Execution replay for intrusion analysis

by

George Washington Dunlap III

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2006

To my loving wife.

ACKNOWLEDGEMENTS

God made me, and gave me the gifts and abilities that I have. He has also provided for me all of my life; through school, money for college, and support. He put me in grad school, provided an advisor who is not only an intelligent computer scientist but also a brother in Christ, and He encouraged me to stay with the program when I was so discouraged that I wanted to quit. All I have done and will do I offer to Him.

My wife, Kristi Dunlap, has been loving and supportive of my work. She has born with working every night and every weekend. She has listened with interest to my description of my work, encouraged me when I was low, helped me to make plans when I was stressed out. She has spent countless hours editing this document, and convincing me when I was stubborn about some phrase or other. Her love, patience, and encouragement has made her yet more dear to me, and I look forward to our years ahead.

My mother, Karen Dunlap, raised me as a single mom. She formed my early years, bought my first, second, and third computers, and has supported me in everything that I've wanted to do.

My advisor, Peter Chen, has supported me financially and guided me through the dissertation process. His insight, understanding, and experience were always helpful and appropriate. Moreover, I was privileged to see him model Christ-like characteristics in his work and family life.

Dominic Lucchetti has been an invaluable friend and co-worker. Without his hours of coding and discussion, and without his keen understanding, SMP-ReVirt would not have happened. I have enjoyed our friendship.

Michael Fetterman was instrumental in understanding the Xen system and getting replay to work. He also invested a very large amount of time working with multiprocessor shadow functionality. His experience and knowledge of the internals of Intel chips was invaluable.

My friend Kwang I could always count on for a “WUZZAH!” and an uplifting conversation. With my friend Misha I have loved our conversations about language, philosophy, and God, and especially talking about life while on a long run. Conversations with my friend Mark P. were always a blessing.

I have received a lot of support, encouragement, and prayer from my friends and family: Ruth, Tom, Hope, Marc, Scott, Erica, Good News Christian a cappella, my church family at Crossroads Community Baptist Church (especially the Living Proof Sunday School class) and at Graduate Christian Fellowship, and the Ann Arbor Scottish Country Dance group.

And finally, I would like to thank XenSource Inc. for bringing me on their team and being flexible while I finished my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

Introduction

Securing today's computers is an urgent and difficult problem. Over the last thirty years, Moore's law has been a great boon to the computer industry, doubling the size and complexity of code available to developers every eighteen months. At the same time, the increasing availability of computers, along with the rise of networking and the Internet has lead to global connectivity, which in turn has brought about a global culture of discussion, information sharing, and commerce.

Unfortunately, the increasing complexity of modern software is exceeding both the developer's ability to write correct, bug-free code, and the administrator's ability to configure a correct, secure system. The rate of increase, along with the economic reality of the marketplace, favors software that gets to the market soonest with the highest number of bugs tolerable, and businesses that spend the least tolerable amount of money on security and administration. These factors guarantee that exploitable bugs and insecure system configurations will exist; the global connectivity and ubiquitous availability of computers guarantees that someone will exploit them. For the foreseeable future, dealing with intrusions will be an inevitable part of every system administrator's job.

The infeasibility of preventing computer compromises makes it vital to analyze intrusions after they occur. Post-intrusion analysis is used to understand the attack, fix the vulnerability that allowed the compromise, repair any damage caused by the intruder, and determine what privileged information the intruder may have stolen. The administrator typically has at his disposal a disk image and system logs, and ideally firewall and network logs [6]. A typical Unix installation logs login attempts, mail processing events, TCP connection requests, filesystem mount requests, and commands issued by the superuser. Windows 2000 systems can log logon and logoff events, file accesses, power start and exit events, security policy changes, and restart and shutdown events.

These logs are used both for detecting an intruder and analyzing what he did. Unfortunately, current logging systems fall short for intrusion analysis because they fail to provide enough information to recreate or understand all attacks. Typical loggers only save a few types of system events, and these events are often insufficient to determine with certainty how the break-in occurred or what damage was inflicted. Instead, the administrator is left to guess what might have happened, and this is a painful and uncertain task. The intrusion analysis published by the Honeynet project typifies this uncertainty by containing numerous phrases such as “may indicate the method”, “it seems reasonable to assume”, “appears to”, “likely edited”, “presumably to”, and “not clear what service was used” [1].

More secure installations may log all inputs to the system, such as network activity and keyboard input. However, even such extensive logging does not enable the administrator to re-create intrusions that involve non-deterministic effects. Many attacks exploit the unintended consequences of non-determinism (e.g., time-of-check to time-of-use race conditions [11])—recent advisories have described non-deterministic exploits in the Linux kernel, Microsoft Java VM, FreeBSD, NetBSD, kerberos, ssh, Tripwire, KDE, and Windows Media Services. Furthermore, the effects of non-deterministic events tend to propagate, so it becomes impossible to re-create or analyze a large class of events without replaying all earlier events deterministically. Encryption is a good example of this: encryption algorithms use non-deterministic events to generate entropy when choosing cryptographic keys, and all future communication depends on the value of these keys. Without logging non-deterministic events, encrypted communication can be decrypted only if the intruder neglects to delete the key.

In summary, current loggers are not **complete**. Although the stated goal of post-mortem analysis is to reconstruct the past, current loggers leave large gaps in knowledge of the past.

1.1 Execution replay for intrusion analysis

We propose filling the knowledge gap left by current loggers using a technique called **execution replay**. Rather than logging events that happen at a system level, execution replay systems log events that affect the execution of a system, with the goal of reconstructing the execution. The original execution is called the **record run** or the **logging run**; subsequent runs are called **replay runs**. During the logging run, the execution replay system logs non-deterministic events that affect the flow of execution. These events are then replayed during the replay run, to make the execution of the replay run identical to that of the logging run.

Replaying the execution takes the system through the exact states that it went through before, during, and after the intrusion. During replay we can stop at arbitrary places to inspect the state and gather more information. Because the entire system is being re-executed, instruction by instruction, we have access to every event on the system and the state at any point; and because we can re-execute the system multiple times, we can refine the level of detail at which we collect information. Execution replay allows access to more information than system-level logs, and counter-intuitively, can also result in a smaller log size.

1.2 Thesis and overview

We intend to show that execution replay is a practical way to add completeness to forensic logging.

Execution replay is not a new technique; many systems have been developed for debugging and fault tolerance. Chapter II makes a survey of prior execution replay systems, their uses, and techniques. In Chapter III we describe the design of a replay system for forensic analysis. We discuss requirements that security places on execution replay systems, contrasting them with those of debugging and fault tolerance, and also the implications for security regarding where the reply layer resides.

Chapter IV describes ReVirt, a system we have built that will log and replay a uniprocessor paravirtualized virtual machine running on a Linux operating system. We make no assumptions about the programs running within the VM. Our worst-case benchmark will log for 18 days[1] before filling a 100GB dedicated log disk, and during logging adds 0-11% run-time over the cost of running within a virtual machine.

With hyperthreaded and multiple-core architectures becoming more commonplace, we need to know how well ReVirt will run on multiprocessor machines. Chapter V discusses methods of combining multiprocessor systems with logging, and describes how we extended ReVirt to detect and replay data sharing between multiple

[1]There are indications that this can be increased to over a month.

processors using pagetable hardware available on all modern processors. To our knowledge, our system is the first to deterministically replay a multiprocessor kernel outside of a simulation.

And finally, Chapter VI contains a summary of the dissertation and the conclusion.

CHAPTER II

Related Work

ReVirt is the first execution replay system targeted at forensic analysis. But execution replay has been used by two other fields: fault tolerance and debugging. This chapter discusses the goals of these different fields, and how they use execution replay to aid them in accomplishing those goals.

2.1 Execution replay for fault tolerance

The goal of fault tolerance is to minimize the effects of systems failures. These types of techniques are used primarily in two types of systems: primary-backup systems and distributed computations.

Primary-backup techniques are targeted at systems that require a very high availability. The goal is to add redundancy to the system so that if the primary suffers any kind of hardware or software failure, a backup can take its place as seamlessly and quickly as possible.

Distributed fault-tolerant techniques are targeted at distributed computations, where the operating system may crash, lose connectivity, or lose power. Models generally assume that when a processor crashes, it will lose all state in **volatile storage**, such as memory, but will retain all **stable storage**, such as disk, and be available for computation at some point in the future. The computation depends upon all nodes running to completion, and the failure of a single node will cause the entire computation to be lost; thus the probability of failure of the entire computation is more than the probability of failure of each of the individual nodes. The goal of fault tolerance in this scenario is to minimize the lost work due to transitory failure of one or more of the nodes.

The primary solution to failure is **checkpointing**, where the state of the process is copied to a place that will be unaffected by the failure. In the primary-backup approach, this is generally another processor that is ready to step in immediately when the primary fails. In distributed fault tolerance, this is generally a disk or other stable storage, which is read after the node has been restored to recover the state of the process.

However, checkpointing alone is not an ideal solution. Frequent checkpoints can be expensive, slowing down the system. In order to make failure **transparent** to the outside world, the system must checkpoint before every output (called **output commit**), so that an external observer cannot tell that a failure has occurred; a full checkpoint before every output causes a perception of lag in the system.

Furthermore, in a distributed computation, simply restoring a process to a previous checkpoint may move the entire system to a state which is not **consistent**: if process P_0 sends a message to process P_1, and then fails and rolls back to a point before sending the message in question, then P_1 has received a process that P_0 does not remember sending. Consistency may require P_1 to be rolled back to a previous checkpoint as well. But this roll-back may subsequently cause a series of **cascading roll-backs**, possibly rolling all processes in the system back to their initial state, losing any advantage that checkpointing would have gained.

Coordinated checkpointing is a class of solutions to the cascading roll-backs problem in which processes coordinate to take checkpoints at times which guarantee that the most recent checkpoint of all processes is in a consistent state. Under these protocols the most any process has to roll back is to the previous checkpoint. This solves the cascading roll-back problem and guarantees that there are no useless checkpoints[25], but does not solve the output commit problem, and can result in an even greater slow-down due to the costs of coordinating a group checkpoint.

Execution replay can be used in conjunction with checkpointing to mitigate these problems. In the fault tolerant computing field, systems which use execution replay are called **message-logging systems**, because the first systems to use it generally modeled all process input and output as a series of messages[13]. This model assumes that if a computation starts in the same state, and is delivered the same messages in the same order, the process will go through the same set of states.

More generally, a process under these systems is assumed to be **piecewise deterministic**(PWD). That is, all non-deterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's **determinant** [4, 5, 25]. Logging and replaying the non-deterministic events in their exact original order can recreate a process's pre-failure state even if that state has not been checkpointed[25]. There are two important parts to the determinant: the **content** of the message, and the **order** in which the message was received.

The primary-backup approach implements message-logging by delivering the same messages to the primary as to the backup; the result should be that both processes are in the same state. In the distributed computation approach, after a failure a process starts at its most recent checkpoint and replays messages until the global system is in a consistent state.

For distributed computation, forms of message logging differ in where the logs are stored and when they make it to **stable storage**. In the most basic form, **receiver-based logging**, each message is stored in a log on the receiver before it is delivered to a process. In **sender-based logging** [32], the process that sends the message keeps it, along with information about the order in which it was received, in volatile memory until it knows the receiver has checkpointed. If the receiver fails, the sender re-sends the message, along with ordering information, to the failed process. In **causal logging** [5], a process piggybacks any determinants not yet saved to stable storage onto messages that it sends to other processes. If that process then fails, processes that received messages from it can help it recover to the state just after sending the message.

Most implementations of primary-backup systems require either special-purpose hardware or software designed with fault-tolerance in mind. This is extra complication is expensive to add to systems. To solve this, Bressoud [14] inserted a **hypervisor** (also called a **virtual machine monitor**) between the hardware and the operating system, using it to implement a primary-backup system. The hypervisor intercepted all hardware events that went to the primary, and sent them to the hypervisor on the backup machines. Thus the execution on the primary was replayed on the backup machines so that they were exact replicas of the original. Running inside of a virtual machine allows the fault-tolerance capability to be added to unmodified hardware and software. Napper [40] recently developed a similar system using a Java virtual machine.

For further reading on message-logging systems for fault tolerance, the reader is referred to [25].

2.2 Execution replay for debugging

The goal of debugging is to discover why a program is not functioning the way it was intended to function. One of the classic methods of debugging is called **cyclic debugging**, in which a programmer iteratively runs a program exhibiting a bug, centering in on the cause of the bug by inspecting state, setting breakpoints, single-stepping through execution, etc. For this purpose, the programmer needs to see the **state** of the program, and the **execution** as it affects and is affected by that state.

The basic tool for debugging, therefore, is re-execution, and the most ubiquitous form is what might be called "hopeful re-execution": re-execute the program with similar starting conditions, and hope that the resulting execution will be similar enough to some previous execution to be able to track down the bug. Checkpointing can be used to speed up this process [27].

While this works for **deterministic** bugs, it does not work well for **non-deterministic** bugs, which only occur probabilistically; or for **heisenbugs**[1], which disappear once debugging tools are enabled.

The most common source of non-deterministic bugs has been in tightly-coupled shared-memory multiprocessors. For this reason the earliest work on execution replay for debugging was in the parallel computing community, because **race conditions**, which depend on the exact interleaving of shared memory reads and writes, can be very difficult to debug.

Early **loosely-coupled** parallel computation systems were based on message passing architectures, and thus used protocols similar to the message-logging proto-

[1]Heisenbugs are named for Heisenberg's Uncertainty Principle, which in quantum mechanics describes fundamental limits on the precision at which a quantum particle's velocity and position can be measured. Metaphorically it refers to any situation where the act of observing something changes what is observed.

cols of distributed fault tolerance [22, 37, 50]. Pan, et al.[42] attempted to do the same for **tightly-coupled** systems by instrumenting every shared read and logging the value read. However, this turned out to be prohibitively expensive [20].

Because logging the data value of shared-memory reads was too expensive, the parallel computing field went quickly to an **order-based** approach. The observation was that if the **order** of shared-memory operations was preserved, then the reads would result in the same values. Instant Replay [38] started the field in this area. Instant Replay surrounded all accesses to shared objects with per-object locks, logging the order in which processes acquired locks and replaying it in subsequent executions. Subsequent research using this technique reduced redundant constraints[41, 48] and further optimized logging by using Lamport clocks per objects[7, 43]. JaRec[29] implemented a replaying of locking order within a JVM.

Instant replay and subsequent systems which replay locking order rely on all accesses to shared memory to be protected consistently by locks. They can tolerate **synchronization races**, which occur when locks are incorrectly applied, but they cannot tolerate **data races**, where locks are absent or inconsistently applied.

A quantum leap for all execution replay systems came with the implementation of a hardware instruction counter in the HP-RISC architecture, and the description of an efficient software **instruction counter** in [39]. Previous debugging and replay execution systems could handle **synchronous** events in a system, but could not handle **asynchronous** events, such as interrupts or task-switches, that can happen at arbitrary points in the instruction stream. Instruction counters can be used to identify precise points in an **instruction stream** in the presence of loops and recursion, so that interrupts can be re-delivered at the same point. In the absence of hardware support, code can be instrumented to count every backwards branch with a reasonable amount of overhead[39, 48, 49]. These event counters are especially useful for debugging to identify a particular instruction in an instruction stream, rather than a particular instruction in the program [12].

Bacon et. al[8] explored supporting multiprocessor replay in hardware by snooping the cache-coherence protocol. Their simulated system uses a hardware instruction counter piggy-backed to cache-coherence messages to identify races. Flight Data Recorder (FDR)[55] is a modern version of the same idea. FDR uses specialized hardware to be able to log and replay the last 1 second of execution. In further work, Xu et al. developed **regulated transitive reduction**[56], a way of modifying constraints which preserves the necessary ordering properties while reducing the number of constraints and making constraints more easily compressed.

Russinovich et al introduced the **repeatable scheduling algorithm** (RSA)[45, 46], which extends the idea of asynchronous events in [39] to include schedule preemptions, and discusses what operating system support is necessary to replay discusses extending the idea of an them.

Execution replay is also discussed by Elnozahy in [26] for debugging, along with trace generation, but no further work seems to have been done on the project.

There are a number of more recent projects for uniprocessor replay that have covered various aspects of data collection and methods of dealing with scheduling. The DejaVu project [18] implements a modified Java Virtual Machine (JVM). Input methods are instrumented to log the input to a program, and scheduling is constrained to happen at pre-defined **yield points**; other Java-based solutions include jRapture[51], which did only data replay (no asynchronous events). Other systems that do only data-replay include a system described in [12] to implement reverse debugging commands (reverse single-step, reverse breakpoint, etc), and TORNADO[21]. Both of these instrument system calls, logging the results and replaying them.

CHAPTER III

Design Requirements

The target use of a system should guide its design. Execution replay systems have existed for years, but to date none have targeted security. Security places unique design requirements on an execution replay system. The previous chapter discussed related work for execution replay in the fields of debugging and fault tolerance. This chapter contrasts the requirements for debugging and fault tolerance with the use of execution replay for security.

3.1 Security Demands

3.1.1 Function for arbitrary code

Debuggers are used by programmers, who presumably have access to the code being debugged, and frequently are able to chose, or at least influence, the programming platform and discipline. Because of this, execution replay systems targeted at debugging frequently require modifications that only a developer would be able to make: most require the system to be written in a particular language, instrumented with a particular compiler, or linked with a specific library. Similarly, fault tolerant systems are generally written with some kind of fault tolerance in mind[1].

[1]The exception being Bressoud [14].

The system administrators who have to deal with intrusions are rarely involved in the development of software running on the systems they administer. They may have input into which of several available programs are run, but they are not in a position to dictate what language, library, discipline, or compiler is used in those programs, and they are rarely able to re-compile a program with any kind of instrumentation. Furthermore, a successful attacker can usually introduce arbitrary (uninstrumented) machine code.

Therefore, a replay execution system must be able to log and replay arbitrary machine code, independent of language, library, or compiler.

3.1.2 Consideration of corner cases

Execution replay for security is considerably different from replay for debugging purposes. Where security is an issue, all corner cases must be considered because any possible weakness can be exploited by an attacker. For ordinary debugging corner cases that are unlikely to happen can be ignored.

Replay debuggers are typically targeted for specific functionalities that are limited by the use for which the program is being written. The programmer using execution replay for debugging has the goal of making the program work. The only enemy in debugging is the programmer himself. Programmer mistakes occur, but they are generally very limited in scope. This enables the designers to only log a subset of possible events—those which are expected to be significant to the execution of the target system. Early systems, for instance, targeted scientific parallel computations, which only needed to read static input and communicate with one another; external network, or even process IDs of the running processes were not considered important to be replayed. More recent systems targeted at debugging general uniprocessor applications log more information, but still only enough to be useful to a large enough segment of programs. Convoluted corner cases of questionable use can be safely "legislated away" — simply documented (if that) and left to the programmer to avoid.

For instance, we have an example we encountered from a version of ReVirt for normal host processes. Suppose that two processes are sharing memory. In order to replay their execution, under normal conditions, it suffices to replay their scheduling order [46]. But suppose that one process calls the `read` system call, with the target buffer in memory shared between the process, and that the read is interrupted by a page fault, allowing the other process to execute. To replay correctly, the schedule interrupting a system call in the middle must also be replayed.

The previous scenario is difficult to deal with but not impossible. But how many programs make `read` system calls into explicitly shared memory? One can easily write this behavior off as "not supported" and still debug a large number of applications. These types of scenarios are not even discussed in other host-based replay execution systems.

Many replay execution systems are only targeted at surviving a very limited subset of bugs. Most software-based parallel execution replay systems, for example, surprisingly assume the absence of data races, and will not execute correctly in their presence. They rely instead on using dynamic race detection algorithms during replay, which will stop the program once a race is detected so that it can be removed. This is tolerable because the goal is to find bugs. This method finds the first bug, though any buggy executions past the first race in the execution are lost. (Benign data races such as spin locks cannot be tolerated by these systems.)

