensions

The current version of XenTT supports replay of single-CPU, paravirtualized, 32-bit Linux guests. The following extensions to XenTT are possible and relatively easy to implement. They will become immediate steps for enhancing the XenTT infrastructure.

11.2.1 Support for SMP guests

It is possible to extend XenTT with support for multi-CPU guests. Such extension is feasible and is relatively simple through application of existing multiprocessor replay approaches [65, 111]. Essentially, replay of the SMP guests requires implementation of a concurrent-readers-exclusive-writers (CREW) protocol [52] at the level of memory pages. Page protection mechanisms can be used to track writes to the shared pages.

11.2.2 Replay of fully-virtualized guests.

XenTT can be extended to support replay of unmodified guest systems, e.g., Windows guests, by implementing an interposition layer for the Xen HVM interface. The CPU-

level interposition for the HVM quests should not pose implementation problems. In fact, it is possible that the nonparavirtualized Xen interface is even smaller and cleaner than the paravirtualized CPU interface. The only challenge for supporting the HVM quests is efficient interposition of the emulated devices, which is implemented in Xen by the parts of the QEMU full-system emulator. XenTT's device interposition layer leverages the uniform nature of the shared ring communication to implement a general interposition architecture. It is not clear if such approach can be applied to the QEMU drivers.

11.2.3 Integration with GDB debugger

It should be simple to extend XenTT with support for attaching the GDB debugger to the instance of the guest's execution during replay. GDB was extended to support Xen virtual machine guests. These extensions can be leveraged to register GDB breakpoints with the XenTT VMI layer, and make sure that GDB does not alter determinism of replay.

11.3 Future research directions

As a practical deterministic replay and analysis framework, XenTT can serve as a foundation for several research projects. Several obvious directions, in which XenTT can evolve, are discussed below.

11.3.1 Automatic analysis of execution

XenTT is a critical part of the vision for automatic debugging, analysis, and explanation of anomalous execution. Determinism of reexecution, and availability of the complete state of the system enables implementation of automatic backward slicing algorithms, which search through the execution history of the system to reconstruct a chain of critical control flow transitions, and data dependencies, which force the system to take a faulty execution path. Such automatic backward slicing technique has a potential to become a new way of debugging software systems. The main motivation for the automatic approach to debugging is the growing complexity of the software systems.

11.3.2 Effect analysis and taint tracking

If a security attack can be explained with the backward slicing algorithm, a natural question to ask is what was the effect of the attack? To answer this question, deterministic

replay needs to be combined with the data-flow tracking engine. Practical extensions of Xen with the data-flow tracking frameworks are possible [90]; however, they need to be modified to preserve the determinism of execution during replay.

11.3.3 Symbolic execution and replay with modifications

Replay with modifications is a promising area of research, which might enable novel techniques of software testing, and model checking. Unfortunately, an attempt to modify the state of a deterministically replayed system in a meaningful way is a challenging task. The problem stems from the fact that even the smallest single-bit modification may propagate through the state of a system, and force the system to take an arbitrary control path. At this point, the state of the system becomes incoherent with the deterministic log, and further replay is impossible. The replay engine expects to find the system in a particular state to inject a nondeterministic event from the original run; however, the system is in a random unknown state. First, replay of the original log data does not make sense. Second, being in a random new state, a modified system is likely to request an input, which is not saved in the log of the original execution.

To keep track of how the changes propagate through the state of the system, and to keep the replayed execution on track of the original log, deterministic replay needs to be extended with support for symbolic execution. It is possible to execute the system with the symbolic values when concrete values are not available from the log, or from the modified environment. When concrete execution is possible, replay continues to run from the recorded log.

REFERENCES

[1] Iprobe. Digital internal tool.

[2] VTune, 2009. Intel's visual tuning environment. `http://software.intel.com/en-us/intel-vtune/`.

[3] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003), pp. 74–89.

[4] ALPERN, B., CHOI, J.-D., NGO, T., SRIDHARAN, M., AND VLISSIDES, J. M. A perturbation-free replay platform for cross-optimized multithreaded applications. In *IPDPS* (2001), p. 23.

[5] ALTEKAR, G., AND STOICA, I. Odr: output-deterministic replay for multicore debugging. In *SOSP* (2009), pp. 193–206.