Finally, programmer mistakes are limited in scope. For instance, a programmer may call the `htonl()` function instead of the `ntohl()` function, but he will not accidentally call functions internal to the replay system (assuming an appropriate naming convention). Even wild writes can be modeled as random, and unlikely to affect the replay system even if it shares the same memory space. The chance of the programmer accidentally calling a system call directly to change the protections on the code of the replay library, and then modify it a meaningful way, is slim indeed.

Fault-tolerance systems attempt to deal with faulty hardware or non-deterministic bugs in the program or the operating system. Frequently failures are modeled as **fail-stop**: that is, they assume that on a failure, the process or processor will simply stop executing. After a time-out, a recovery system will activate and attempt to restore the failed system. The chances that a program bug or a hardware failure will subvert the recovery system are considered extremely low, and therefore not considered.

In security, the situation is far different. The execution replay system must be able to tolerate an active adversary. This adversary must be assumed to have the binary code, if not the source code, for every program on the system. An adversary will engineer the system to a state where unlikely corner cases can be exploited.

Therefore, an execution replay system targeted at intrusion analysis must be bulletproof: any execution possible during logging must be able to be replayed. Corner cases, such as the one discussed above, must be either handled correctly or eliminated.

3.1.3 Logging for sufficient time periods

Few of the debugging papers we surveyed described how long they intended the logging phase of their replay execution system to be run, or (related but not the same) how far back in time they are able to go. Although it is feasible for a filesystem or database bug to go unnoticed for days or weeks, many bugs are fail-stop and cause an application crash within a short time of being triggered. In fact, a recent hardware-based multiprocessor replay execution design had the explicit goal of being able to replay only the last 1 to 1.3 seconds of execution[55].

The main factor limiting the logging phase is the amount of storage necessary for the logs. This varies widely from system to system, depending on the technique used and the type of benchmarks run, though numbers tend to be bimodal: less than 1 KB/sec for some systems, more than 1 MB/sec for others. One megabyte per second translates to 84 GB/day; under such a system, a dedicated 100GB log disk would fill up in less than two days.

Distributed fault-tolerant systems keep only enough data replay from the last checkpoint. Several checkpoints per day is not an unreasonable assumption.

In the primary-backup replay execution system, data from the primary is only stored long enough to deliver it to the backups. Once the backups consume the data, it is discarded. This means that from a log size perspective, they need only make sure that the data captured at the primary fits in the “pipeline” to the backups.

An intrusion is rarely discovered less than a few hours after the initial break-in, and frequently is not discovered until days or weeks afterward. A replay execution system targeting intrusion analysis needs to deal with these long intervals in two ways: first, it needs to be able to log for extended periods of time in an efficient manner; and second, during replay it needs to travel efficiently to different points in the log.

3.1.4 Practicality for production use

A security "feature" must not render a system unusable. In order to be useful for security, an execution replay system must not have an unreasonable deployment cost. There are two general costs to consider. The first cost is overhead while logging. We believe that a 2-times slowdown is a reasonable slowdown for sites interested in security; but the lower the overhead the better.

The second is the cost of moving to a new system, either software or hardware. Moving to a new OS, new incompatible hardware, or rewriting all software means throwing away a large amount of work that has gone into making existing applications secure. An execution replay system for security should minimize the number of changes required to use the system.

3.2 Interface and security

Computing systems are generally composed into layers: applications call libraries, libraries make system calls into the kernel, kernels access devices, etc. Between layers, there is an **interface** that defines how the two layers interact. The execution replay system resides in the layer below the system being replayed. It replays the interface to the layer above it, logging and replaying non-deterministic inputs that cross the interface.

There are two ways that an attacker could subvert a system, and the interface affects both. First, the attacker could try to break through the interface itself, into the layer providing the interface. This is called a **layer-below attack**. Accordingly, the layer providing the interface needs to be free of exploitable bugs. The more wide, rich, and complex this interface, the more complex the code providing it and the harder it is to keep free of bugs. A smaller, simpler interface will result in smaller, less complex code, which is easier to code accurately.

Secondly, even if the attacker can not break through the interface, there may be corner cases the interface provides which the execution replay system can not handle correctly[2]. If an attacker can engineer the system into one of these corner cases, he can break replay, rendering the remaining log useless, and preventing anyone from observing his actions. The simpler and smaller the interface, the more likely the replay system is going to be able to replay all cases.

For security purposes, a smaller interface is better. Unfortunately, with a smaller interface there is trade-off between security and functionality. For example, consider replaying the entire interface provided by the Linux kernel. The interface is very rich and functional, but as a result, less secure, because there is a higher chance that we will make an error imlementing it. We could increase security by limiting the interface. However, limiting the interface reduces functionality, thereby reducing the range of native applications we can log and replay.

One way to eliminate this trade-off is to use a **virtual machine**. A virtual machine eliminates the trade-off between security and functionality by adding a layer of indirection: a **guest kernel** runs on the small interface provided by a **virtual machine monitor** (VMM), and provides a rich, full-kernel interface to applications above it. Placing the execution replay system in the VMM gives the advantage of a small interface while retaining the rich functionality provided by a kernel, but at the cost of virtualization. Chapter IV discusses virtual machines in more detail.

[2]This applies to any security service. Garfinkel [28] discusses some of the potential pitfalls of trying to observe the UNIX interface.

3.3 Summary

To be practical for intrusion analysis, an execution replay system must work for arbitrary code, under all corner cases; it must be able to keep several weeks worth of logs; it must not add more than 2x overhead. It should also minimize both the changes required of a user, and should have as narrow an interface as possible.

CHAPTER IV

Execution Replay for Virtual Machines

This chapter describes our work on execution replay for virtual machines. We begin by describing virtual machines and why they are useful to our work. We then describe User-Mode Linux (UML), the virtual machine on which we implemented the second version of ReVirt[1]. Then we discuss the implementation of the replay execution system for UML. Finally, we describe the results of our experiments, and our experience using ReVirt to enable forensic analysis.

4.1 Virtual machines

A **virtual-machine monitor** (VMM) is a layer of software that faithfully emulates the hardware of a complete computer system[30]. The abstraction created by the virtual machine monitor is called a virtual machine. The hardware emulated by the VMM is very similar (often identical) to the hardware on which the VMM is running, so the same operating systems and applications that run on the physical machine can also run on the virtual machine. The host platform that the VMM runs on can be another operating system (the **host** operating system) or the bare hardware. The operating system running in the virtual machine is called the **guest**

[1]The original ReVirt system was built on a modified version of UMLinux [16, 24], from the University of Erlangen-Nurnberg.

operating system to distinguish it from the host operating system, which is running on the bare hardware. The applications running on top of the guest operating system are called guest applications, to distinguish them from applications running on the host operating system (of which the VMM is one). The VMM runs in a separate domain from the guest operating system and applications; for example, the VMM may run in a supervisor mode and the guest software may run in unprivileged mode.

Our research project, CoVirt, is interested in enhancing security by running the target operating system and all target services inside a virtual machine, (making them guest operating system and applications), then adding security services in the VMM or host platform [17].

Of course, even the VMM may be subject to security breaches. Fortunately, the VMM makes a much better trusted computing base than the guest operating system, due to its narrow interface and small size. The interface provided by the VMM is similar or identical to the physical hardware (CPU, memory, disks, network card, monitor, keyboard, mouse), whereas the interface provided by a typical operating system is much richer (processes, virtual memory, files sockets, GUIs). The narrow VMM interface restricts the actions of an attacker. In addition, the simpler abstractions provided by a VMM lead to a code size that is several orders of magnitude smaller than a typical operating system, and this smaller code size makes it easier to verify the VMM. As we discussed in Chapter III, the narrow interface of the VMM also makes it easier to log and replay.

Virtual machines can be classified by how similar they are to host hardware. At one extreme, traditional virtual machines such as IBM's VM/370 [30] and VMware [53] export an interface that is backward compatible with the host hardware (the interface is either identical or sightly extended). Operating systems and applications that were intended to run on the host platform can run on these VMMs without change. At the other extreme, language-level virtual machines like the Java VM export an interface that is completely different from the host hardware. These VMMs can run only in operating systems and applications written specifically for them.

Other virtual machines such as the VAX VMM security kernel [34] fall somewhere in the middle—they export an interface that is similar to but not identical to the host hardware [10]. These types of VMMs typically deviate from the host hardware interface when interacting with peripherals. Virtualizing the register interface to peripherals controllers is difficult and time consuming, so many virtual machines provide higher-level methods to invoke privileged instructions and I/O. This technique is called **paravirtualization**. A guest operating system must be ported to run on these VMMs. Specifically, the device drivers in the guest kernel must use the higher-level methods in the VMM; for example, a disk device driver might use host system calls read and write to access the virtual disk. The work required to port a guest operating system to these types of VMMs is similar to that done by device manufacturers who write drivers for their devices.

4.2 User-Mode Linux

The version of ReVirt discussed in this chapter uses a virtual machine called User-Mode Linux, or UML [23]. UML is a paravirtualized virtual machine—the VMM in UML exports an interface that is similar but not identical to host hardware.

4.2.1 UML structure and operation

The virtual machine in UML runs as two user processes on the host[2]. The guest operating system runs in one host process, and all guest applications run inside the other host process. The guest operating system in UML runs on top of the host operating system and uses host services (such as system calls and signals) as the interface to peripheral devices). We call this virtualization strategy **OS-on-OS**, and we call the normal structure where target applications run directly on the host operating system **direct-on-host**. The guest operating system used in this paper is Linux 2.4.18, and the host operating system is also Linux 2.4.18[3].

The VMM in our version of UML is a combination of the POSIX PTRACE interface, and some extensions implemented as a loadable module in the host Linux kernel. The PTRACE system allows our guest kernel process to interpose whenever a guest user process makes a system call or receives a signal (such as a `SEGV`), while the kernel module allows us to flexibly add other features to the VMM.

Most instructions executed within the virtual machine execute directly on the host CPU. Memory accesses are translated by the host's MMU. Memory spaces for guest processes are set up through an extension to the Linux `/proc` filesystem. A special file is made, named `/proc/mm`, which allows the guest kernel to allocate and manipulate address spaces for guest processes. When the guest kernel creates a new guest process, it allocates an address space by opening `/proc/mm`. To context switch from one guest process to another, it switches the active address space via the (new)

[2]The mode we run UML in is called SKAS mode, or single-kernel address space mode. This requires an extension to the host kernel to allow UML to maintain different address spaces. UML can be run in another mode, called tracing-thread or TT mode, in which no host kernel modification is necessary.

[3]The guest and host operating systems can also be different. We use the same operating system for guest and host to enable more direct comparison between running applications on the UML guest and running applications directly on the host.

Host component or event	Emulation mechanism in UML
hard disk	host raw partition
CD-ROM	host `/dev/cdrom`
floppy disk	host `/dev/floppy`
network card	`TUN/TAP` virtual Ethernet device
console	`stdout`
video card	none (display to remote X server)
current privilege level	guest kernel or guest user process executable
system calls	ptrace notification
timer interrupt	timer + `SIGALRM` signal
I/O device interrupts	`SIGIO` signal
memory exception	`SIGSEGV` signal
enable/disable interrupts	mask signals

Table 4.1: Mapping between host components and UMLinux equivalents

`switch_mm()` system call.

Because the guest kernel is in a separate address space from the guest process, the guest processes have the same memory range available to them as they would if they were running on the host. This is useful in forensic testing, because many of the exploits are very sensitive to stack, heap, and text segment placement in memory.

UML provides a software analog to each peripheral device in a normal computer system. Table 4.2.1 shows the mapping from each host component or event to its software analog in the virtual machine. UML uses a host file or raw device to emulate the hard disk, CD-ROM, and floppy. Our version of UML uses the `TUN/TAP` virtual Ethernet device in Linux to emulate the network card. UML uses standard `input/output` (`stdin/stdout`) on the host to display console output and read keyboard input. UML also uses no video card; instead it displays graphical output to a remote X server (which would typically be the host's X server).

UML has two processes, one for the guest kernel and another for guest processes, only one of which is running at any given time[4] This is the analog for the virtual

[4]The unmodified PTRACE interface allows a minor race where both processes are schedulable

mode bit. When the guest user-mode process is running, any system calls it makes are redirected to the guest kernel process, which executes the system call on behalf of the process, and may context-switch the user-mode process to a different guest process. When the guest kernel process is running, it can make host system calls, handle its own segmentation faults, access all of the virtual memory, and modify the guest's memory map.

`SIGALRM`, `SIGIO`, and `SIGSEGV` signals are used to emulate the hardware timer, I/O device interrupts, and memory exceptions. If the guest user process receives a `SIGSEGV`, the host kernel redirects it to the guest kernel. The guest kernel process delivers these signals to the registered signal handler in the guest kernel. These signal handlers are the equivalent of the timer-interrupt, I/O-interrupt, and memory exception handlers in a normal operating system.

UML emulates the enabling and disabling of interrupts by masking signals (using the `sigprocmask` system call).

4.3 Logging and replaying UML

4.3.1 Overview

Logging is widely used for recovering state. The basic concept is straightforward: start from a checkpoint of a prior state, then roll forward using the log to reach the desired state. The type of system being recovered determines the type of information that needs to be logged: database logs contain transaction records, file system logs contain file system data. Replaying a process requires logging the non-deterministic events that affect the process's computation. These log records guide the process as it re-executes (rolls forward) from a checkpoint. Most events are deterministic (e.g.

for a short time. We modified the PTRACE interface to take out this race to help reduce the size of the log.

arithmetic, memory, branch instructions) and do not need to be logged; the process will re-execute these events in the same way during replay as it did during logging.

Non-deterministic events fall into two categories: time and external input. Time refers to the exact point in the execution stream at which an event takes place. For example, to replay an interrupt, we must log the instruction at which the interrupt occurred. External input refers to data received from a non-logged entity, such as a human user or another computer. External input enters the processor via a peripheral device, such as a keyboard, mouse, or network card.

Note that output to peripherals does not affect the replaying process and hence need not be logged (in fact, output to peripherals will be reconstructed during replay). Non-determinism in the micro-architectural state (e.g. cache misses, speculative execution) also need not be logged, unless it affects the architectural state. Replaying a shared-memory multiprocessor requires logging the interleaving of memory operations. Chapter V discusses execution replay for a shared memory multiprocessor.

4.3.2 ReVirt

This section describes how we apply the general concepts of logging to enable replay of UML running on x86 processors. ReVirt is implemented as a set of modifications to the host kernel.

Replay must start at the same point as logging did, so we need to restore the state. Logically, we do this by checkpointing the state of the virtual disk before starting, and then taking checkpoints periodically to avoid replaying from the very beginning of the run.

Log records are added and saved to disk in a manner similar to that used by the Linux syslogd daemon. The VMM kernel module and the kernel hooks add log records to a circular buffer in host kernel memory, and a host process(`rlogd`) consumes the buffer and writes the data to a log file on the host.

ReVirt must log all non-deterministic events that can affect the execution of the virtual-machine process. Note that many non-deterministic host events do not need to be logged, because they do not affect the execution of the virtual machine. For example, host hardware interrupts do not affect the virtual-machine process unless they cause the host kernel to deliver a signal to the virtual-machine process. Likewise, the scheduling order of the other host processes does not affect the virtual-machine process because there is no inter-process communication between the virtual-machine process and other host processes (no shared files, memory, or messages).

ReVirt does have to log asynchronous virtual interrupts (synchronous exceptions like `SIGSEGV` are deterministic and do not need to be logged) and non-deterministic scheduling events. Before delivering a `SIGALRM` or `SIGIO` host signal (representing virtual timer and I/O interrupts) to the virtual-machine process, ReVirt logs sufficient information to re-deliver the signal at the same point during replay. To uniquely identify the interrupted instruction, ReVirt logs the program counter and the number of branches executed since the last interrupt [15, 39]. Because the x86 architecture allows a block memory instruction (repeat string) to be interrupted in the middle of its execution, we must also log the register (`ecx`) that stores the number of iterations remaining at the time of the interrupts.

x86 processors provide a hardware performance counter that can be configured to compute the number of branches that have executed since the last interrupt[2]. The `retired_branch_type` configuration of this performance counter on the Intel P4 processor counts branches. We configure the `retired_branch_type` counter to count only user-mode branches. This makes it easier to count the number of branches precisely, because it keeps the count independent of the code executed in the kernel interrupt handlers.

In addition to logging asynchronous virtual interrupts, ReVirt must also log all input from external entities. These include most virtual devices: keyboard, mouse, network interface card, real-time clock, CD-ROM, and floppy. Note that input from the virtual hard disk is deterministic, because the data on the virtual hard disk will be reconstructed and re-read during replay. One can imagine requiring the user to insert the same floppy disk or CD-ROM during replay, in which case reads from the CD-ROM and floppy would also be deterministic and would not need to be logged. However, we do not expect data from these sources to be a significant portion of the log, because these data sources are limited in speed by the user's ability to switch media[5].

The UML guest kernel reads these types of input data by issuing host system calls `recv`, `read`, and `gettimeofday`. The VMM kernel module logs the input data by intercepting these system calls. In general, ReVirt must log any host system call that can yield non-deterministic results.

The x86 architecture includes a few instructions that can return non-deterministic results, but that do not normally trap when running in user mode. Specifically, the x86 `rdtsc` (read timestamp counter) and `rdpmc` (read performance monitoring counter) instructions are difficult to log. To make the virtual-machine process completely deterministic during replay, we set the processor control register (`cr4`) to trap when these instructions are executed. We emulate the `rdtsc` instruction, executing and logging the result during the logging run, and replaying the result from the log during replay. We disallow the `rdpmc` in the guest kernel and guest applications.

[5]If the CD-ROM is switched by an automated jukebox, then the jukebox can participate in replay and CD-ROM reads can be considered deterministic

During replay, ReVirt prevents new asynchronous virtual interrupts from perturbing the replaying virtual-machine process. ReVirt plays back the original asynchronous virtual interrupts using the same combination of hardware counters and host kernel hooks that were used during logging. ReVirt goes through two phases to find the right instruction at which to deliver the original asynchronous virtual interrupt. In the first phase, ReVirt configures the `retired_branch_type` performance counter to generate an interrupt after most (all but 128) of the branches in that scheduling interval. In the second phase, ReVirt uses breakpoints to stop each time it executes the target instruction. At each breakpoint, ReVirt compares the current number of branches with the desired amount. The first phase executes at the same speed as the original run and is thus faster than the second phase, which triggers a breakpoint each time the target instruction is executed. The second phase is needed to stop at exactly the right instruction, because the interrupt generated by the retired_branch_type counter does not stop execution instantaneously and may execute past the target number of branches.

Replay can be conducted on any host with the same processor type and stepping as the original host[6]. Replaying on a different host allows an administrator to minimize downtime for the original host.

4.3.3 Using ReVirt to analyze attacks

ReVirt enables an administrator to replay the complete execution for a computer before, during, and after the attack. Two types of tools can be built from this foundation to assist the administrator to understand the attack.

The first type of tool runs outside of the virtual machine. Examples might include

[6]The main limiting factor on x86 to how similar two processors must be is the `cpuid` instruction, which is non-privileged and cannot be trapped, but exposes detailed information about the processor type, including its "stepping" number.

debuggers, disk analyzers, or other tools that analyze the state of the virtual machine (address space, registers, disk data). The advantage of these off-line tools is that they do not depend on guest kernel or guest applications. For example, an off-line tool can inspect the contents of a directory even if the attacker has replaced the command that normally lists the directory.

A good example of this tool is Backtracker[35]. Backtracker is a tool that records information about system calls so that, given a detection point (such as a modified binary like `/bin/ls`) you can track backwards to the original break-in.

IntroVirt[33] can use this method as well. IntroVirt assumes that when a vulnerability is discovered, the software authors will write a **predicate** that can detect when a vulnerability is triggered. If deployed on a running system after a vulnerability is discovered, it can detect when a given vulnerability is triggered, allowing system administrators extra leeway when applying patches to their system. But the real power comes when used in conjunction with ReVirt, because it can then answer the question, "Was this vulnerability ever exploited?" It allows the administrator to retroactively determine if the vulnerability was exploited before the vulnerability was known to the security community. If an intrusion is discovered, ReVirt will enable the administrator to gather detailed information about the attack and subsequent illicit activity.

The second type of tool runs inside the virtual machine. ReVirt supports the ability to continue live (i.e., non-replaying) execution at any point in the replay. An administrator can use this ability to run new guest commands to probe the virtual machine state. For example, the administrator can stop the replay after a suspicious point and use normal guest commands to edit the current files, list the processes, and debug processes. Furthermore, an external tool trying to analyze the state has to duplicate much of the guest kernel functionality: cached data may not have made it to disk, guest virtual pages may have been swapped out to disk, and so on.

IntroVirt uses this method as well. When necessary, IntroVirt can probe the virtual machine by executing live requests inside the virtual machine. The virtual machine cannot switch back to replaying after being perturbed in this manner, because the instruction counts will not apply to the revised state. To continue the replay beyond the perturbed point, the process is checkpointed before perturbing it, restoring after the perturbation is done.

The real power of ReVirt is the ability to repeat the break-in to gather more information. This power was demonstrated during the development of Backtracker. The first step was to put up a honeypot with a known vulnerability running ReVirt and "collect" attacks. ReVirt allowed us to use these same number of attacks repeatedly to develop and hone the Backtracker tool. Without ReVirt, Backtracker could have been developed, but each new iteration the developers would have had to set up and wait for more attacks.

4.4 Experiments

This section validates correctness and quantifies virtualization and logging overhead for our modified UML and ReVirt logging and replay system. All experiments in this chapter are run on a computer with an Intel P4 processor, 256MiB of memory. The guest kernel is Linux 2.4.18 ported to UML, running in SKAS mode, and the host kernel is Linux 2.4.18 with the UML host SKAS patch and the ReVirt patch. The virtual machine is configured to use 192MB of "physical" memory. The virtual hard disk is stored on a raw disk partition on the host to avoid double buffering the virtual disk data in the guest and host file caches, and to prevent the virtual machine

Workload	UML runtime (normalized to direct-on-host)
POV-Ray	1.00
SPECweb99	1.15
kernel build	1.76
daily use	≈1

Table 4.2: Virtualization overhead. This table shows the overhead of running applications in UML. Runtime is normalized to the runtime when running directly on the host.

from benefiting unfairly from the host's file cache.

We evaluate our system on five workloads. All workloads start with a warm guest file cache. POV-Ray is a CPU-intensive ray-tracing program. We render the benchmark image from the POV-ray distribution at quality 8. Kernel-build compiles the complete Linux 2.4.18 kernel (`make clean`, `make dep`, `make bzImage`). SPECweb99 is a benchmark that measures web server performance; we use the 2.0.36 Apache web server. We configured SpecWeb99 with 15 simultaneous connections spread over two clients connected to a 100 Mb/s Ethernet switch. Both workloads exercise the virtual machine intensively by making many system calls. They are similar to the I/O-intensive and kernel-intensive workloads used to evaluate Cellular Disco [31]. We also used ReVirt and UML as the author's desktop machine for 24 hours to get an idea of the virtualization and logging overhead for day-to-day use.

Each result represents the average of three runs (except for the daily-use test, which represents a single 24-hour period). Variance across runs is less than 3%.

4.4.1 Virtualization overhead

Our first concern is the time overhead that arises from running all applications in the UML virtual machine. We compare running all application within UML with running them directly on a host Linux 2.4.18 kernel. The host and guest file systems have the same versions of all software exercised in the tests (based on Redhat 9).

Table 4.2 shows the results. UML with our host optimizations adds very little overhead for compute-intensive applications such as POV-Ray. We also perceive no overhead when using UML for interactive jobs such as e-mail, editing, word processing, and web browsing.

The overheads for SPECweb99, kernel-build, and postmark are higher because they issue more guest kernel calls, each of which must be trapped by the VMM kernel module and reflected back to the guest kernel by sending a signal. In addition, kernel-build causes a large number of guest processes to be created, each of which maps its executable pages on demand. Each demand-mapped page causes a signal to be delivered to the guest kernel, which must then ask the host kernel to map the new page.

We believe the overheads for using UML are low enough to be unnoticeable for normal desktop use. While overheads are higher for workloads that use the guest kernel intensively, we believe that even an overhead of 76% (normalized runtime of 1.76) is not prohibitive for sites that value security.

For reference, VMWare Workstation 3.1 has a normalized runtime of approximately 1.25 for kernel-build. Xen[9] is a paravirtual VMM designed specifically for virtualization, and the kernel-build numbers are much lower, at about 1.03. Chapter V discusses our implementation of ReVirt on Xen.

4.4.2 Validating ReVirt correctness

Our next goal was to verify that the ReVirt system faithfully replays the exact execution of the original run. For these runs, we add extensive error checking to alert us if the replaying run deviated from the original. At every system call and virtual interrupt, we log all register values and the branch count, and verify that these values are the same during replay.

In addition, ReVirt's mechanism for replaying interrupts verifies correctness. The `eip,branch_count` tuple "space" is very sparse, so if the program takes a different execution path during replay than it did during logging, it is very likely that the `eip,branch_count` tuple will simply not exist in the replaying run; even if it does, the chances that all subsequent tuples will add up is extremely unlikely. ReVirt is configured to halt and give an error if it can not find the instruction during the replay run. We found that this detected errors effectively while we were developing ReVirt.

We first ran two micro-benchmarks in the virtual machine to verify that virtual interrupts were being replayed at the same point at which they occurred during logging. The first micro-benchmark ran two guest processes that share an `mmap`'ed memory region. Each guest process increments a shared variable 10,000,000 times, prints the resulting value, then repeats. Because the two guest processes share this variable, the output of process A depends on how many iterations process B executed by the time process A prints the value. The second micro-benchmark ran a single process that increments a variable in an infinite loop. The process prints the current value when it receives a signal. This test verifies that the guest kernel re-delivers the signal at the same point as logging.

We ran each micro-benchmark 5 times, and each time the output during replay matched the original output, and all error checks passed.

We next ran a macro-benchmark to verify that ReVirt faithfully played back input from external systems and to exercise the system as a whole for longer periods. During the macro-benchmark, we booted the virtual machine, started the GNOME window manager (displaying to a remote X server), opened several interactive terminal windows, and concurrently built two applications (freeciv and mup) on a remote

Workload	Runtime with logging	Log growth rate	Replay runtime
POV-Ray	1.00	0.34 GB/day	1.01
kernel-build	1.10	0.59 GB/day	1.01
SPECweb99	1.11	7.01 GB/day	1.18
daily use	≈1	0.2 GB/day	0.03

Table 4.3: Time and space overhead of logging and replay. Logging slowdown shows the overhead caused by logging, relative to running UML without logging. Log growth rate shows the average rate of growth of the log during the workload. Replay runtime is normalized to the runtime of UML with logging. Replay runtime values less than 1 indicate that replay ran faster than logging, due to replay's ability to skip over idle time.

NFS server. The logging run of this benchmark generated 15,000,000 system calls and 55,000 virtual interrupts. At each of the 15 million system calls, the system call order, all registers, and the branch count were the same; and each of the 55,000 virtual interrupts, the eip and branch count tuple existed in the replay instruction stream. We are confident that during replay, the virtual machine took the same execution path as during logging.

For the other tests used in this paper, we disabled the extra error checking mentioned above. However, ReVirt always checks the branch count at the interrupted instruction and matches the branch count seen at that instruction during logging. This "checking" mechanism is intrinsic to ReVirt's algorithm and is "on" all the time during replay.

4.4.3 Logging and replay overhead

Next we seek to quantify the space and time overhead of logging. We do not include the time and space overhead to checkpoint the system, since we expect a checkpoint to be amortized over a long period of time (e.g. a few days). Table 4.3 shows time and space overhead for logging on the POV-Ray, kernel-build, and SPECweb99 workloads. Logs are stored in a compressed format using `gzip`.

Table 4.3 shows that the time overhead of logging is small (at most 11%). The space overhead of logging is small enough to save the logs over a long period of time at low cost. Workloads with little non-determinism (e.g. POV-Ray, kernel-build) generate very little log traffic. Note that the log data needed to replay local compilations takes much less space than the disk data generated in compilation.

The log growth rate for SPECweb99 is higher because of the need to log incoming network packets. However, it is still not prohibitive. For example, a 128 GB disk can store the volume of log traffic generated by SPECweb for over two weeks[7]

We also used ReVirt for UML as the first author's desktop machine for 24 hours to get an idea of the virtualization and logging overhead for day-to-day use[8]. We experienced no perceptible time overhead relative to running directly on the host, and the log occupied 0.2 GB after one day.

Table 3 shows that workloads typically replay at the same speed as they ran during logging. It is possible to replay a workload faster (sometimes much faster) than it ran during logging because replay skips over periods of idle time, such as those encountered while the user was thinking or reading during the daily use workload. We call this effect **idle-time compression**.

4.5 Related work

We discuss related work in general in chapter II; in this section, we discuss work related specifically to our work in ReVirt for virtual machines.

Bressoud and Schneider's work on hypervisor-based fault tolerance (generally

[7]Recent analysis of the logs reveals that less than 18% of the log content is actually network data; the rest is data due to system calls related to network data, many of which are very inefficient. Optimizing the logging of these system calls should improve the logging space overhead by at least 50%.

[8]This test was run using Linux 2.2.20 as the guest operating system, and an older version of ReVirt which used UMLinux as the virtual machine. We expect numbers to be similar

called “Hypervisor” for convenience)[15] shares several techniques with ReVirt. Bressoud and Schneider use a virtual machine for the PA-RISC architecture to interpose a software layer between the hardware and an unchanged operating system, and they log non-determinism to reconstruct state changes from a primary computer onto its backup.

While ReVirt shares several mechanisms with Hypervisor, ReVirt uses them to achieve a different and new goal. Hypervisor is intended to help tolerate faults by mirroring the state of a primary computer onto a backup. ReVirt takes some of the techniques developed for fault tolerance and applies them to provide a novel security tool. Specifically, ReVirt is intended to replay the complete, long-term execution of a computer. To illustrate the difference between these goals, compare the usefulness of checkpoints for each goal. Recovering a backup to a prior point in time can be accomplished either by checkpointing the primary’s state periodically or by logging the primary’s operations. However, checkpoints are not sufficient for intrusion analysis because they do not show how the system transitioned between checkpoints; checkpoints can only be used to initialize the replay procedure.

Besides a difference in goals, Hypervisor and ReVirt also differ in several design choices. Because Hypervisor only seeks to restore the backup to the last saved state of the primary, it discards log records after each synchronization point. In contrast, ReVirt enables replay over long periods (e.g. months) of the computer’s execution, so that it must save all log records since the last checkpoint. Another difference is that Hypervisor defers the delivery of interrupts until the end of a fixed number of instructions (called an epoch), while ReVirt delivers interrupts as soon as they occur (or when the guest kernel re-enables interrupts). Hypervisor also logs more information than ReVirt (e.g. Hypervisor logs disk reads).

There are several virtual machines similar to UML. FAUMachine [16] (originally called UMLinux) shares many of the same goals as UML. We had originally used UMLinux for our early work, in fact, because UML had, at that time, two host processes for each guest process. FAUMachine's main goal is source-code compatibility, however, at the expense of speed and security. SimOS's direct-execution mode is also similar to these systems but is targeted at an architecture that is easier to virtualize than the x86 [44]. The Xen VMM [9] is also a paravirtualized hypervisor that requires the guest operating system to be ported to run on it; but by choosing the virtualization interface carefully, were able to get overheads as low as 3% for the kernel-build test on the Xen VMM.

ReVirt shares a similar philosophy of security logging with S4 [52]. Both ReVirt and S4 add logging below the target operating system to protect the logging functionality and data from compromised applications and operating systems. ReVirt adds logging to a virtual machine, while S4 adds it to disk drives. The logging in ReVirt captures different information than the logging in S4. ReVirt enables replay of the entire computer's execution, while S4 logs and replays disk activity. ReVirt and S4 save different data to the log (ReVirt saves non-deterministic events, S4 saves disk data), so a comparison of log volume generated will depend on workload.

4.6 Conclusions

ReVirt applies virtual-machine and fault-tolerant techniques to enable a system administrator to replay the long-term, instruction-by-instruction execution of a computer system. Because a target operating system and target applications run within a virtual machine, ReVirt can replay the execution before, during, and after the intruder compromises the system. This capability is especially useful for determining and fixing the damage the intruder inflicted after compromising the system. Because ReVirt logs all non-deterministic events, it can replay non-deterministic attacks and executions, such as those that trigger race conditions. Because ReVirt can replay instruction-by-instruction sequences, it can provide arbitrarily detailed observations about what transpired on the system.

ReVirt has been successfully used by several subsequent projects to aid in post-mortem analysis. ReVirt adds reasonable time and space overhead. The overhead of virtualization ranges from imperceptible for interactive and CPU-bound applications to 13-76% overhead for kernel-intensive applications. The time overhead of logging ranges from 0-11%, and logging traffic for our workloads can be stored on a single disk for several months.

CHAPTER V

ReVirt on Multiprocessors

The previous chapter describes execution replay for single-processor virtual machines. However, multiprocessor systems are becoming more commonplace. Multiprocessor systems previously required specialized motherboards and interconnects, but new hyper-threaded and multi-core CPUs make multiprocessor systems much cheaper, and are likely to become more widely used. In order for ReVirt to be practical, we need to determine how best to use multiple processors on a logging system.

5.1 Methods of using multiprocessors

When running virtual machines on a multiprocessor, we can give the virtual machines the number of processors we want. There are two primary methods of using a multi-processor host. We can choose a single multi-processor guest, where the number of virtual processors in the guest is equal to the number of physical processors on the host. Alternately, we can partition the host into several guests. For this discussion, we will consider multiple single-processor guests, where the number of domains is equal to the number of virtual processors (see Figure 5.1).

Logging a series of single-processor virtual machine on a multiprocessor system is

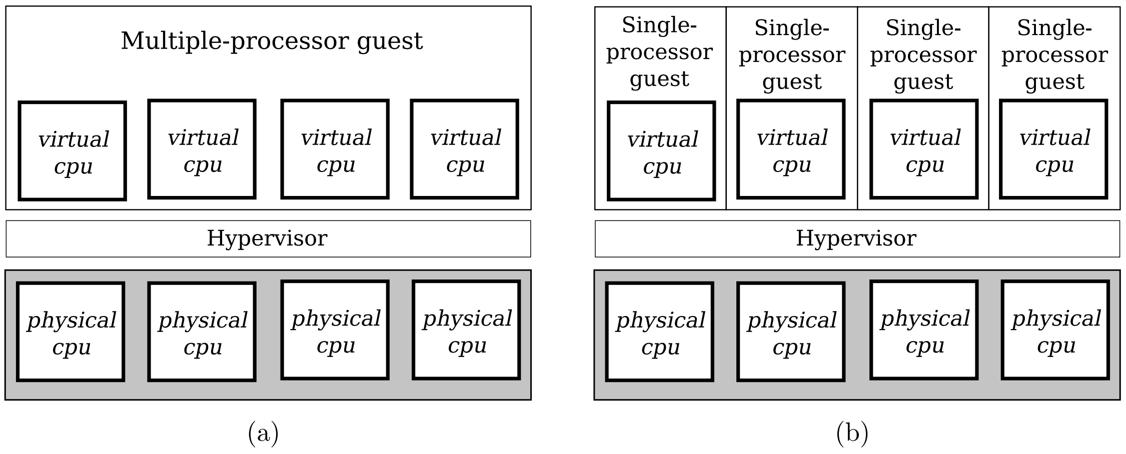


Figure 5.1: Different ways of partitioning a multiple-cpu machine

simple from a technical standpoint—it is no different than logging a single-processor virtual machine on a single-processor system. However, administration becomes more difficult. The operator must install multiple operating systems and keep them up-to-date, and deal with multiple logs and multiple IP addresses. The administration, in fact, looks very similar to administering a group of networked single-processor machines. Each virtual machine needs its own disk image and its own memory image of the kernel and the various programs to be run. The duplication of resources is another overhead.

The similarity to a group of networked single-processor machines extends to running parallel workloads. In order for a single workload to take advantage of the multiple processors, it must be set up to run over the internal virtual network of the machine. Rather than being able to share data directly from memory, data must be marshaled into messages and copied through the network stacks of multiple operating systems. This adds extra complication to the programming and management of applications and workloads. It also makes sharing more expensive than a single multiprocessor virtual machine, both in terms of time and memory.

A single, multiprocessor virtual machine is much easier to administer and use. Unfortunately, it is technically more difficult to log and replay. While running parallel workloads over groups of networked machines has been well-studied, logging and replaying an entire shared memory multiprocessor system has not been done, to our knowledge, outside of simulation.

5.2 General multiprocessor replay

The general problem to be solved when replaying shared-memory systems is that reads from memory by one process are affected by writes of another process, which may happen in any arbitrary interleaving. In order to replay a multiprocessor shared memory system correctly, an execution replay system must replay the **order** of memory accesses [1]. The observation is that if the order of memory accesses is preserved, then the results of the reads will be the same. More specifically, two instructions must be ordered if both of the following are true:

- They both access the same memory.
- One of them is a write.

We indicate that instruction a is ordered before instruction b by $a \rightarrow b$. This is read, "$a happens - before b$". In order to enforce an order between two processes, **constraints** must be introduced between them. Constraints are of the form $a \Rightarrow b$, and indicate that the replay system will ensure that b does not execute until a has executed.

Two points on the instruction stream may be ordered even if there is no direct constraint from one to the other. Within a single process, there is an implicit or-

[1]Early systems [42] attempted to log the results of all reads from shared memory, but this proved to be both too slow and to result in too much log data.

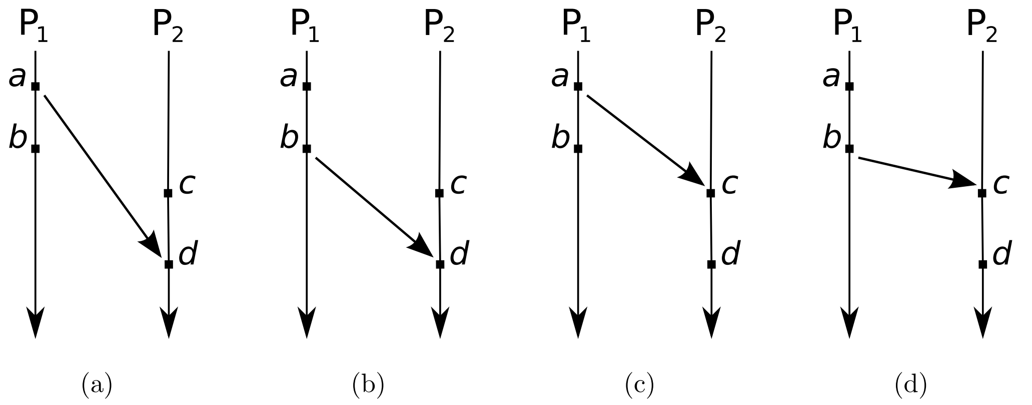


Figure 5.2: Constraints sufficient to guarantee the order $a \rightarrow d$

dering, based on the order the instructions were executed. Furthermore, ordering is transitive: $a \rightarrow b$ and $b \rightarrow c$ implies $a \rightarrow c$.