[6] AMAZON. Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[7] ANDERSON, J.-A. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S.-T., SITES, R. L., VANDEVOORDE, M. T., WALDSPURGER, C. A., AND WEIHL, W. E. Continuous profiling: Where have all the cycles gone? In *SOSP* (1997), pp. 1–14.

[8] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proc. OSDI* (Oct. 2012), pp. 307–320.

[9] AUSTIN, T. M., LARSON, E., AND ERNST, D. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer 35*, 2 (2002), 59–67.

[10] BALZER, R. M. Exdams: extendable debugging and monitoring system. In *AFIPS '69 (Spring): Proceedings of the May 14-16, 1969, spring joint computer conference* (New York, NY, USA, 1969), ACM, pp. 567–580.

[11] BANDA, V. P., AND VOLZ, R. A. Architectural support for debugging and monitoring real-time software. In *Proceedings of Euromicro Workshop on Real Time* (1989), pp. 200–210.

[12] BARATTO, R. A., KIM, L. N., AND NIEH, J. Thinc: a virtual display architecture for thin-client computing. In *SOSP* (2005), pp. 277–290.

[13] BARFORD, P., AND CROVELLA, M. Critical path analysis of TCP transactions. In *SIGCOMM* (2000), pp. 127–138.

[14] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *OSDI* (2004), pp. 259–272.

[15] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP* (Bolton Landing, NY, 2003), pp. 164–177.

[16] BARTLETT, J. A nonstop kernel. In *SOSP* (New York, NY, USA, 1981), pp. 22–29.

[17] BASRAI, M., AND CHEN, P. M. Cooperative ReVirt: Adapting message logging for intrusion analysis. Tech. Rep. CSE-TR-504-04, Department of Electrical Engineering and Computer Science, University of Michigan, 2002.

[18] BELLARD, F. Qemu, a fast and portable dynamic translator. In *FREENIX* (2005), pp. 41–46.

[19] BERGAN, T., ET AL. Deterministic process groups in dOS. In *Proc. OSDI* (Oct. 2010), pp. 177–192.

[20] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *VEE* (2006), pp. 154–163.

[21] BHATIA, S., KUMAR, A., FIUCZYNSKI, M. E., AND PETERSON, L. L. Lightweight, high-resolution monitoring for troubleshooting production systems. In *OSDI* (2008), pp. 103–116.

[22] BLOG, X. C. Xen 3.3 feature: Shadow 3.

[23] bochs: the open source ia-32 emulation project. `http://http://bochs.sourceforge.net/`.

[24] BOOTHE, B. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), ACM, pp. 299–310.

[25] BORG, A., BAUMBACH, J., AND GLAZER, S. A message system supporting fault tolerance. *SIGOPS Oper. Syst. Rev. 17*, 5 (1983), 90–99.

[26] BORG, A., BLAU, W., GRAETSCH, W., HERRMANN, F., AND OBERLE, W. Fault tolerance under UNIX. *ACM Trans. Comput. Syst. 7*, 1 (1989), 1–24.

[27] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault-tolerance. In *SOSP* (1995), pp. 1–11.

[28] BUCHACKER, K., AND SIEH, V. Framework for testing the fault-tolerance of systems including OS and network aspects. In *HASE* (2001), pp. 95–105.

[29] BURKS, A. W., AND BURKS, A. R. First general-purpose electronic computer. *IEEE Annals of the History of Computing 3* (1981), 310–389.

[30] BURTSEV, A., SRINIVASAN, K., RADHAKRISHNAN, P., BAIRAVASUNDARAM, L. N., VORUGANTI, K., AND GOODSON, G. R. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *USENIX Annual Technical Conference* (2009).

[31] CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference* (2004), pp. 15–28.

[32] CARGILL, T. A., AND LOCANTHI, B. N. Cheap hardware support for software debugging and profiling. In *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems* (Los Alamitos, CA, USA, 1987), IEEE Computer Society Press, pp. 82–83.

[33] CARTER, W. C. Fault detection and recovery algorithms for fault-tolerant systems. In *EURO IFIP'79* (1979), pp. 725–734.