Consider Figure 5.2. Suppose that a and d are writes to the same memory, but that b and c are unrelated—only $a \rightarrow d$ is necessary. One constraint sufficient to guarantee the ordering is $a \Rightarrow d$. But any of the following constraints would imply the order $a \rightarrow b$ as well:

- $b \Rightarrow d$ (because $a \rightarrow b$ by program order)
- $a \Rightarrow c$ (because $c \rightarrow d$ by program order)
- $b \Rightarrow c$ (because $a \rightarrow b$ and $c \rightarrow d$ by program order)

If b and c are unrelated, we say that the constraints above are **over-constrained**, because they cause the replay system to run more strictly than necessary: either P_2 must wait until P_1 reaches b (although the data was ready at a), or P_2 stops and waits at c (although it is not necessary to stop and wait until d), or both. Over-constraining reduces the potential parallelism during a replay run, but can be taken advantage of to reduce the number of constraints or simplify logging.

Suppose instead that a and d are writes to one area of memory, and b and c were writes to a second area of memory. In this case, $b \Rightarrow c$ would be a necessary constraint. However, the constraint $a \Rightarrow d$ would be **redundant**, because the ordering $a \rightarrow d$ it is implied by the constraint $b \Rightarrow c$. Removing redundant constraints can decrease the log size.

A logging system for a shared memory system must generate a set of constraints that will preserve ordering between writes and other accesses, but is free to chose any set of constraints that will meet that ordering.

5.3 Detecting sharing with the CREW protocol

In order to log constraints necessary for multiprocessor replay, we implement a **concurrent-read, exclusive-write** (CREW) protocol between **virtual cpus** in a multiprocessor virtual machine. This technique for detecting constraints was first introduced by [38]. The CREW protocol stipulates that each page may be in one of the following two states:

- **concurrent-read**: All cpus may read the page, but none may write to it.
- **exclusive-write**: One cpu (called the **owner**) may read and write to the page; all other virtual cpus have no access.

The CREW protocol has the following property: if two memory instructions on different processors access the same page, and one of them is a write, there will be a CREW event between the instructions on each processor. We can take advantage of this property to generate constraints sufficient to replay the order of accesses for a given execution.

5.3.1 Implementing CREW with page protections

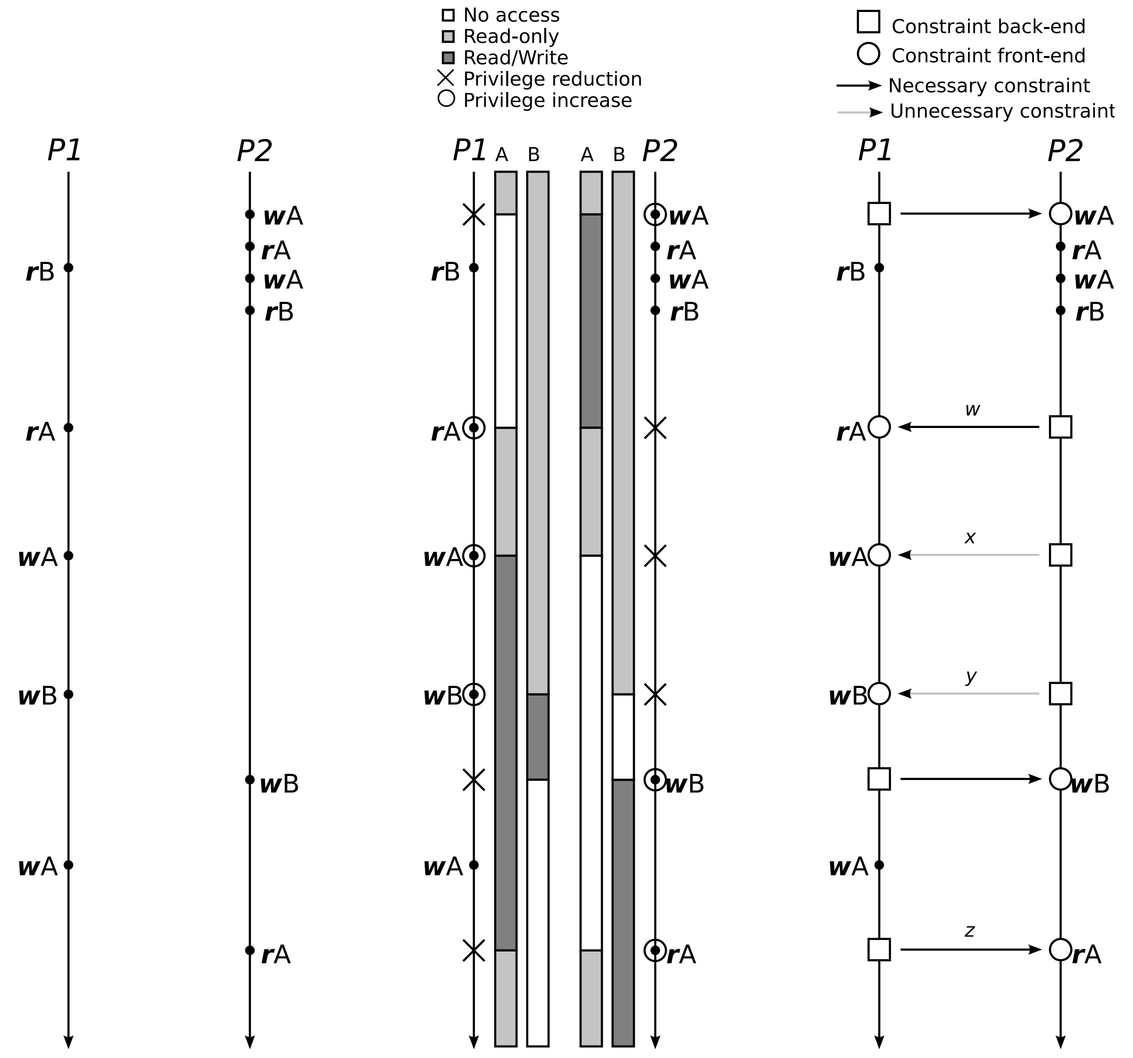
In order to enforce the access restrictions required by the CREW protocol, we use the hardware page protections available on all modern desktop processors to share memory on a page granularity. Page protections generally allow a page to be in one of three modes: read-write, read-only, and no-access[2]. If an instruction access exceeds its permission, it causes a hardware page fault and traps into the hypervisor. If the page fault handler determines that this fault was due to the CREW protocol, it will call the CREW subsystem. The CREW system in the hypervisor will then obtain more permission for the page on behalf of the virtual cpu, and then return, allowing the faulting instruction to execute.

If the page is in concurrent-read mode and a virtual cpu attempts to write to the page, the CREW system will remove read permission from all other virtual cpus and give write permission to the virtual cpu in question before continuing. Similarly, if a page is in exclusive-write mode, and a virtual cpu other than the owner tries to read the page, the CREW protocol will remove write permissions from the owner, put the page in concurrent-read mode, and give read permission to the faulting virtual cpu. If a page is in exclusive-write mode, and a virtual cpu other than the owner tries to write to the page, the faulting cpu is given ownership. All read and write permission is taken away from the previous owner and given to the new owner.

Using page protections allows us to interpose on reads and writes without modifying software running in the guest.

Figure 5.3 shows an example of this protocol. In (a), we have a particular run, accessing two shared pages, A and B. Time progresses downward, and the results of reads and writes follow the time-line (i.e., read instructions read the value of the last

[2]For simplicity, we do not consider execute bits, or the potential for write-only protection.



(a) An example memory interleaving for two pages, A and B.

(b) Processor permissions with the CREW protocol

(c) Constraints generated by logging the CREW protocol

Figure 5.3: An example of logging with the CREW protocol

write).

Figure 5.3 (b) shows the same run with the CREW protocol. The shaded bars between the two process execution lines indicate the CREW permissions of each virtual cpu to each page. A circle on the execution line indicates a CREW fault at that event, and subsequent privilege increase; an 'X' indicates a loss of permissions.

5.3.2 Replaying constraints

Constraints between two virtual processors A and B must be defined between individual instructions, a_m and b_n. (For the convenience of the following discussion, when we have a constraint $b_n \rightarrow a_m$, we will call a_m the **front-end** of the constraint, and b_n the **back-end**.) When virtual processor A reaches the front-end instruction a_m, it cannot proceed until processor B reaches the back-end instruction b_n. If there are more than two processors in the system, there may be several constraints associated with a_n. We consolidate these by defining the constraint as a vector: $< b_n, c_p, d_o, ... > \rightarrow a_m$.

To specify which instruction, we use the **instruction count** at the instruction in question[3]. The instruction count is a register on the physical cpu; other cpus do not have direct access to read it. The constraint replay system keeps a vector of instruction counts per virtual cpu. In order for the cpu at the front-end of the constraint to know that it can continue, each other cpu must update its instruction count in the global vector when it reaches the back-end instruction.

Each virtual cpu will have its own log of events. Out of this log will be played all of the normal ReVirt logs—interrupts, `rdtsc` values, and so on. In order to implement constraints efficiently, we define two types of log events: **constraint** events, and

[3]Here we discuss an abstract notion instruction count, as a unique label for a given instruction, not necessarily one individual counter.

back-end events.

Constraint events are the front-end of the constraint. They contain the instruction count of the instruction that needs to wait, and a vector of the instruction count that must be passed by the other processors before the front-end instruction can continue. When a constraint event is delivered during replay, the replay system will compare the vector in the log with the current global vector. If all elements of the global vector are greater than or equal to the corresponding elements in the log vector, the system allows execution to continue. If any of the counts in the global vector is less than the corresponding count in the log, it records the vector that it is waiting for and pauses.

When a back-end event is delivered during replay, the replay system will update the global vector with the processor's instruction count. It will also check to see if any cpus are waiting that need to be woken up.

5.3.3 Instrumenting CREW

The simplest approach to generate the constraint logs is as follows. We keep a global instruction count vector, as during replay. When we reduce a virtual cpu's privileges, we copy its instruction count into the global vector and insert a back-end event into its log. When a cpu gains privileges, we generate a constraint event using the global instruction count vector as the constraint vector. This is correct because the CREW protocol always reduces privileges before granting more privileges. Figure 5.3 (c) shows the constraints generated by this technique.

Note, however, the redundant constraints x and y. Netzer developed an algorithm that will guarantee the minimum number of constraints required for correctness.[41]. Briefly, the algorithm keeps a **last-accessed vector** containing the last instruction that each virtual cpu accessed the object. Each cpu also keeps a last-constraint vector, containing the instruction count of the back-end instruction of the last effective constraint with each other cpu. When a processor A gains privileges for an object, before adding a constraint to a processor B, the system checks the instruction count of last time B accessed the object. If the B's last access to the page was before A's constraint to B, then this constraint will be redundant. Furthermore, if processor A does add a constraint to B, it scans B's last-constraint vector for more recent constraints of other cpus. Thus, suppose processor A's last constraint to processor C was at c_1. Then it adds a constraint to processor B. If processor B's last constraint to C was at c_5, processor A is now effectively constrained, by transitivity, to c_5 as well. So processor A can put c_5 in its last-constraint vector for processor C, to prune out unnecessary constraints.

This algorithm will provably report the minimum number of constraints. However, it requires knowing the last time that a given page was accessed. Because the CREW protocol is not involved on every read or write, it does not have access to this information. To see why this is so, consider Figure 5.3(c). Constraints x and y are unnecessary because pages A and B were not accessed after constraint w. Constraint z is necessary, because page A was accessed after constraint y. The CREW protocol cannot distinguish z from x and y because it does not know the last access time.

5.3.4 Replaying with CREW enabled

We have described how to log and replay with abstract constraints and with CREW disabled during replay. Another possibility is to use the CREW protocol during replay instead of replaying constraints, replaying the timing of permission-reduction messages instead. The observation is that if the privilege reduction events are made to occur the same way during the replay run as during the logging run, the CREW faults to request an increase privilege will occur in the same way as well.

In this scenario, the system logs a privilege reduction event on a virtual cpu as an asynchronous event, storing the timestamp, the page for which privilege was reduced, and the virtual cpu that requested it. Privilege increase is not logged. During replay these privilege reduction events are replayed at the same point in a virtual cpu's execution. The system replays a privilege reduction event to a virtual cpu, it reduces privileges on that cpu and sends a message to the virtual cpu that requested the privileges during logging. During replay, that processor will fault on the same instruction during replay as it did during logging. When it does, instead of sending a privilege-reduction request, it waits for the message sent after the request is replayed from the log to the other processors. If the message from the replayed privilege reduction event is sent before the processor is waiting, it is put into a message queue to be delivered when it is required.

This method has some interesting properties. For one, the system does not log both sides of a constraint; it only logs the "privilege reduction" or back-end. On average, we would expect this to cause about half as many log entries as the constraint method described above. However, using this method adds to replay the complexity and overhead of maintaining the CREW protocol, along with replaying the virtual interrupts, device inputs, etc. Furthermore, it requires that the CREW protocol be replayed exactly; no rearrangements are possible. This would preclude the possibility of performing any kind of constraint optimizations. For these reasons, we chose to implement abstract constraints.

5.3.5 Direct Memory Access

Modern hardware systems allow physical devices to write directly to main memory, without involving the processor. This is called **direct memory access** (DMA). DMA eliminates the overhead of the processor copying data from the device to memory.

Replaying DMA presents some difficulties. In DMA, a device acts as another processor with respect to memory transactions. A single-processor system with DMA-enabled devices is effectively a multiprocessor system from replay's perspective. However, unlike peer processors in an SMP system, the devices do not have page protections that we can use to interpose on accesses[4]. How are we to involve devices in the CREW protocol?

The key observation is that DMA devices are not generally self-motivated peers. They only write to memory in response to a request from a cpu. Requests typically follow a **transaction** model, where a cpu will specify an operation and an area of memory. The device will write to the memory during the operation, and inform the cpu when the operation is completed. After the transaction is finished, the device will not write to the memory again. While this transaction is taking place, it is generally not correct for the cpu to access the memory assigned to the device to do DMA.

If the device follows this type of transaction model , where the cpu does not access memory to the device until a transaction is completed, and if the hypervisor can interpose and understand the commands from the guest to the device and the

[4]Some new systems include an IO-MMU, for controlling DMA access to memory. However, these systems are designed to prevent buggy drivers and devices from corrupting system state, and do not necessarily provide ways to continue an interrupted operation after a fault. Re-executing a faulting operation is one of the fundamental techniques we use.

device's responses, we can model the device as a **non-preemptible actor** in the CREW protocol. A non-preemptible actor does explicit grab and release of pages before and after a transaction, rather than grabbing them on demand and having them preempted, as preemptible actors such as virtual cpus do. When a cpu issues a DMA command to the device, the hypervisor informs the CREW protocol, which acquires the appropriate privileges on behalf of the device (either concurrent-read or exclusive-write, depending on the transaction). When the device informs the cpu that the transaction is done, the hypervisor informs the CREW protocol, which will release access on behalf of the device. If any cpu tries to access a page in a way that is incompatible with the CREW privileges of some device on a system, it must block until the device has finished the transaction associated with that page.

Non-preemptible actors cannot be interrupted at an instruction granularity. So instead of an instruction count, we represent points for front- and back-ends of constraints with a **crew event count**. This count is incremented at the beginning and end of each transaction. The beginning of the transaction corresponds to the constraint, and the end corresponds to a loss of privilege.

During replay, the constraint replay system must ensure that the DMA is replayed at the proper time with respect to the other processors. If we do not use the device during replay, we must log the data from the DMA during logging in order to replay it from the log during replay; otherwise, we must ensure that the device does the DMA properly.

5.4 Design discussion

5.4.1 Alternate ways to detect sharing

Any technique that can be used to implement a distributed shared memory could be used to detect sharing and implement multiprocessor execution replay. LeBlanc[38] instrumented reads and writes to shared memory to implement a CREW protocol over shared objects. Techniques such as compiler instrumentation or binary rewriting[47] could also be used.

Detecting sharing with software instrumentation has some advantages over detecting with page protections. The main advantage is that data can be shared at an object level granularity, or smaller. Shasta[47] could flexibly share on several levels of granularity. This would reduce the amount of false sharing in the system.

One possible disadvantage is that the checks are done explicitly in software, rater than in hardware. Overheads include extra instructions before each access, and the resulting less effectively available code and data cache space, even in cases where there is no sharing. Using hardware page protections avoids the extra overhead of software checks when a cpu has access.

The other disadvantage of software checks is the security aspect. An attacker can introduce arbitrary code, so we cannot use anything which requires compiler instrumentation or a particular programming disciplines. Dynamic binary rewriting could probably be made to work. However, then we would need to work correctly for self-modifying and self-inspecting code. There is also the extra complexity of making the instrumentation SMP-friendly.

5.4.2 Cache consistency models

Cache-coherent multiprocessors can have different consistency models. The most basic is **sequential** consistency, where the reads and writes are guaranteed to have a global ordering. The `x86` processors we used for ReVirt have a **processor** consistency: each process' stores are observed in program order, but the observed interleaving of two processors might not be the same on other processors. Other possible consistency models include various forms of **release** consistency, in which there are no guarantees without explicit synchronization operations such as memory barriers. See [3] for a summary on consistency models.

The CREW protocol actually enforces a **sequential** memory consistency, even on processes with a weaker underlying consistency. On systems whose underlying cache coherence is a weaker consistency model such as **release** consistency, the CREW protocol would need to do a memory barrier along with a TLB flush when taking away write permission in order to ensure that all writes were seen by future readers and owners of the page.

5.5 The Xen Hypervisor

Similar to User-Mode Linux (UML) from the previous chapter, the Xen hypervisor requires **paravirtualized** guests: that is, the guest kernel is modified to use the interface provided by the hypervisor for privileged instructions, rather than the hypervisor emulating the interface of native hardware[5]. Unlike UML, which is constrained to using the interface provided by the Linux kernel, the Xen paravirtualized interface was designed from the ground up with speed of paravirtualization in mind.

[5]Modern versions of Xen are able to take advantage of hardware virtualization technologies, such as Intel's VMX and AMD's AMD-V, to run unmodified guests as well. This hardware was publicly unavailable the time our research was conducted. The basic principles still apply.

This allows Xen to run kernel-intensive benchmarks (usually worst case for virtual machines) at speeds near native[9].

5.5.1 Xen “hardware” interface

Xen guests use **hypercalls** to perform privileged operations. These operations include memory operations, setting trap tables and interrupt gates, accessing debugging hardware, switching stack pointers and segment registers, and so on.

Xen uses a shared page for passing certain kinds of information between the guest and Xen; this is called the **shared info** page. Some of this is information that requires privileges to access directly from the hardware, and may be read often by the guest: for instance, the speed of the CPU, what kinds of memory and hardware functionality are available, and the system time. Because domains have direct access to the `rdtsc` instruction, there is also a “timestamp counter offset” value, which is updated when a domain is scheduled in or out to help the domain compensate for scheduling when using the timestamp counter. There is also a per-virtual-cpu area of the shared info structure that contains information related to each virtual cpu.

Virtual interrupts in Xen are called **events**. Each domain has 1024 **event channels**. Each event channel is assigned to a specific virtual cpu. Events are always sent through the hypervisor, either from hardware-based virtual devices (such as a virtual timer interrupt) or from virtual devices in other domains.

An event channel is marked as **pending** in a bit array in the shared info page. There is also a bit array for masking event channels, and a per-virtual-cpu event pending and event mask flag. These are used to implement once-only virtual interrupt delivery, as well as interrupt disabling.

One of the common techniques of Xen is to make the interface to the hypervisor the same as the interface to the hardware. This both minimizes the changes necessary to paravirtualize an operating system and allows Xen to use the hardware to implement its interface as much as possible.

One example of this technique is Xen's interrupt and trap frame stacks. When an event is delivered, the frame put on the stack is very similar to that used by a hardware interrupt. This allows Xen to re-use interrupt handling code inside of guest operating systems. It also allows Xen to let some traps go directly to the guest kernel without involving the hypervisor. The system call trap (`int 0x80` on x86) is set in the hardware to go directly to the guest system call handler. When a guest process makes a system call, it goes directly to the guest kernel without involving the hypervisor at all.