[34] CARVER, R. H., AND TAI, K.-C. Reproducible testing of concurrent programs based on shared variables. In *ICDCS* (1986), pp. 428–433.

[35] CHEN, M. Y., ACCARDI, A., KICIMAN, E., PATTERSON, D. A., FOX, A., AND BREWER, E. A. Path-based failure and evolution management. In *NSDI* (2004), pp. 309–322.

[36] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *HotOS* (2001), pp. 133–138.

[37] CHEN, Y., AND CHEN, H. Scalable deterministic replay in a parallel full-system emulator. In *PPoPP* (2013).

[38] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2011), ASPLOS '11, ACM, pp. 265–278.

[39] CHISNALL, D. *The definitive guide to the Xen hypervisor*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[40] CHOI, J.-D., AND SRINIVASAN, H. Deterministic replay of Java multithreaded applications. In *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools* (Aug. 1998), pp. 48–59.

[41] CHOI, J.-D., AND SRINIVASAN, H. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (New York, NY, USA, 1998), ACM, pp. 48–59.

[42] CHOI, J.-D., AND ZELLER, A. Isolating failure-inducing thread schedules. In *ISSTA* (2002), pp. 210–220.

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

[44] CHOW, J., LUCCHETTI, D., GARFINKEL, T., LEFEBVRE, G., GARDNER, R., MASON, J., SMALL, S., AND CHEN, P. M. Multi-stage replay with crosscut. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2010), VEE '10, ACM, pp. 13–24.

[45] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proc. NSDI* (Boston, MA, May 2005), pp. 273–286.

[46] CLARKE, S., AND MCDERMID, J. Software fault trees and weakest preconditions: a comparison and analysis. *Software Engineering Journal 8*, 4 (Jul 1993), 225–236.

[47] CORNELIS, F., GEORGES, A., CHRISTIAENS, M., RONSSE, M., GHESQUIERE, T., AND BOSSCHERE, K. D. A taxonomy of execution replay systems. In *In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet* (2003).

[48] CORNELIS, F., RONSSE, M., AND BOSSCHERE, K. D. Tornado: A novel input replay tool. In *In Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA03)* (2003), CSREA Press, pp. 1598–1604.

[49] CORPORATION, A. AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2007.

[50] CORPORATION, I. IA-32 Intel Architecture Optimization Reference Manual, 2006.

[51] CORPORATION, I. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, 2009.

[52] COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. Concurrent control with "readers" and "writers". *Commun. ACM 14*, 10 (1971), 667–668.

[53] CROVELLA, M. E., AND LEBLANC, T. J. Performance debugging using parallel performance predicates. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging* (1993), pp. 140–150.

[54] CROVELLA, M. E., AND LEBLANC, T. J. Parallel performance using lost cycles analysis. In *Supercomputing '94: Proceedings of the 1994 Conference on Supercomputing* (Los Alamitos, CA, USA, 1994), pp. 600–609.

[55] CUI, T., JIN, H., LIAO, X., AND LIU, H. A virtual machine replay system based on para-virtualized Xen. *IFIP International Conference on Network and Parallel Computing Workshops* (2009), 44–50.

[56] CURTIS, R., AND WITTIE, L. D. Bugnet: A debugging system for parallel programming environments. In *ICDCS* (1982), pp. 394–400.

[57] DARLINGTON, J. An experimental program transformation and synthesis system. In *Artif. Intell. 16* (1981), pp. 1–46.

[58] DE OLIVEIRA, D. A. S., CRANDALL, J. R., WASSERMANN, G., WU, S. F., SU, Z., AND CHONG, F. T. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In *ASID '06: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (2006), pp. 66–71.

[59] DIKE, J. A user-mode port of the Linux kernel. In *ALS'00: Proceedings of the 4th Annual Linux Showcase & Conference* (2000).

[60] DIONNE, C., FEELEY, M., AND DESBIENS, J. A taxonomy of distributed debuggers based on execution replay. In *In Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications* (1996), pp. 203–214.

[61] DODD, P. S., AND RAVISHANKAR, C. V. Monitoring and debugging distributed real-time programs. *Softw. Pract. Exper. 22*, 10 (1992), 863–877.