Another example of this technique is Xen's pagetable interface. Xen gives guest domains direct access to the hardware pagetables. This allows guests to walk the pagetables and do other memory operations without keeping two copies of the pagetables, and gives them direct access to hardware information, like the dirty and accessed bits[9].

5.5.2 Domains, drivers, and the shared ring idiom

Xen calls a single running instance of a guest virtual machine a **domain**. The most important of the domains is the privileged domain, also known as **domain 0**. Domain 0 is automatically started when Xen boots. It contains the drivers to all of the devices on the system[6], and runs the software that manages other domains, which allows the Xen hypervisor to remain a thin layer of code between the guests and the hardware, rather than becoming a large and complex piece of machinery like

[6]There are plans in Xen to create **driver domains**, which will allow certain drivers to run in their own domains for better fault isolation. This functionality has not yet been implemented for production use.

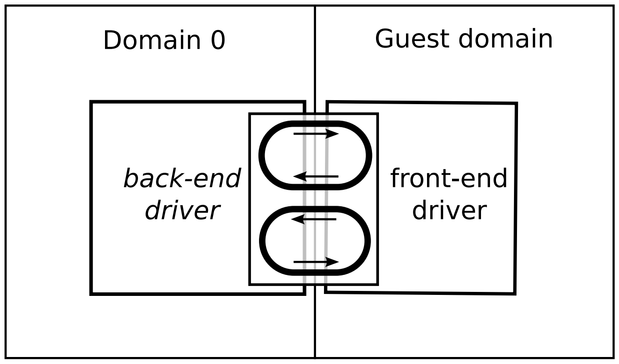


Figure 5.4: Front-end and back-end communicating over shared rings

a full kernel.

Unprivileged domains do not have direct access to hardware devices. Instead, they use virtual devices provided by domain 0. These paravirtualized devices are broken down into two halves: a **front end**, which resides in the unprivileged domain, and a **back end**, which resides in domain 0. The front end marshals requests to the virtual device from the unprivileged domain and sends them to the back end. The back end then satisfies these requests through domain 0's kernel and its access to physical devices.

The two halves communicate over a pair of **shared rings**(see Figure 5.4). Shared rings are two circular buffers on a single physical page, one for requests from the front end to the back end, and one from requests from the back end to the front end. The actual memory is shared between the two drivers, to reduce copying overhead and maximize batching and interleaving.

There are three general types of paravirtualized drivers which concern us: the control ring, block drivers (i.e. hard disks), and network drivers.

The control ring is implicit to every paravirtualized guest, and is started on boot. This ring implements the virtual console device. It also serves as the interface to bring up all other virtual devices, and any of the other domain management features implemented in the domain 0 management software.

The paravirtualized block device acts much like a simple, idealized disk device. The front end can request information about the block device, and make asynchronous read and write requests to memory.

The paravirtualized network device is more subtle. The physical network device has no way of knowing for which domain an incoming packet is destined. Rather than copying incoming packets, the paravirtualized network drivers use a **page-flipping** technique. When a domain wants to receive packets, it gives up a certain number of physical page frames to the back-end driver. These frames are added into a pool of pages into which the hardware network device writes packets as they come in. The back-end driver then looks at the frame containing the packet and and gives the frame to the appropriate domain. The frame the domain receives back may or may not be one of the frames that it gave up. Domains can never receive back more frames than they have given up. If a packet comes for a domain which has no outstanding donated frames, the frame containing the packet is merely put back into the pool (and the packet effectively dropped). In this way, a single network device can be safely multiplexed over several virtual network devices without the overhead of copying.

5.6 Replaying Xen

The hardware interface of Xen has few surprises, in regards to non-determinism. The results of hypercalls are deterministic; these results do not need to be logged and replayed. The results of `rdtsc` instructions must be logged and restored, and the timing of virtual interrupt deliveries must be logged as an asynchronous event. Updates to the shared info page are visible to the guest as well. These must be logged and replayed as asynchronous events.

5.6.1 Shadow pagetables

As we mentioned in section 5.5.1, Xen gives a guest direct access to the pagetables. This exposes the underlying machine frames to the guest. Unfortunately, these underlying machine frames change every time a guest is run[7].

Furthermore, for guests with multiple virtual cpus, the dirty and accessed bits are written directly to the physical frame by the hardware when a virtual address that they map accessed. Because this write does not go through the pagetables, it is not possible to detect when it changes in order to detect and enforce an ordering between this write and subsequent reads of those bits by other processors.

And finally, during logging on an SMP, each virtual cpu needs a different set of hardware page protections to implement the CREW protocol. But this cannot be done if SMP guests share pagetables across virtual cpus. Furthermore, these protections will be different during logging than during replay.

For these reasons, we use a feature of Xen called **shadow pagetables** for replaying guests[8]. The guest's pagetables are virtualized and not used directly by the MMU. Instead, the hypervisor creates copies, or **shadows**, of the guest's pagetables, which are used by the actual hardware. By introducing this level of indirection we lose some performance, but we gain an abstraction that makes it much easier to work with. The guest will now see the same frame list each time; these frames will be translated into the actual underlying machine frames by the hypervisor. Dirty and accessed bits must be emulated, so that we can be certain that the writes to them

[7] In fact the frames are purposely made as random as possible to ensure that coders do not accidentally make assumptions about them.

[8] As noted before, during normal operation, the hardware pagetables are exposed directly to the guest. There are times when a shadow pagetables are turned on however, for features such as live migration. See [19] for more information on live migration.

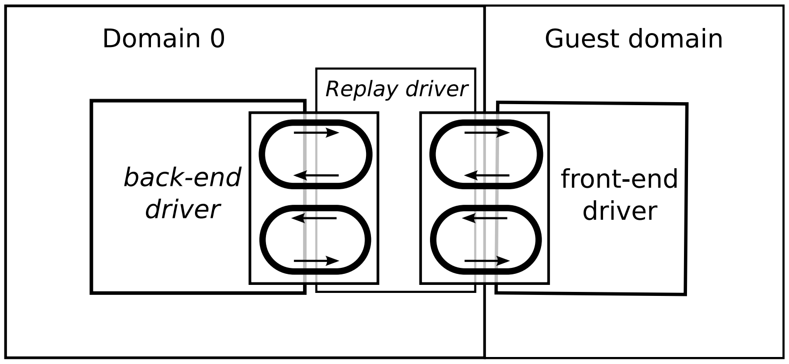


Figure 5.5: Using a replay driver to interpose on shared rings

are ordered properly in the CREW protocol. Each virtual cpu on a multi-virtual-cpu guest domain can have its own shadow of a shared pagetable, and the reduction in protections due to the CREW protocol is not visible to the guest.

5.6.2 Paravirtualized devices and CREW

The difficulty in replaying Xen domains comes with the paravirtualized devices. Devices share memory directly with other domains which are not being logged or replayed. This is, in fact, the Xen analog of DMA.

This turned out to be an advantage during the development of the CREW protocol. We model the back-end drivers as non-preemptible actors and integrate them into the CREW protocol. Because the interactions between back-end drivers and a single-virtual-cpu domain are limited in scope, it allowed us to develop and test the basic CREW protocol functionality without having to address many of the sticky issues that we faced when dealing with CREW between two virtual cpus.

In an attempt to minimize the modifications to the system, we developed the concept of a **replay driver** (see Figure 5.5). A replay driver is a process that sits between the front-end and back-end. The idea is that the replay driver knows about the CREW protocol, and how to log and replay, as well as the requirements of the disk driver. It has two sets of shared rings—one set with the front-end driver, and one with the back-end driver. It reads requests and responses from both rings, interprets them, and if appropriate passes them to the other ring (possibly after taking some action).

Before reading or writing requests from the ring with the front-end, the replay driver grabs non-preemptible write permission permission[9] on the page. It also grabs and releases other guest pages, based on its knowledge of the protocol.

Because the replay driver itself and the rest of the virtual devices in domain 0 are not being replayed precisely, the only concept of "time" is with respect to crew events relative to the guest. The only thing being replayed is the data, and the CREW protocol detects when timing is important. Because of this, logs are stored with crew vectors, rather than timestamps. The replay drivers are as active in the CREW protocol during replay as during logging: they not only wait until certain vectors have been reached before doing certain actions, they also inform the constraint replay system when they have reached certain CREW event counts, so that other CREW actors can safely proceed.

There are three different ways to treat devices during replay. Some devices, such as the network, are not necessary during replay; only the results of the device from the log need to be replayed. Other devices, such as the console, are not strictly necessary to use during replay, but it is useful to be able to see the console output and have the normal integration with the tools. And still other devices, such as the virtual hard disk, we want to use during replay, because it is too expensive to log and replay the data.

The networking driver has not been implemented yet, but we predict that it would be sufficient to log timing of reads from the ring, as well as data and timing

[9]Reading a request from the ring involves updating ring indexes within the page.

for writes to the ring. Xen enforces that a guest must remove all references to a page before passing it to the back end, so we need do nothing when giving up a page. When a packet comes in, we can simply log the packet, and the timing of when the page is flipped back to the guest. During replay, we can simply copy packet data from the log at the appropriate time.

For the console driver, it is sufficient to log the timing of reading requests from the front end, and both the timing and the value of responses written to the front end. During replay, requests from the front end are read at the CREW event vector that they were during logging, and forwarded on to the back end. Responses and requests from the back end generating during replay are dropped; instead, responses and requests are read from a log and delivered after the CREW vector logged.

When logging the block device, we do not log the data read from the disk; instead, we restore the disk to its original state, and re-execute both read and write requests during replay. As described in section 5.3.5 the replay driver needs to co-ordinate the back-end's accesses to guest memory with the replay, to ensure that both reads and writes happen appropriately. During logging, each request to the back-end is associated with a constraint. During replay, a request is not passed to the back-end until the constraint has been met. The replay driver then waits for the transaction to be completed before increasing its CREW event count.

Most disk controllers (including Xen's paravirtualized back-end) can re-order outstanding reads, so that they finish in a different order than requested. Because we use the disk device during replay, we have the possibility that the order that reads are completed during replay may differ than the order they are completed during logging. The replay driver must therefore re-order these requests during replay to match the order seen by the guest during logging. This may involve delaying request-completion messages being forwarded to the guest until other requests are completed. The CREW protocol will guarantee that this re-ordering will not be seen by the guest.

5.6.3 Multiprocessor CREW on Xen

Implementing the basic CREW protocol for Xen is straightforward. There are a finite number of **actors**. Each virtual cpu is a preemptible actor, and each device is a non-preemptible actor. (See section 5.3.5.) For each actor, we keep the following information:

- A CREW event count. This replaces the instruction count in the constraint vector for preemptible actors.
- If it is waiting, and what it is waiting for (either a guest page during logging, or a crew constraint vector during replay).

For each guest page, we keep the following information:

- The current owner (if in exclusive-write mode) or nobody (if in read-only mode)
- A count of the non-preemptible actors that have read access to this page

During logging, when an actor requests more permission than it has for a guest page (either due to a virtual cpu faulting or a replay driver grabbing permission), the CREW protocol will look at the state. If the request can be fulfilled by removing permissions from virtual cpus this permission is removed (increasing CREW event counts and logging if appropriate), the owners updated (if appropriate), and a constraint for the requesting actor logged (if necessary). If the request cannot be fulfilled because some non-preemptible actors are holding permissions needed, then the actor must wait for the page.

When a non-preemptible actor releases permissions for a page, any waiting actors are woken up. They then contend for the page normally.

Removing permissions and adding to the log of a virtual cpu running on another processor requires some care. The log must be taken on the other processor, because certain information (such as the registers and hardware performance counters) are only directly available on that physical cpu while it is running. Furthermore, the TLB must be flushed there as well, to make sure that modifications made to the pagetables are actually reflected in the TLB of the processor.

The most simple option is to send an **inter-processor interrupt** (IPI) to the other processor, and have the other processor do everything: remove permissions, flush the TLB, and take the log. However, removing permissions can be a long process. While this is happening, the processor which is being preempted is busy removing permissions, and the one trying to gain permissions is waiting for it to be done.

However, we can improve this process with a clever trick. The shadow pagetables in Xen are protected by one lock per domain, called the **shadow lock**. This lock is called at the beginning of handling a shadow page fault (which is a super-set of CREW fault handling), and held until the fault is done. We also grab the shadow lock whenever we are doing any CREW action. This enables us to have the one processor which is requesting more permissions do the removal of permission from the pagetables, sending the IPI only to make the log and do the TLB flush. This allows the other processor to continue running while the brute-force search is going on, as long as it doesn't access the page being removed. If it does access the page, it will spin waiting for the shadow lock until the entire operation is completed. In either case, the point the log is taken is after the last access could possibly have happened.

5.6.4 Hypervisor accesses to guest state

The page protections arbitrate accesses by the guest virtual cpus, and replay drivers integrate the devices into the CREW protocol, but the hypervisor accesses guest state as well. This access is deterministic and generally synchronous with respect to a single virtual cpu; but must be considered when dealing with multiple virtual cpus.

The hypervisor reads and writes guest-visible memory in the following places:

- Hypercalls
- The shadow code, reading guest page table entries and writing dirty and accessed bits.
- Setting an event channel "pending". This is a series of reads and writes to the shared info page.
- Event check. Event delivery code first reads the shared info page to see if an event is pending for a virtual cpu, and also if interrupts are blocked.
- Event delivery. If the event check determines that an event needs to be delivered, the hypervisor writes the interrupt frame on to the guest stack.
- Updates to TSC offset, domain time, etc. write to the shared info page.

The first task is to integrate all of these accesses with the CREW protocol. Hypercalls account for the vast majority of unique references to guest state from the hypervisor. Fortunately, the hypercalls were designed to be able to handle access to guest state through the guest's page tables. Because of this, the normal CREW faults will happen, and are handled in the appropriate way, without any modifications to the hypercalls themselves.

For other accesses, we instrument the code to grab the page required before access. In order to be certain that there is not a race between the grab of the page and the access within the hypervisor, we require that the shadow lock be held continuously from the time the page is grabbed until the access to the page is complete. Since the shadow lock must be acquired to gain permission, this guarantees that no other cpu can remove the code's permission before its operation is complete.

The second task is to deal with potential races within the hypervisor, and between hypervisor access and guest access. Consider the following potential races:

- A hypercall on one virtual cpu accesses page A, then page B; a hypercall on another cpu, executing concurrently, accesses page B then page A.
- A hypercall writes to page A and B. Between the two writes, the other virtual cpu, in guest mode, reads the modified page A, and the unmodified page B.
- A hypercall writes to a guest pagetable. Concurrently, the shadow code of another virtual cpu is reading the guest pagetable to generate a new shadow entry, which is about to be used.

Unfortunately, it is difficult to log and replay asynchronous events (including constraints) within the hypervisor. The main reason is that although ReVirt guarantees that execution in guest rings 1-3 are identical during logging and replay, the execution in the hypervisor is, by definition, different between logging and replay. This causes several difficulties. First, the hardware counters we use to replay asynchronous events are configured to count only events in rings 1-3. If we were to try to use them in ring 0, the counters would be changing as we tried to read and act based on them. Secondly, although the results of the hypercalls are deterministic, the exact path taken is not guaranteed to be deterministic. Determining where to deliver the interrupt using instruction counters becomes impossible in this case.

If all accesses to guest memory by hypercalls happen through routines such as `copy_to_user()` and `copy_from_user()`, and all accesses happen in exactly the same order during logging and replay, we could replay the order that these reads and writes happen. This is feasible, but results in extra complication in the logging and replay system.

Another reason it is difficult to replay constraints in a hypervisor is that, unlike the Linux kernel, the hypervisor has no per-vcpu stack; if it calls schedule(), it loses all its context. So the only acceptable form of blocking while waiting for a constraint to be satisfied is to spin, an action which could lead to deadlocks. Consider a hypercall, holding a lock, which is spinning waiting for another processor to reach a certain point in its execution. Suppose that between where it is now and that point, that other processor needs to acquire and release the lock that the hypercall is now holding. Here we have a classic hold-and-wait deadlock scenario. If the acquisition of locks is the same during logging and replay (a condition which may not be true), we could solve this problem by logging and replaying the locking order. In our example, the hypercall would not be allowed to grab the lock until the other hypercall had finished using it; this would guarantee that holding and waiting would not cause a deadlock as we have described.

All of these techniques are possible, but add a lot of complication to an already complicated system, with the only gain being more parallelism between hypercalls. What we would like is to avoid hypervisor races entirely, treating all hypervisor operations as atomic operations, so that if a hypervisor operation (such as a hypercall) accesses the same memory as an instruction or another hypervisor operation (and one of them is a write), the one will happen either entirely before or entirely after the other.

In order to implement atomicity, we use a global lock (called the "hypervisor lock") that allows only one virtual cpu in a given domain inside a hypercall or fault handler at a time. When entering a hypercall or shadow fault handler, the hypervisor tries to grab the domain lock on behalf of the virtual cpu. If it successful, it continues; if not, it waits until the lock is available before continuing. While a virtual cpu is in a hypercall or hypervisor fault handler, we also delay interrupt-driven changes like interrupt pending and TSC offset updates.

The hypervisor lock allows us to treat a hypercall as an atomic unit for ordering purposes. Any constraints generated during the hypercall are pushed logically to the beginning of the hypercall. Any CREW events will be pushed until after the end of the hypercall. (These modifications are consistent with the rules for over-constraining, discussed in section 5.2.)

The hypervisor lock solves both hypercall-hypercall races and hypercall-guest races. Consider the hypercall-guest race mentioned in the list above. Before the virtual cpu can read the modified version of page A, it must get read access; but to get read access, it has to complete a shadow fault. The fault cannot execute until after the hypercall has completed. So any virtual cpu reading A or B will either see the state before the hypercall or the state after, but not in the middle.

The hypervisor lock reduces the potential parallelism if two hypercalls happen at the same time. This is an unusual case. Furthermore, in the subsequent runs, the vast majority of the time spent waiting for the hypervisor lock is actually done by the shadow fault handler. The Xen shadow fault handler already grabs another per-domain lock, the **shadow lock**. So in our code, the hypervisor lock replaces most of the shadow lock contention.

As we shall see, lock contention due to the page fault handler is a major source of overhead, especially as we increase the number of virtual cpus. The only way to reduce it, other than reducing the number of CREW faults overall, is to enable finer-grained locking. This involves a lot more subtle thinking, and would require a major revision of the Xen shadow pagetable code.

5.7 Evaluation of multiprocessor ReVirt

5.7.1 Workloads

To evaluate how well traditional SMP workloads run under ReVirt, we ran the SPLASH2 benchmark suite from Stanford University[54]. This is a well-studied suite of computationally intensive parallel applications designed to evaluate the design of parallel processors. Most of the tests have parameters or input values that can be set. The test comes with a set of default parameters and input; however, it was tuned to state-of-the-art processors of over ten years ago. Using those parameters on modern processor, most applications finished in a small fraction of a second. This is not enough time to distinguish the actual workload from start-up effects. We chose input parameters such that the tests ran for around 60 seconds. The tests we ran were FMM, LU, ocean, radiosity, radix, and water-spatial[10].

We also ran two more server-oriented workloads. They are as follows:

- **kernel-build**: parallel build of the Linux kernel. This is a make of the stock Linux kernel with the default configuration. We use `gcc` version 3.4.5, and Linux kernel version 2.6.17. In order to make this a parallel workload we used

[10]We were unable to find input for some workloads to run longer than "perceptibly instantaneous". We do not present results for those tests here.

the `-j` option of `make`, which tells `make` how many outstanding child processes to try to keep at one time (maintaining build dependencies). Our experience indicates that `-j 3` produces the optimum throughput on a domain with two virtual cpus. Two concurrent processes do not fill up the cpus, and using more than three does not give any benefit.

- **dbench**: a filesystem benchmark for Linux that's meant to emulate a workload that a Linux Samba server might generate under the NetBench Windows file server benchmark. As one might expect, the workload is almost entirely in the kernel.

5.7.2 Predicted results

Based on our knowledge of the workloads, we should be able to predict aspects of their results. The execution of these workloads can be divided into two levels: process-level and kernel-level. For most of the tests, the kernel plays only a supporting role. The exception is dbench, where the kernel is being tested, and the process-level workload is only intended to generate the kernel-level workload.

To understand what to expect from a workload, it is important to understand the properties of both levels: first, what are the sharing characteristics of the processes; and secondly, how much does the workload involves the kernel and what are the sharing characteristics of the kernel.

Most of the SPLASH2 benchmarks have very little kernel interaction. Because their sharing properties have been studied in detail, we should be able to understand how SMP-ReVirt affects each of them.

Woo et al. [54] did a comprehensive study of the SPLASH2 benchmarks on idealized hardware to learn how different parameters affected their runtime. These parameters include concurrency, working set size, communication to computation ratio, and spatial locality. We do not change the hardware parameters (cache size, bandwidth, and so on). We do change two parameters. The first is the granularity of sharing, which goes from a 16-byte cacheline to the 4096-byte page size [11]. The second is the increased latency from a miss. Our early tests indicated that a remote cacheline miss on our hardware was around 400 cycles; the average time for a CREW fault is over 40,000 cycles. Therefore we can expect that workloads that are prone to false sharing for large cacheline sizes will have unnaturally high amounts of communication, causing performance to suffer. We also expect that workloads with a naturally high communication rate regardless of cacheline size will suffer from the increased latency.

Woo et al listed ocean and LU as tests that have very regular access patterns, and are not generally subject to false sharing at higher cachelines. FMM and water-spatial are listed as workloads which can be prone to false sharing, depending on how well data structures fit in cache lines. Radix and radiosity are listed as workloads that have very random aspects of data accesses, and can be very prone to false sharing for larger cacheline sizes.

The Linux kernel is a parallel application, and has been finely tuned to the architectural parameters of the `x86`. If the kernel is sensitive to high latency and large-granularity sharing, any workload that involves the kernel will suffer. Kernel-build contains a mix of unshared process-level computation and kernel interaction. Dbench is almost entirely a kernel workload. If kernel sharing is expensive, we expect

[11]There are two effects of increasing cacheline size. The first effect is an increase in false sharing. The second is that in data that has a high spatial locality, increased cacheline size can reduce cache misses by "pre-fetching" data. SMP-ReVirt is subject to the first effect, because we increase the granularity of sharing, but is not subject to the second effect, because each cacheline must still be fetched by the processor individually.

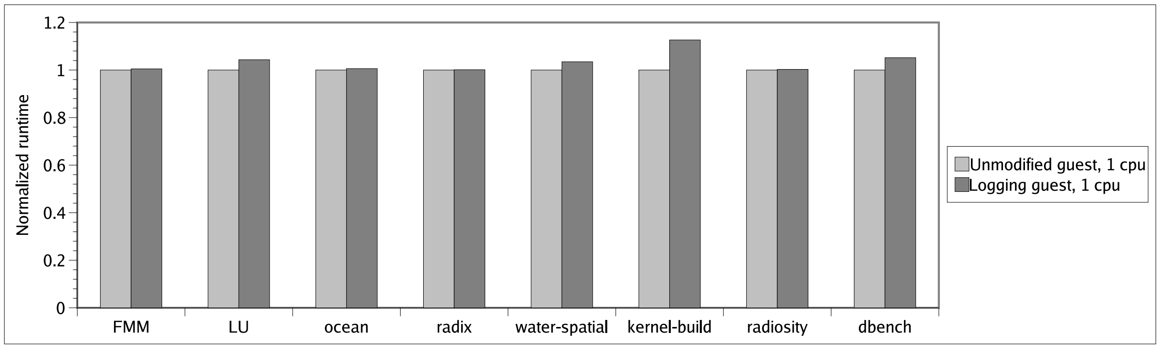


Figure 5.6: Overhead of ReVirt for a single-processor Xen guest

Workload	Logging rate, compressed	Time to fill a 300GB disk
FMM	.234 GB/day	1280 days
LU	.238 GB/day	1261 days
ocean	.232 GB/day	1295 days
radix	.292 GB/day	1025 days
water-spatial	.231 GB/day	1296 days
kernel-build	.562 GB/day	534 days
radiosity	.232 GB/day	1295 days
dbench	.557 GB/day	1280 days

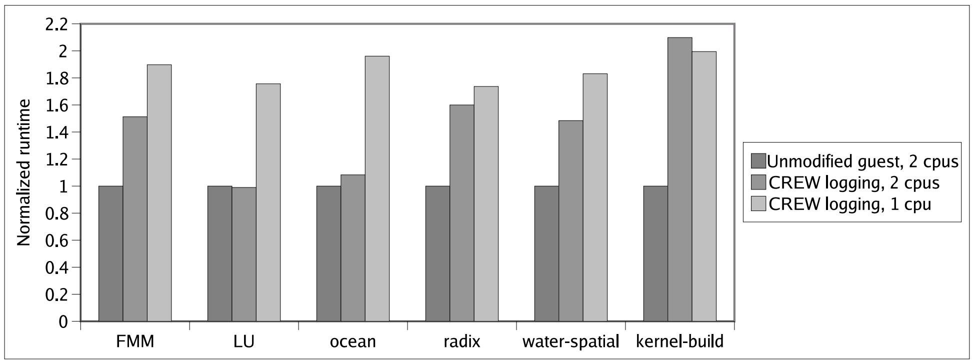
Table 5.1: Space overhead of logging a single-processor guest.

the performance of kernel-build to suffer somewhat, and dbench to suffer greatly.

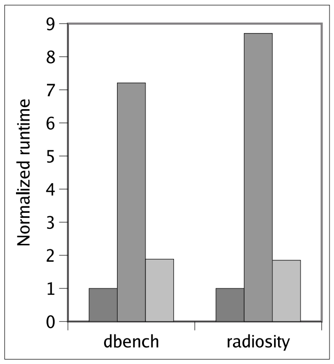
5.7.3 Results and analysis

Figure 5.6 shows the normalized runtime for ReVirt for uniprocessor workloads in Xen. All workloads are run with only one thread. The results are similar to those under UML. Kernel build has approximately 12% overhead. Some of the workloads of the SPLASH2 suite have approximately 5% overhead, and most have negligible overhead. Table 5.1 shows the space overhead. As with uniprocessor ReVirt, we show the log size in compressed gigabytes per day, and the time to fill a dedicated 300GB logging disk.

Figure 5.7 shows the normalized runtime of ReVirt for a two processor system, compared to the same test running on unmodified one- and two-processor systems.



(a)



(b)

Figure 5.7: Overhead of Revirt for a two-processor Xen guest

Workload	Logging rate, compressed	Time to fill a 300GB disk
FMM	34.5 GB/day	8.7 days
LU	3.23 GB/day	92 days
ocean	4.34 GB/day	69 days
radix	39.9 GB/day	7.5 days
water-spatial	36.3 GB/day	8.3 days
kernel-build	43.3 GB/day	6.9 days
radiosity	88.4 GB/day	3.4 days
dbench	77.0 GB/day	3.9 days

Table 5.2: Space overhead of logging a two-processor guest.

Workloads on 2-cpu guests are run with two processes, with the exception of kernel-build, which uses three; all workloads on one-vcpu guests are run with one process. Table 5.2 shows the space overhead.

For LU and ocean, the time overhead is negligible. FMM and water-spatial are significantly slower than an unmodified two-vcpu system, but significantly faster than a one-processor system. Radix is only slightly faster than a one-processor system. Kernel build is slower than the one-processor system. Radiosity and dbench perform extremely poorly: Radiosity runs 8.7 times slower than an unmodified domain with 2 virtual cpus, and dbench runs 7.2 times slower.

All systems have significant log size requirements, but even our worst case applications can run for several days before filling up a 300GB disk.

To determine the source of the overhead, we instrumented different sections of the code that we expected to be major contributors to the overhead. Figure 5.8 gives the results. The bar at the bottom shows the runtime of the unmodified Xen guest. Next up from the bottom are the measured times of different code paths. Note that shadow fault contains CREW fault, and that CREW fault contains the time waiting for the remote TLB flush. The “delayed log” is time spent actually logging constraints and events, just before the hypervisor lock is released.

Because each of the codepaths is executed often, there is a measurable overhead to adding the tracing. We calculated this overhead by subtracting the time of the tracing run to the time of the logging run without tracing.

There is still some unaccounted overhead. This is to be expected. There are many sources of hardware overhead that the CREW protocol will cause. These sources include hardware and software context switch time for the page faults (i.e., the time between the faulting instruction beginning to execute and the time control

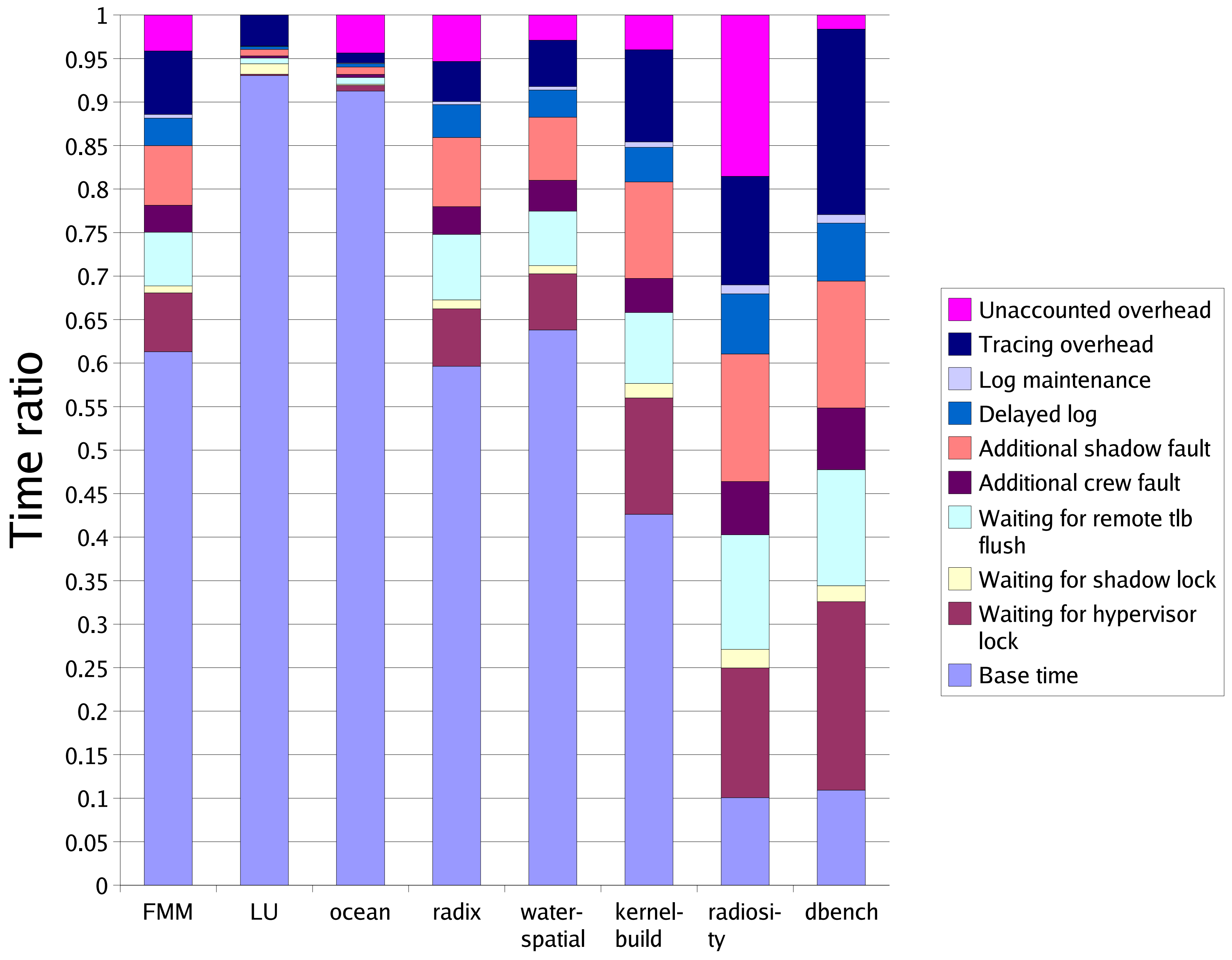


Figure 5.8: Breakdown of overheads logging a 2-processor guest

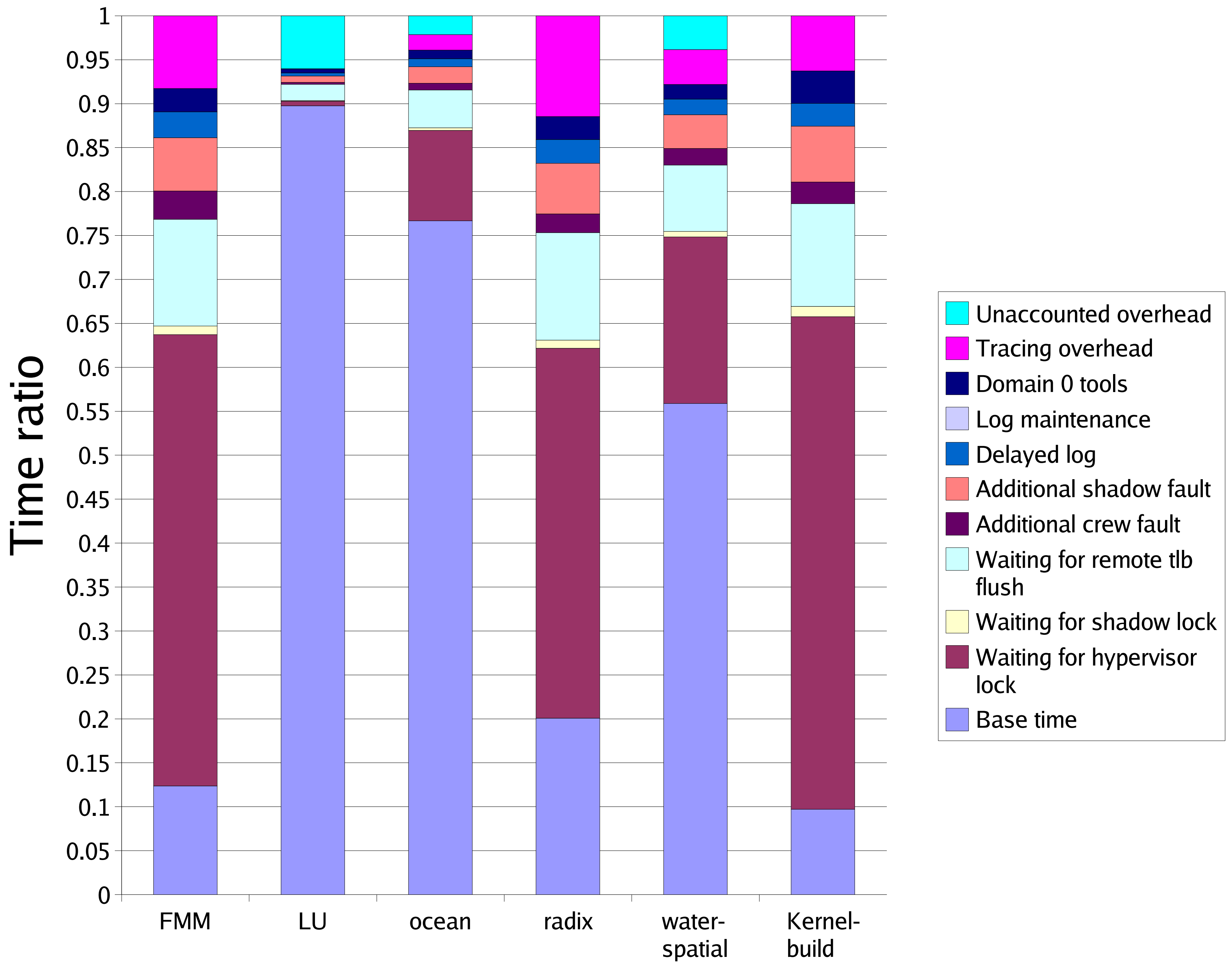


Figure 5.9: Breakdown of overheads logging a 4-processor guest

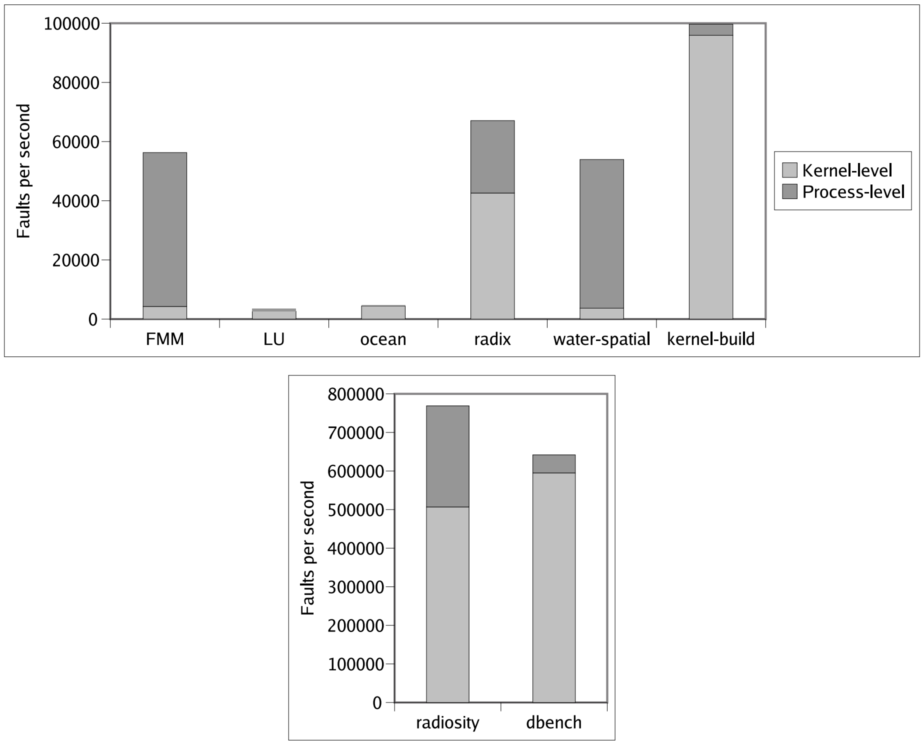


Figure 5.10: Kernel- and process-level sharing rates, in faults per second, for a 2-processor guest

reaches the shadow fault handler, and vice versa), TLB misses due to the extra flushes, and cache misses from all the code and data touched on the shadow and CREW codepaths. Also unmeasured is the overhead of running the domain 0 tools, including the process taking the log from the hypervisor and writing it to disk, and the replay drivers.

Note that the hypervisor lock reduces the contention for the shadow lock. Even if we could eliminate the hypervisor lock and allow concurrent hypercalls, much of the time waiting for the hypervisor lock would be replaced with time waiting for the shadow lock.