[62] DUNLAP, G. The paravirtualization spectrum, part 2: From poles to a spectrum, 2012. `http://blog.xen.org/index.php/2012/10/31/the-paravirtualization-spectrum-part-2-from-poles-to-a-spectrum/`.

[63] DUNLAP, G. W. *Execution replay for intrusion analysis*. PhD thesis, University of Michigan, 2006.

[64] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI* (2002), pp. 211–224.

[65] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *VEE* (2008), pp. 121–130.

[66] The dwarf debugging standard. `http://dwarfstd.org/`.

[67] ELNOZAHY, E. N. M., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (2002), 375–408.

[68] EMC. The EMC Celerra Family. http://www.emc.com/ products/family/celerra-family.htm.

[69] ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69*, 1-3 (2007), 35–45.

[70] FIELDS, B. A., RUBIN, S., AND BODÍK, R. Focusing processor policies via critical-path prediction. In *ISCA* (2001), pp. 74–85.

[71] FORD, B. Icebergs in the clouds: the other risks of cloud computing. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing* (Berkeley, CA, USA, 2012), HotCloud'12, USENIX Association, pp. 2–2.

[72] FORTUNE, S. J. Dynamic variables, 1981.

[73] Freebench open-source benchmark suite. `http://code.google.com/p/freebench/`.

[74] GAIT, J. A probe effect in concurrent programs. *Softw., Pract. Exper. 16*, 3 (1986), 225–233.

[75] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *NDSS* (2003).

[76] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[77] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *Proc. USENIX* (2006), pp. 289–300.

[78] GNU PROJECT. GDB, 2009. `http://www.gnu.org/software/gdb/`.

[79] GOLDSMITH, S. F., O'CALLAHAN, R., AND AIKEN, A. Relational queries over program traces. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 385–402.

[80] A tale of two pwnies. `http://blog.chromium.org`.

[81] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction* (1982), pp. 120–126.

[82] GRAY, J. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems* (1986), pp. 3–12.

[83] GRISWOLD, R. E., HANSON, D. R., AND KORB, J. T. Generators in icon. *ACM Trans. Program. Lang. Syst. 3*, 2 (1981), 144–161.

[84] GROUP, C. R. Xen-based ReVirt distribution, 2006. `http://www.eecs.umich.edu/~pmchen/covirt/software.html`.

[85] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An application-level kernel for record and replay, 2008.

[86] GUPTA, D., VISHWANATH, K. V., AND VAHDAT, A. Diecast: Testing distributed systems with an accurate scale model. In *NSDI* (2008), pp. 407–422.

[87] GUPTA, D., YOCUM, K., MCNETT, M., SNOEREN, A. C., VAHDAT, A., AND VOELKER, G. M. To infinity and beyond: Time-warped network emulation. In *NSDI* (2006).

[88] HANSEN, W. J. User engineering principles for interactive systems. In *AFIPS '71 (Fall): Proceedings of the November 16-18, 1971, fall joint computer conference* (New York, NY, USA, 1971), ACM, pp. 523–532.

[89] HARRIS, T. L. Dependable software needs pervasive debugging. In *ACM SIGOPS European Workshop* (2002), pp. 38–43.

[90] HO, A., FETTERMAN, M. A., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *EuroSys* (2006), pp. 29–41.

[91] HO, A., HAND, S., AND HARRIS, T. Pdb: Pervasive debugging with xen. *Grid Computing, IEEE/ACM International Workshop on 0* (2004), 260–265.

[92] HUNT, G., AND BRUBACHER, D. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium* (1998), pp. 135–143.

[93] HUSELIUS, J. Debugging parallel systems: A state of the art report. Tech. rep., 2002.

[94] IBM CORPORATION. IBM Storage Controllers. http://www-03.ibm.com/systems/storage/network/index.html.

[95] III, C. B. M., AND GRUNWALD, D. Peabody: The time travelling disk. In *IEEE Symposium on Mass Storage Systems* (2003), pp. 241–253.

[96] JEFFERSON, D. R. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3 (July 1985), 404–425.

[97] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP* (2005), pp. 91–104.

[98] JR., G. B. L. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst. 8*, 1 (1986), 50–87.

[99] K. FRASER, S. HAND, R. N. I. P. A. W., AND WILLIAMSON, M. Safe hardware access with the xen virtual machine monitor. In *OASIS* (2004).