The majority of the overhead is traceable directly to sharing, measured in the number of faults. Figure 5.10 presents the sharing rate for the 2-processor guest, broken down by kernel- and process-level, in faults per second. We calculate the

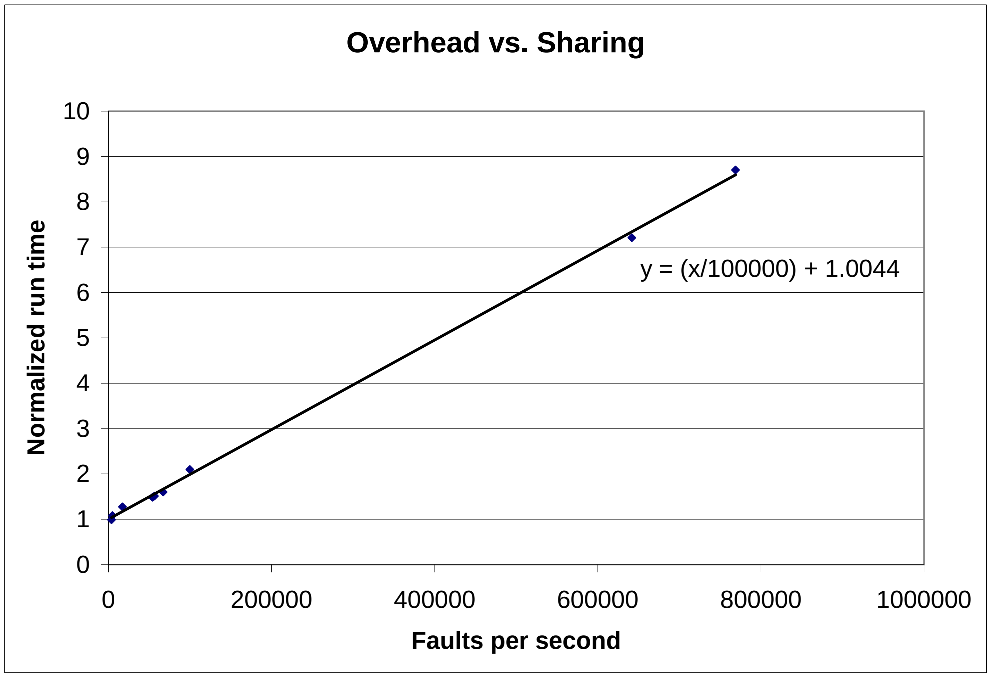


Figure 5.11: Normalized run time plotted against faults per second. A regression analysis gives a slope of 1/100000.

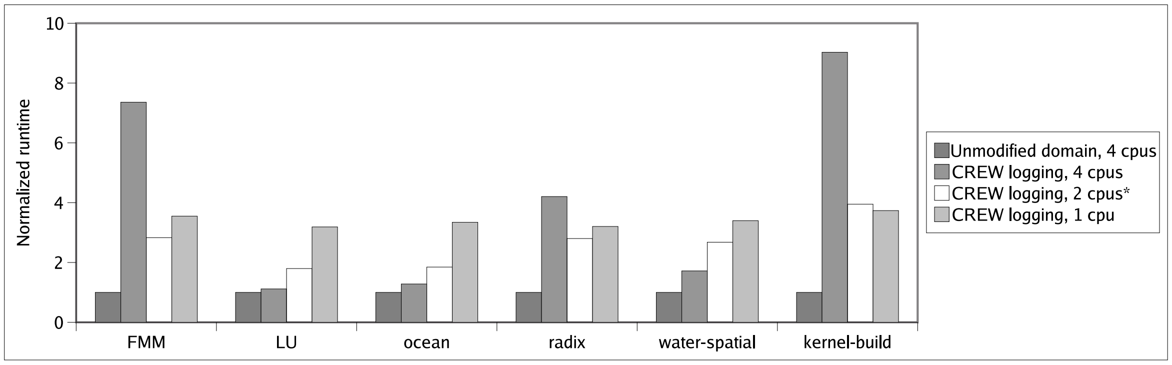


Figure 5.12: Overhead of Revirt for a four-processor Xen guest. The 2-processor CREW Logging was run on different hardware, but has been normalized to the 1-processor CREW logging case.

sharing rate by dividing the total number of faults in the logging run by the time of the unmodified Xen guest run[12].

We can see from Figure 5.10 that LU and ocean have very little sharing; the little sharing that occurs is in the kernel. This is consistent with what we expected from our knowledge of the workload. Water-spatial and FMM have a large amount of processes-level sharing. Because FMM and water-spatial have minimal kernel sharing, their total sharing is low enough to beat a one-processor system. Radiosity has large amount of process-level sharing, and an even larger amount of kernel-level sharing. Nearly all of the sharing in dbench and kernel-build comes from the kernel.

Figure 5.11 plots the normalized run time against the sharing rate. The result is highly linear, with a slope of around $1/100,000$.

Figure 5.12 shows the overhead on a 4-cpu system. The system was a system with two dual-core Xeons (four cores in all). The graph shows the unmodified domain, as well as 4-cpu and 1-cpu logging runs on the system. The graph also includes

[12]It may seem more natural to divide the logging fault count by the logging runtime. However, dividing by logging runtime gives misleading results. Large fault counts are made to look less severe by the very overhead they produce. Dividing by the unmodified guest runtime treats the fault rate as a property of the workload.

Workload	Logging rate, compressed	Time to fill a 300GB disk
FMM	83.6 GB/day	3.6 days
LU	11.7 GB/day	25.7 days
ocean	28.1 GB/day	10.7 days
radix	88.7 GB/day	3.4 days
water-spatial	58.5 GB/day	5.1 days
kernel-build	90.0 GB/day	3.3 days

Table 5.3: Space overhead of logging a four-processor guest.

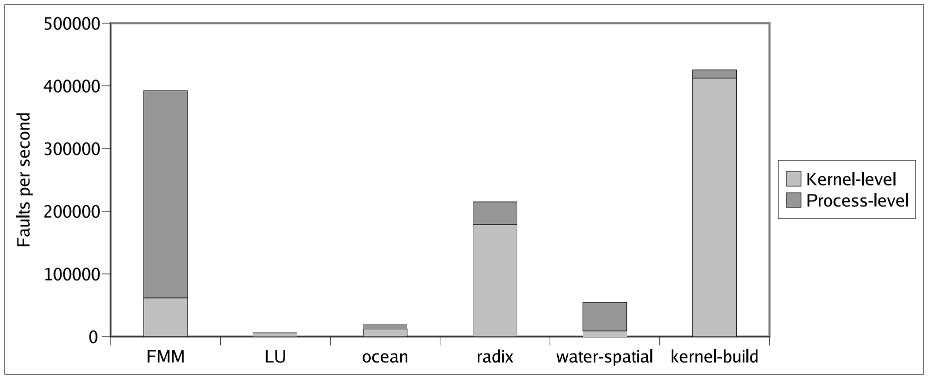


Figure 5.13: Kernel- and process-level sharing rates, in faults per second, for a 4-processor guest

2-cpu runtimes from the other hardware, normalized to the 1-cpu logging run on this machine. We did not run radiosity and dbench. Table 5.3 summarizes the space overhead.

Interestingly, Kernel-build ran much slower on the 4 vcpu system than on the 2 vcpu system. Radix and FMM ran slower, and FMM ran proportionally slower than radix. LU and ocean still run close to unmodified, although ocean begins to show some more overhead. Water-spatial runs surprisingly well.

Figure 5.13 shows the sharing rate, broken down by kernel- and process-levels. We can compare the results to Figure 5.10, but because they are on different systems, we cannot make conclusions without more data. FMM has considerably more sharing on the 4-processor guest; about six times as much for process-level and fourteen times for kernel-level. Radix's process-level sharing is comparable to the 2 vcpu system,

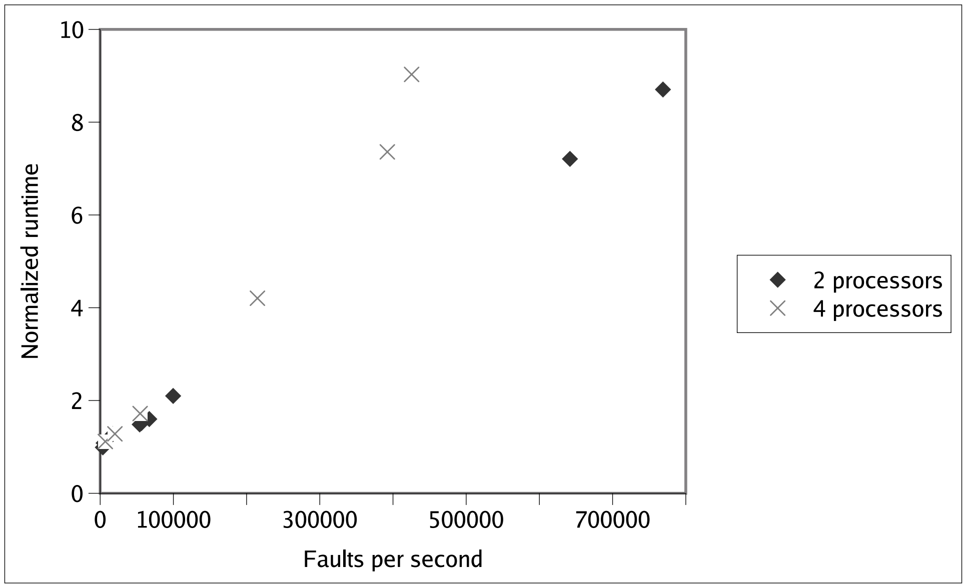


Figure 5.14: Normalized run time plotted against faults per second for 2- and 4-processor guests.

but its kernel-level sharing is a more than four times longer. Water-spatial's process-level sharing is about the same for 2 and 4vcpus, and although there is twice as much kernel-level sharing, its overall effect is still small.

Figure 5.9 shows the overhead breakdown. Note that waiting for the hypervisor lock makes up a much larger percentage of the overhead—over half of the overhead attributable directly to the sharing.

Figure 5.14 shows the plot of normalized runtime versus fault count. The results begin to look non-linear. Further investigation is required to understand the curve.

5.8 Discussion

For the current system, what is the best way to make use of a multi-processor host? We will point out benefits and costs to running a workload in different configurations. The user must decide whether the benefits are worth the cost.

Recall that we have basically three options of how to run the workload on an multiprocessor system: on one single-processor guest (leaving the other processors for other workloads), one multiple single-processor guests (a virtual network of computers), and on one multiprocessor guest. For the multiprocessor guest case, we have the option of choosing the number of processors to be equal to or less than the number of processors on the system.

We have evaluated how each of the workloads runs on one single-processor guest and one multiprocessor guest for different numbers of processors. For a network of single-processor guests, we expect sharing to be more expensive than on a multiprocessor guest, both in terms of time and space. However, only the process level of the workload is shared. By switching to a network of single-processor guests from a single multiprocessor guest, we eliminate the kernel sharing. So although we expect workloads with a large amount of process-level sharing to run slower on a network of single-processor guests, we expect workloads with little process-level sharing but a large amount of kernel-level sharing to run faster over a network, because we eliminate the need for kernel sharing.

We can estimate the potential cost and benefits in terms of time from Figure 5.10. In one class are applications like LU and ocean. These have low sharing rates overall, and so should run well either in a single multiprocessor guest or a network of single-processor guests. We expect that the ease of administering a single multiprocessor guest will make this the best option.

Another class includes applications like FMM and water-spatial, which have significant process-level sharing and little kernel sharing. These applications have some benefit to running on a 2-processor guest, but a significant cost in terms of time and log space. We expect there to be little benefit to running on a network of single-processor guests, because of the extra cost of sharing. Water-spatial does well on a

4-cpu guest, but because of FMM's extra sharing, it should only be run on a 2- or 1-processor guest.

Yet another class includes applications like kernel-build, which have little process-level sharing but a large amount of kernel sharing. There is no benefit to running kernel-build on a multiprocessor guest, but we expect there to be significant benefit to running it in a network of single-processor guests. Again, whether the extra administration cost is worth the benefit is a decision only the user can make.

The radix workload has a moderate amount of process-level sharing, and a lot of kernel-level sharing. There is some benefit to running under a 2-processor guest, but a significant cost in terms of log space. There may be more benefit to running it on a network of single-processor guests, especially in the case of a 4-cpu host.

Finally, we have workloads such as radiosity and dbench. Dbench is a kernel filesystem workload. As such, it cannot be easily distributed over multiple kernels[13]. Since running on a multiprocessor kernel has no benefit, the best option is to run such a filesystem workload in one single-processor guest.

The best option for radiosity is similar. It has significant kernel-level sharing, but also has significant process-level sharing; enough that we expect it to run poorly on a network of single-processor guests. For this workload, the best option is also to run on a single-processor guest.

5.9 Future work

We have evaluated SMP-ReVirt under the current system, but there are many opportunities for improvement. The workloads are very sensitive to the cost per fault; adding individual instructions in the critical path adds measurable overhead

[13]We could conceivably divide the target filesystem into two filesystems on two servers, but that would be a different workload.

to workloads like kernel-build and radix. We have done some initial work at profiling and engineering this, but we expect that with considerable work it could be made even faster. Certain aspects such as the hypervisor lock and the global shadow lock do not scale well and cause significant overhead by themselves. Engineering the CREW and shadow faults would benefit any workload running on the system.

Another option to explore is architecture-style optimizations. For instance, if we frequently see an access to page B following an access to page A, then when a virtual processor accesses page A, we can pre-fault page B in as well. Other potential techniques include page pre-fetching, `eip` profiling, switching to emulation for sub-page sharing, and so on. These may be able to reduce the number of faults and constraints without changing the workload.

Optimizing the kernel will also have a wide impact on a wide variety of workloads. The Linux kernel has been tuned to run on hardware that has significantly different parameters than our CREW system provides.

One promising avenue is to investigate is Linux's support for **non-uniform memory access** (NUMA) architectures, also called cc-NUMA architectures (for cache-coherent NUMA). While processors in symmetric multiprocessor systems have equal access to main memory, in NUMA machines each **node** has memory that is local to it. It can access all memory in the system, but it is more expensive to access memory from other nodes than its own. If we divide up memory between virtual cpus, it should induce the kernel to reduce unnecessary sharing between them.

Making the kernel CREW-aware will significantly reduce the sharing in the kernel, so that workloads like kernel-build, and possibly even a workload like dbench, will run well.

CHAPTER VI

Conclusions

The intrusion logging and analysis systems are incomplete—they lack enough information to consistently understand and recover from an intrusion. We have proposed execution replay as a practical way to add completeness to forensic logging. Execution replay can fill the gaps in current logging systems because it can recreate the entire state of the system at any point in time.

To demonstrate our thesis, we have described ReVirt, an execution replay system for virtual machines. ReVirt satisfies all the requirements listed in Chapter III for an execution replay system targeted at security. It functions for arbitrary code and handles all corner cases. It can run all current Linux application software without modification, and requires no modifications to the kernel other than what the virtual machine requires. It can run a variety of workloads at a reasonable time and space overhead. The narrow virtual machine interface that ReVirt replays is reasonably secure from layer-below attacks. ReVirt has been used by several subsequent security research systems, including Backtracker[35] and IntroVirt[33], as well as the TTVM debugging system[36].

Execution replay in multiprocessor environments introduces some additional complexity, but we have shown that there are good ways to use execution replay on a multiprocessor. SMP-ReVirt can log and replay multiprocessor guests on commodity hardware. To our knowledge, this is the first system that can log and replay unmodified, multi-processor kernels and software without specialized hardware. Using SMP-ReVirt, many workloads can take advantage of multiple processors to gain speed without the complexity of dividing the workload over several single-processor guests. Making the guest kernel aware of SMP-ReVirt should allow even more workloads to take advantage of multi-processor guests.

We therefore conclude the execution replay is indeed a practical way to add completeness to forensic logging.

APPENDICES

APPENDIX A

Execution replay: The Dirty Details

This appendix is a summary of our experience with implementing execution replay on `x86` hardware.

A.1 Precise event replay

In order to replay an event at the same point in the instruction stream, we need a way to identify individual instructions in an instruction stream. In the text, we call this an **abstract instruction count**. In practice, this might be an actual count of instructions, but could also be some other number or tuple of numbers. In order to label uninstrumented code, we need the assistance of the hardware. Most modern `x86` processors define a number of **performance counters**, which count various events on the system. We can use these in our abstract instruction count.

A count or tuple must have three properties to be usable for precise event replay. The first property is **monotonicity**. This is inherent to any counter, but it is important if we are to be able to re-deliver events efficiently. The second property is **repeatability**. We must be able to generate the exact same count every time the same code is run. If there are any non-deterministic effects, we must be able to compensate for them. Finally, we need **uniqueness**. That is, each instruction must have a different label, so that we can distinguish between them[1].

The tuple that we use is $< branchcount, \texttt{eip} >$[2]. The observation is that if two different instructions in an instruction stream have the same instruction pointer, a branch of some kind must have occurred between them. The branch count must include any event which can modify an instruction count, including interrupts or other exceptions, as well as normal conditional branches.

A.1.1 Hardware counters

The Intel Pentium 4 (P4) has two branch counters. The one we use in our system is called `retired_branch_type`. It counts only branches, not interrupts and exceptions. Unfortunately, it does not count FAR calls: if two instructions with the same `eip` occur in an instruction stream with only FAR calls between them, our system will not be able to distinguish between the two. We have verified this problem by writing code to exercise this scenario and observing the break in replay. We have not seen this failure in the normal course of our runs, but it is a security problem, as an attacker could easily introduce and run code which will trigger this bug as soon as he broke in, breaking replay of anything past that point.

The second counter is called `branch_retired`. Unfortunately, along with normal branches, it counts `rti` (return from interrupt) and `rsm` (return from SMM[3]) instructions that end in ring 3. The hypervisor can infer `rti` instructions, since it handles all interrupts; but SMM interrupts are by design transparent to the hypervisor and operating system, so we cannot easily compensate for these without instrumenting

[1]Note that accuracy of the counter (whether it accurately counts exactly what it is designed to count) is not one of the requirements.

[2]When comparing two tuples for ordering purposes, we begin by comparing the first element of each tuple; if they match, compare the second element, and so on.

[3]SMM stands for "system-management mode". It is designed for motherboard manufacturers to add functionality or fixes transparent to the operating system.

the BIOS.

The AMD Athlon has a "Retired Branch" event. It counts interrupts and exceptions that happen in ring 3, so these must be subtracted by the hypervisor or operating system. We have not encountered any problems with SMM on the Athlon with this counter.

The counters for both P4 and Athlon are contained model-specific registers (MSR). The MSRs themselves are 64-bits, but the counters are only 40 bits on the P4 and 48 bits on the Athlon, so they must be sign-extended when read. Care must also be taken that the counters do not overflow as well. A 3 GHz processor could overflow a 40-bit counter in a few minutes.

A.1.2 Repeat-string instructions

The `x86` instruction set include a series of instructions called **repeat-string** instructions. These repeat a specific operation (for instance, a memory-to-memory copy) a large number of times, without changing the eip or branching. These instructions can be interrupted in the middle and resumed. This is possible because the instruction uses the `ecx` register as a counter. At each iteration of the instruction, `ecx` is decremented. Depending on the nature of the instruction, execution ends either when `ecx` reaches 0, or when some condition of the target data is met.

In order to repeat interruptions of repeat-string instructions, we must add the `ecx` register to our tuple. Thus our final tuple is $< branchcount, \texttt{eip}, \texttt{ecx} >$. When determining a match or an order, we first compare the branch count, then the `eip`, then the `ecx` register.

A.1.3 Delivering asynchronous events

Logging asynchronous events is straightforward: we enable the appropriate branch counter, and compensate for any non-deterministic effects (such as subtracting interrupts from the Athlon branch count). When we deliver an asynchronous event during logging, we record the branch count, `eip`, and `ecx` at that point. We will refer to this as `log.bc`, `log.eip`, and `log.ecx` for the remainder of this section.

Replaying is more subtle. We begin by setting up the branch counter as before, and by setting up the **performance counter overflow** interrupt. This interrupt will trigger whenever the counter overflows (i.e., goes from negative to positive). Then, to re-deliver the interrupt, we use the following is the idealized algorithm:

- Set the performance counter to `-log.bc`. Let it run until the performance counter overflows. Now, `current.bc==log.bc`.
- Set the breakpoint for `log.eip`, and let it run until you get a breakpoint exception. Now `current.eip==log.eip`.
- Single-step until `current.ecx==log.ecx`.