[100] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode* (New York, NY, USA, 2007), WORM '07, ACM, pp. 46–53.

[101] KANG, M. G., YIN, H., HANNA, S., MCCAMANT, S., AND SONG, D. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security* (New York, NY, USA, 2009), VMSec '09, ACM, pp. 11–22.

[102] KILLIAN, C. E., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI* (2007).

[103] KING, S. T., AND CHEN, P. M. Backtracking intrusions. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 223–236.

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

[105] KONURU, R., SRINIVASAN, H., AND CHOI, J.-D. Deterministic replay of distributed java applications. *Parallel and Distributed Processing Symposium, International 0* (2000), 219.

[106] KORF, R. Inversion of applicative programs. In *Proceedings 7th International Joint Conference on Artificial Intelligence* (Vancouver, Canada, 1981), pp. 1007–1009.

[107] LAADAN, O., BARATTO, R. A., PHUNG, D. B., POTTER, S., AND NIEH, J. Dejaview: a personal virtual computer recorder. In *SOSP* (2007), pp. 279–292.

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

[109] LAMPSON, B. Bravo manual, 1978.

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

[111] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. Respec: efficient online multiprocessor replay via speculation and external determinism. In *ASPLOS* (2010), pp. 77–90.

[112] LEFEBVRE, G., CULLY, B., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Tralfamadore: unifying source code and execution experience. In *EuroSys* (2009), pp. 199–204.

[113] LEFEBVRE, G., ET AL. Execution mining. In *Proc. VEE* (Mar. 2012), pp. 145–158.

[114] LEVON, J., AND ELIE, P. Oprofile, 2009. `http://oprofile.sourceforge.net`.

[115] LIN, C.-C., AND LEBLANC, R. J. Event-based debugging of object/action programs. *SIGPLAN Not. 24*, 1 (1989), 23–34.

[116] LIU, H., JIN, H., LIAO, X., HU, L., AND YU, C. Live migration of virtual machine based on full system trace and replay. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing* (New York, NY, USA, 2009), ACM, pp. 101–110.

[117] LIU, H., JIN, H., LIAO, X., AND PAN, Z. XenLR: Xen-based logging for deterministic replay. *Japan-China Joint Workshop on Frontier of Computer Science and Technology* (2008), 149–154.

[118] LIU, X., LIN, W., PAN, A., AND ZHANG, Z. Wids checker: Combating bugs in distributed systems. In *NSDI* (2007).

[119] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., JANAPA, V., AND HAZELWOOD, R. K. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI* (2005), pp. 190–200.

[120] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM, pp. 190–200.

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

[122] MARTIN, M., LIVSHITS, B., AND LAM, M. S. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 365–383.

[123] MCCARTHY, J. The inversion of functions defined by turing machines. In *Automata Studies, Annals of Mathematical Studies*, J. M. C.E. Shannon, Ed., no. 34. Princeton University Press, 1956, pp. 177–181.

[124] MELLOR-CRUMMEY, J. M., AND LEBLANC, T. J. A software instruction counter. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1989), ACM, pp. 78–86.

[125] MOGUL, J. C. Emergent (mis)behavior vs. complex software systems. In *In EuroSys* (2006), pp. 293–304.

[126] MORITZ JODEIT AND MARTIN JOHNS. USB Device Drivers: A Stepping Stone into Your Kernel. In *European Conference on Computer Network Defense* (2010).

[127] MUELLER, T. Virtualised USB Fuzzing for Vulnerabilities. `https://muelli.cryptobitch.de/paper/2010-usb-fuzzing.pdf`.

[128] MUSUVATHI, M., AND QADEER, S. Chess: Systematic stress testing of concurrent software. In *LOPSTR* (2006), pp. 15–16.

[129] NETAPP, INC. NetApp Storage Systems. http://www. netapp.com/products.

[130] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI* (2007), pp. 89–100.

[131] NETZER, R. H. B. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging* (New York, NY, USA, 1993), ACM, pp. 1–11.

[132] NETZER, R. H. B., AND MILLER, B. P. Optimal tracing and replay for debugging message-passing parallel programs. In *In Proceedings of Supercomputing '92* (1992), pp. 502–511.

[133] NETZER, R. H. B., AND WEAVER, M. H. Optimal tracing and incremental reexecution for debugging long-running programs. *SIGPLAN Not. 29*, 6 (1994), 313–325.

[134] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *SOSP* (2005), pp. 191–205.

[135] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing security checks on commodity hardware. In *ASPLOS* (2008), pp. 308–318.

[136] NIST. Summary for CVE–2009–2692. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2692`.

[137] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of zap: A system for migrating computing environments. In *OSDI* (2002).

[138] PAN, D. Z., AND LINTON, M. A. Supporting reverse execution for parallel programs. *SIGPLAN Not. 24*, 1 (1989), 124–129.

[139] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP* (2009), pp. 177–192.

[140] PATIL, H., AND FISCHER, C. N. Efficient run-time monitoring using shadow processing. In *AADEBUG* (1995), pp. 119–132.

[141] PERL, S. E., AND WEIHL, W. E. Performance assertion checking. In *SOSP* (1993), pp. 134–145.

[142] Phoronix Test Suite: An automated, open-source testing framework. `http://www.phoronix-test-suite.com/`.

[143] POOL, J., WONG, I. S. K., AND LIE, D. Relaxed determinism: making redundant execution on multiprocessors practical. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–6.

[144] PRASAD, V., COHEN, W., EIGLER, F., HUNT, M., KENISTON, J., AND CHEN, B. Locating system problems using dynamic instrumentation. In *Ottawa Linux Symposium* (2005).

[145] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 235–248.

[146] REYNOLDS, P., KILLIAN, C. E., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[147] ROSE, C. D. Master's thesis.

[148] ROSE, C. D., AND FLANAGAN, J. K. Constructing instruction traces from cache-filtered address traces (citcat). *SIGARCH Comput. Archit. News 24*, 5 (1996), 1–8.

[149] RUSSINOVICH, M., AND COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (New York, NY, USA, 1996), ACM, pp. 258–266.

[150] SAIDI, A. G., BINKERT, N. L., REINHARDT, S. K., AND MUDGE, T. N. Full-system critical path analysis. In *ISPASS* (2008), pp. 63–74.

[151] SAITO, Y. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging* (New York, NY, USA, 2005), ACM, pp. 69–76.

[152] SALKELD, R., XU, W., CULLY, B., LEFEBVRE, G., WARFIELD, A., AND KICZALES, G. Retroactive aspects: programming in the past. In *Proceedings of the Ninth International Workshop on Dynamic Analysis* (New York, NY, USA, 2011), WODA '11, ACM, pp. 29–34.

[153] SCHUPPAN, V., BAUR, M., AND BIERE, A. Jvm independent replay in java. *Electr. Notes Theor. Comput. Sci. 113* (2005), 85–104.

[154] SHAFIQUE, F., PO, K., AND GOEL, A. Correlating multi-session attacks via replay. In *HOTDEP'06: Proceedings of the 2nd conference on Hot Topics in System Dependability* (Berkeley, CA, USA, 2006), USENIX Association, pp. 3–3.

[155] SHAPIRO, M. W. rdb: A system for incremental replay debugging. Tech. rep., Dept. of Computer Science, Brown University, 1997.

[156] SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Automatic reverse engineering of malware emulators. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2009), SP '09, IEEE Computer Society, pp. 94–109.

[157] SICKEL, S. Invertibility of logic programs. In *Proceedings of the 4th Workshop on Automated Deduction* (1979), pp. 103–109.

[158] SLYE, J. H., AND ELNOZAHY, E. N. Supporting nondeterministic execution in fault-tolerant systems. In *FTCS '96: Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)* (Washington, DC, USA, 1996), IEEE Computer Society, p. 250.

[159] SLYE, J. H., AND ELNOZAHY, E. N. Support for software interrupts in log-based rollback-recovery. *IEEE Trans. Comput. 47*, 10 (1998), 1113–1123.

[160] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (Berlin, Heidelberg, 2008), ICISS '08, Springer-Verlag, pp. 1–25.

[161] SRINIVASAN, S. M., KANDULA, S., ANDREWS, C. R., AND ZHOU, Y. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), USENIX Association, pp. 3–3.

[162] STEVEN, J., CHANDRA, P., FLECK, B., AND PODGURSKI, A. jrapture: A capture/replay tool for observation-based testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2000), ACM, pp. 158–167.

[163] SUNDMARK, D. Replay debugging of embedded real-time systems: A state of the art report, 2002.

[164] SÜSSKRAUT, M., WEIGERT, S., NOWACK, M., DE BRUM, D. B., AND FETZER, C. Parallelize the runtime checks - not the application. In *Workshop on Exploiting Concurrency Efficiently and Correctly (EC)2 (CAV 2009)* (2009).

[165] TAI, K.-C., CARVER, R. H., AND OBAID, E. E. Debugging concurrent ada programs by deterministic execution. *IEEE Trans. Software Eng. 17*, 1 (1991), 45–63.

[166] TEITELMAN, W. Interlisp. *SIGART Bull.*, 43 (1973), 8–9.

[167] THANE, H., AND HANSSON, H. Using deterministic replay for debugging of distributed real-time systems. In *ECRTS* (2000), pp. 265–272.

[168] THANE, H., SUNDMARK, D., HUSELIUS, J., AND PETTERSSON, A. Replay debugging of real-time systems using time machines. In *IPDPS* (2003), p. 288.

[169] THANE, H., SUNDMARK, D., HUSELIUS, J., AND PETTERSSON, A. Replay debugging of real-time systems using time machines. In *IPDPS* (2003), p. 288.

[170] TIP, F. A survey of program slicing techniques. Tech. rep., Amsterdam, The Netherlands, The Netherlands, 1994.

[171] TSAI, J. J. P., FANG, K.-Y., CHEN, H.-Y., AND BI, Y.-D. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Trans. Software Eng. 16*, 8 (1990), 897–916.

[172] VERHOFSTAD, J. S. M. Recovery techniques for database systems. *ACM Comput. Surv. 10*, 2 (June 1978), 167–195.

[173] VETTER, J. S., AND WORLEY, P. H. Asserting performance expectations. In *SC* (2002), pp. 1–13.

[174] VMWARE. A performance comparison of hypervisors, 2007. `http://www.vmware.com/pdf/hypervisor_performance.pdf`.

[175] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: scalable threads for internet services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 268–281.

[176] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *ACM SIGOPS Operating Systems Review* (2005), vol. 39, ACM, pp. 148–162.

[177] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *OSDI* (2002).

[178] WALLACE, S., AND HAZELWOOD, K. M. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO* (2007), pp. 209–220.

[179] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *OSDI* (2004), pp. 77–90.

[180] WHITAKER, A., COX, R. S., SHAW, M., AND GRIBBLE, S. D. Constructing services with interposable virtual hardware. In *NSDI* (2004), pp. 169–182.

[181] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Denali: Lightweight virtual machines for distributed and networked applications. In *USENIX ATC* (2002).

[182] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI* (Boston, MA, Dec. 2002), pp. 255–270.

[183] WITTIE, L. The bugnet distributed debugging system. In *EW 2: Proceedings of the 2nd workshop on Making distributed systems work* (New York, NY, USA, 1986), ACM, pp. 1–3.

[184] WITTIE, L. D. Debugging distributed c programs by real time reply. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging* (New York, NY, USA, 1988), ACM, pp. 57–67.

[185] The paravirtualization spectrum, part 2: From poles to a spectrum. `http://blog.xen.org/`.

[186] XENSOURCE. A performance comparison of commercial hypervisors, 2007. Unpublished.

[187] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation* (2007).

[188] YANG, C.-Q., AND MILLER, B. P. Critical path analysis for the execution of parallel and distributed programs. In *ICDCS* (1988), pp. 366–373.

[189] YANG, Z., ET AL. ORDER: Object centric deterministic replay for Java. In *Proc. USENIX ATC* (Mar. 2011), pp. 341–354.

[190] YOURST, M. T. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS* (2007), pp. 23–34.

[191] YUMEREFENDI, A. R., MICKLE, B., AND COX, L. P. Tightlip: Keeping applications from spilling the beans. In *NSDI* (2007).

[192] ZELLER, A. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE* (2002), pp. 1–10.