There are several things which complicate this simple algorithm. The first complication is that the performance counter overflow interrupt is not precise; many cycles may pass between the time the counter overflows and the time the processor is interrupted. Our solution is to add a “slack” value to the overflow. We have found 128 to be a good number on our hardware. So we set the performance counter to `-(log.bc-128)`. We then need to breakpoint until `current.bc==log.bc`.

The second complication is that if you are already at `log.eip` but not yet at `log.bc`, you need to step over it before setting the breakpoint again. One option would be to use the **resume** flag in the `eflags` register. Unfortunately, we have had problems with this option, and use alternatives instead.

For normal instructions, you can single-step once to get to the next instruction, then set the breakpoint. However, repeat string instructions (discussed above) will only step partway through the instruction. The best solution is to decode the instruction length, set a breakpoint at the next instruction, then set a breakpoint at the target instruction again.

When single-stepping to step over an instruction, the instruction might cause a fault to be delivered to the guest (for instance, a page fault). Because the initial instruction did not finish, no single-step trap is generated by the hardware[4]. The first instruction of the guest fault handler will then execute without the replay system being involved.

What we want is to single-step into the fault. Hypervisor fault delivery handlers must be instrumented to detect that the replay system is single-stepping, and to return control to the replay engine after a fault is delivered if so.

Another complication concerns single-stepping to the middle of a repeat-string instruction. The Pentium 4 has a feature we call "fast repeat string" instructions; if the `ecx` value is more than 128, it will do 128 iterations for a single-step, rather than just one. For example, suppose that `current.ecx==256` and `log.ecx==250`. If you single-step, then when you return `current.ecx` will be 128, far past past where you wanted to break.

To solve this problem, we take advantage of the fact that the value of `ecx` has no effect on the string instruction other than when to finish the instruction. So one could set the `ecx` to some arbitrary value (say, 2), and single-step. Because 2 is less

[4]The `x86` defines a trap as an exception that happens after an instruction executes.

than 128, the `ecx` will only be decremented by 1. We can continue setting the ecx each time we single-step. In this case, we need to keep "shadow" values of the `ecx`, so that we know when to deliver the interrupt and restore the proper value of `ecx`.

Single-stepping a repeat-string instruction until the ecx matches can take a long time. Another option is to use "fast repeat-string replay". This technique allows the repeat string instruction to execute at full speed. We do this by setting the `ecx` to `current.ecx-log.ecx`, and breakpointing on the next instruction. When the number of repeat string iterations have happened that we need, `ecx` will be zero, the instruction will finish, and the next instruction will hit the breakpoint. When we hit the breakpoint, we set the `eip` back to `log.eip`, and `ecx` back to `log.ecx`.

The final complication is with the **resume** flag (RF) in the `eflags` register. The RF flag was designed specifically for the case above, where you are at an `eip` and you want to break the next time this `eip` is executed. If the flag is set, and the breakpoint would normally have triggered on this instruction, it does not. In any case, the flag is cleared after one instruction executes.

Unfortunately, the flag is now used for another purpose not directly relating to breakpoints. Under certain conditions, the RF flag must be set on the return from an exception to prevent a double exception from happening. Rather than force the operating system to do this, the hardware will set the RF flag automatically when certain exceptions happen. A side effect is that if there's a fault, and the breakpoint is set on the return address of the fault, the breakpoint exception won't trigger.

As a further impediment, the `iret` instruction, which can set the RF flag, is not a privileged instruction, and cannot be trapped. The `iret` instruction is commonly called by the guest kernel to return to guest user-space, and in our experience frequently sets the RF flag. Furthermore, any attacker could use the `iret` program to set the RF flag; at some point, an interrupt would be delivered when the RF flag was set.

For this reason, in Xen-ReVirt, we do not use the hardware breakpoints; we only use the performance counter overflow and the single-step mechanism. User-Mode Linux is subject to this security vulnerability, but has not yet been updated to avoid use of the hardware breakpoints.

Instead of using the hardware breakpoints, we could instead use the `int3` instruction. The `int3` instruction is a single-byte instruction that can be written in the place of the instruction that we want to break on. The original value is stored, and replaced when we want to execute the instruction again.

There are two primary difficulties with using the `int3` instruction. The first is that the address space may change between the time we "set" the breakpoint and the time the target breakpoint actually happens. With the hardware breakpoint, this did not matter, since the virtual address is compared on each instruction. With a software instruction breakpoint, every time the mapping of the virtual address in question changes, we must restore the old value before the change, and restore the breakpoint after the change.

The second difficulty is that of self-inspecting or self-modifying code. Whenever we modify guest pages, we need to be careful that the data is not read and propagated elsewhere; for instance, read and copied, or written out to disk. An attacker could easily write self-inspecting code in a tight loop purposely, to detect when the system is replaying and do something different than during logging. It may be impossible to defend against this type of attack, as it would involve making a page with execute permissions but no read permissions.

A.1.4 Counters in a multi-process or multi-domain environment

The performance counters, when configured properly, will be active any time the processor is executing in rings 1-3. If the execution replay system has multiple processes or multiple domains, the counters will increment when other domains are executing. This means that some accounting must be done on context switches.

The general idea is to save and restore the performance counters in the per-process (or per-domain) structure on every context switch. Unfortunately, reading and writing performance counters can be expensive; it added a measurable overhead to some benchmarks run on our early system. Saving and restoring the exact performance counter causes a read of the old value and a write of the new value on every context switch. We use instead an algorithm that requires only one read per context switch. When a process is scheduled in, we read the value of the performance counter and save it in the process structure. When the process is scheduled out, we read the value, subtract the new value form the one stored at the beginning, and add the count to a per-process "accumulator". If we are switching from one performance-counted process to another, the same read can be applied to the end of the domain being scheduled out and the beginning of the domain scheduled in.

The final issue to point out regarding performance counters in a multi-process or multi-domain environment is that sometimes the performance counter overflow takes many cycles to actually be delivered. It is possible for the overflow to happen in one process just before schedule is called, and for the interrupt to be delivered in another process, just after the schedule completes. So it is important, when handling interrupts, to be aware that the overflow may have triggered in a different context.

A.2 The Floating Point Unit

Normally, a process initializes its own floating point unit (FPU) before using it; it should make no assumptions about what is in it. Our experience with ReVirt indicates that the initial state of the FPU is as important as the initial state of the process. The initial state can make its way (via floating-point store instructions) into main memory, which can affect the execution of the system later. The FPU for a process should be set to a known state before beginning.

In the original `x86` design, the entire FPU was affected by the FPU save, restore, and initialize instructions. However, in certain situations, certain parts of the FPU state are unimportant from an operating system or program point of view. In order to speed up these instructions, Intel introduced an optimization in the P4 that would only save these under certain conditions. Unfortunately, we found that this optimization can lead to non-determinism in the FPU state, which can then leak into memory. In order to prevent this, we turn off the optimization by enabling the Fopcode compatibility submode.

A.3 Xen shadow synchronization

We described shadow pagetables briefly in chapter V. One of the major challenges with shadow pagetables is keeping the shadows in sync with the guest pagetables. Traditionally, this has been done by removing write permission to any guest page that is being used as a pagetable. Writes are trapped and emulated, allowing the hypervisor to update the shadow pagetable as the guest pagetable is updated. This results in somewhat poor performance when batches of updates to a single pagetable are made, as frequently happens in Linux when doing a `fork()`.

Xen 2.0 introduced a new technique to deal with this synchronization, where shadow pages are allowed to be out of sync with guest pages for periods of time, and brought into sync later. This **shadow-sync** technique gave performance improvement to certain operations in Linux guests running in shadow mode, namely the `fork` system call. However, it resulted in a considerable amount of extra complication in the shadow code, especially for multiprocessor guests.

The shadow-sync algorithm was taken out of Xen as of version 3.0.3, which was released after this thesis was defended in late September, 2006. The shadow code was entirely re-written with a new target: Microsoft Windows$^{\mathrm{TM}}$running under Xen using hardware virtualization such as Intel's VMX or AMD-V. Since Windows does not use the Unix `fork` model to create new processes, the shadow synchronization technique was deemed an unnecessary complication. All writes to guest pagetables are now emulated, so that guest and shadow pages are updated at the same time. The technique may be re-introduced at a later time if situations are found for which it is a significant performance improvement, but it will be limited in scope to very specific instances.

Because this shadow-sync technique is an interesting and reasonable way to implement shadow pagetable updates, we describe it here, along with our multiprocessor extensions, and how we integrated it into SMP-ReVirt.

The observation behind the shadow-sync technique is that real processors generally have a cache of pagetable entries, called a TLB, which must be kept in sync with the actual hardware pagetables. On the `x86`, this "cache" is filled automatically when it does not contain an entry, but must be explicitly flushed when memory pagetable entries are modified. Because of this, most operating systems which run on this hardware must both be aware of the difference between what's in the memory pagetable and in the TLB (flushing it when appropriate), and must also be able to deal with spurious page faults due to differences between the TLB and the in-memory pagetables.

Xen 2.0 takes advantage of this fact, treating the shadow pages as a very large TLB. Guest pages are marked as **in-sync** or **out-of-sync**, depending on whether the shadow accurately reflects the guest pagetable or not. An in-sync pagetable is read-only. When an in-sync guest pagetable is written to, the processor faults. The hypervisor will then take a snapshot of the guest page, mark a page out-of-sync, and give the guest write permission to it. The guest can then write to the page as it will.

Out-of-sync pages are brought back in sync when either when a page fault happens or when the guest kernel does a TLB flush. This mirrors what might happen with a large TLB. If a kernel writes a new pagetable and then accesses the corresponding virtual memory, the TLB misses and the new pagetable entry is read into the TLB. Similarly, if the guest kernel writes a new guest pagetable entry pagetable and then accesses corresponding virtual memory, there will be a page fault, and the shadow pagetables will be updated to reflect the new guest pagetable entry.

If, on the other hand, the kernel modifies an old pagetable entry, the old pagetable entry may still be in the TLB, and may be used until the TLB is flushed. Similarly, if the guest kernel writes a new guest pagetable entry, the old entry will still be in the shadow pagetables until the guest does a TLB flush. The TLB flush will cause the shadow pagetables to be updated to reflect the new pagetable entry.

When pagetables are synchronized, the new guest pagetable is compared to the snapshot taken when the page was marked out-of-sync. The entries that have changed are updated in the shadow pagetable. The snapshot is then discarded, write permission removed from the page, and the page marked in-sync again.

This shadow-sync technique resulted in a performance increase for certain operations in Linux when running in shadow mode. However, it caused problems for execution replay.

The first problem with the shadow-sync technique is that the synchronization state has an effect on both the execution of the system and, through the dirty and accessed bits, on the memory state of the system. This means that if shadow syncs happen at different times during logging than during replay, then the execution and state may diverge, breaking replay.

Suppose a pagetable entry is in sync, and has the dirty bit set. The shadow pagetable will have write permission. Then suppose that the operating system clears the dirty bit, and shortly thereafter the page in question is written to again.

When the operating system writes to the pagetable to clear the dirty bit the pagetable will be marked out-of-sync. If a shadow sync happens between the time the dirty bit is cleared and the next write to the page, the page will have read-only permissions. This will cause a fault, which will allow the hypervisor to update the dirty bit in the guest pagetable. If, on the other hand, there is no shadow sync between the time the dirty bit is cleared and the next write to the page, the old shadow pagetable entry with write permission will still be in effect. No fault will happen, and the hypervisor will not set the dirty bit, even though a write has happened.

To eliminate this problem, we must synchronize during replay at the same points as during logging. This is complicated for single-processor systems, but is even more more complicated in multi-processor systems. In multiprocessor systems, defining what in-sync and out-of-sync means presents some challenges. If a guest pagetable is in-sync on all processors, and one processor writes to it, it must be made out-of-sync on all processors. However, if one processor does a shadow sync, what should happen? Should we synchronize all processors? Synchronize the one processor, but leave the others out of sync? Synchronize the one processor, update its snapshot, and mark the others out of sync again? What happens if, while the other processors are still out-of-sync, the page is written to again? Which snapshot is used to bring a page back into sync? And how are we to make the syncing deterministic, so that it can be made the same for each processor during replay as it was during logging?

We converged on the following design. Each guest pagetable page can be in-sync or out-of-sync on a per-vcpu basis. When a page is promoted to a guest pagetable (i.e., when it first becomes shadowed), write permission is removed from all other CPUs and the page is marked in-sync. When a virtual processor writes to an in-sync pagetable, the page is marked out-of-sync on all processors. A single shared snapshot per domain is taken of the guest pagetable. The snapshot contains a list of the vcpus for which this snapshot is still effective; this is set to the list of currently in-sync virtual cpus. When a vcpu does a shadow sync, the snapshot list is walked. For each entry which has the corresponding vcpu's bits set, the shadow code syncs the shadow page, comparing the current guest page with the snapshot to detect changes. The bit for that page is then cleared, and the page is made read-only on all processors. When the last bit of a snapshot is cleared, it is removed from the list.

Note that there can be several outstanding snapshot pages for a given page; if one vcpu does a sync, then writes to a shadow page, it will have a new snapshot, while the other vcpus which had not synced at that point would have the old snapshot. But for each vcpu, there will only be one active snapshot at a time.

Shadow syncs happen only at well-defined times, so that they will be the same during logging as during replay.

In order to make sure that syncs happen the same with respect to each other, we introduced a pseudo-CREW object, called the "out-of-sync" object, that corresponds to the per-domain out-of-sync vector. Grabbing read and write permission will cause constraints to be generated the same way a they would for a physical page. Read permission for this object is grabbed before reading the out-of-sync vector, and write permission is grabbed before setting or clearing the vector. This will ensure that shadow syncs and marking pages out-of-sync will be ordered with respect to one another; given this ordering, the syncing is deterministic.

Whenever a page is read by the shadow code (such as on syncing or on promotion), it is also grabbed via the CREW protocol. This guarantees that the reads which generate the shadow pages will get the same values during logging and replay.

The final issue we encountered deals with hypercalls that act on another vcpu's shadow pagetables. These operations include shadow sync (which can be passed a vcpu mask), and pagetable updates (which are guaranteed by the API to be reflected in the shadows of all vcpus on return from the hypercall). The general technique is to do operations which are strictly reducing in privilege while holding the shadow lock. When we are done, we do a single TLB flush. All of the updates will seem to happen at the point of the TLB flush. During replay, we want this action to happen at exactly that point. So we introduce two constraints: the vcpu doing the shadow operations must wait until that point before acting on the other vcpu's shadow tables; and the other vcpu must wait until the vcpu doing the operations is complete until it can continue.

A.4 Software assists

We use the term **assist** to describe an action the hypervisor does after an instruction faults to allow the instruction to execute successfully. For example, shadow faults emulating the dirty and accessed bits would classify as an assist. If the dirty bit for a page is clear in its pagetable entry, the shadow pagetable entry will be write-protected. When the guest writes to the page, it will cause a page fault. The shadow code will set the dirty bit, give write permission, and let the original instruction finish executing.

The subtlety with an assist is that the same instruction occurs twice in the instruction stream with no branches between the first and second instances; but typically there is a state difference before and after the assist.

Consider our dirty bit emulation assist. Suppose that during logging, the system delivers an interrupt just after the emulation assist, but before the original instruction can execute again. The page is marked dirty, but the original instruction has not executed, so the tuple will be the same as it was before the assist. During replay, the system will deliver the interrupt **before** the assist (because the tuple matches); the fault and the assist will not happen, and the page will not be marked dirty. Thus the memory state will be different during logging and replay, and the next time the processor reads that pagetable entry and acts on the dirty bit, the execution will diverge.

To solve this, any assist which changes guest state or hypervisor state which affects the guest (for instance, marking a page out-of-sync) must increment the branch counter so that the replay system can differentiate interrupts that are delivered before an assist and those that are delivered after.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Report on the Linux Honeypot Compromise. Technical report, Honeynet Project, November 2000. http://project.honeynet.org/challenge/results/dittrich/evidence.txt.

[2] The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. Technical report, Intel Corporation, 2003.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[4] L. Alvisi. *Understanding the Message Logging Paradigm for Masking Process Crashes*. PhD thesis, Cornell University, 1996.

[5] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[6] J. P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co., April 1980. Contract 79F296400.

[7] K. Audenaert and L. Levrouw. Interrupt Replay: a debugging method for parallel programs with interrupts. *Microprocessors and Microsystems*, 18(10):601–611, December 1994.

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

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[10] J. Bellino and C. Hans. Virtual Machine or Virtual Operating System? In *Proceedings of the 1973 ACM Workshop on Virtual Computer Systems*, pages 20–29, 1973.

[11] M. Bishop and M. Dilger. Checking for Race Conditions on File Accesses. *USENIX Computing Systems*, 9(2):131–152, 1996.

[12] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 299–310, June 2000.

[13] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. *ACM SIGOPS Operating System Review*, 17(5):90–99, October 1983.

[14] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault-Tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.

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

[16] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the 2001 IEEE Symposium on High Assurance System Engineering (HASE)*, pages 95–105, October 2001.

[17] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[18] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the 1998 SIGMETRICS Symposium on Parallel and distributed tools (SPDT)*, August 1998.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.

[20] F. Cornelis, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. DeBosschere. A taxonomy of execution replay systems. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, pages CD–ROM paper 59, 2003.

[21] F. Cornelis, M. Ronsse, and K. De Bosschere. Tornado: A novel input replay tool. In H. Arabnia and Y. Mun, editors, *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, volume IV, pages 1598–1604, Las Vegas, 6 2003. CSREA Press.

[22] R. Curtis and L. Wittie. BugNet: a debugging system for parallel programming environments. In *Proceedings of the 1982 International Conference on Distributed Computing Systems*, October 1992.

[23] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.

[24] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, pages 211–224, December 2002.

[25] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[26] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. An Efficient Technique for Tracking Nondeterministic Execution and its Applications. Technical Report CMU TR 95-157, Carnegie Mellon University, 1996.

[27] S. I. Feldman and C. B. Brown. IGOR: a system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, November 1988.

[28] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)*, February 2003.

[29] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a record/replay system for multi-threaded java applications. In G. Joubert, H. Weberpals, M. Ronsse, and E. D'Hollander, editors, *Parallel Computing: Grids and Applications*, pages 125–146, Clausthal, 5 2002. Communication and Cognition.

[30] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.

[31] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):226–262, August 2000.

[32] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Proceedings of the 1987 International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.

[33] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104, New York, NY, USA, 2005. ACM Press.

[34] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11), November 1991.

[35] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Technical Conference*, pages 71–84, June 2003.

[36] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. Technical Report CSE-TR-495-04, University of Michigan, August 2004.

[37] R. J. LeBlanc and A. D. Robbins. Event driven monitoring of distributed programs. In *Proceedings of the 5th international conference on Distributed Computing Systems*, pages 515–522, May 1985.

[38] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, pages 471–482, April 1987.

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

[40] J. Napper, L. Alvisi, and H. Vin. A Fault-Tolerant Java Virtual Machine. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN)*, June 2003.

[41] R. H. B. Netzer and J. Xu. Adaptive Message Logging for Incremental Program Replay. *IEEE Parallel and Distributed Technology*, pages 32–39, November 1993.

[42] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the 1988 ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.

[43] M. Ronsse and K. D. Bosschere. RecPlay: A Full Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[44] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, January 1995.

[45] M. Russinovich and B. Cogswell. Operating System Support for Replay of Concurrent Non-Deterministic Shared Memory Applications. *IEEE Computer Society Bulletin of the Technical Commitee on Operating Systems and Application Environments (TCOS)*, 7(4), January 1995.

[46] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation*, pages 258–266, May 1996.

[47] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[48] M. Shapiro, S. Kloosterman, and F. Riccardi. PerDiS–A Persistent Distributed Store for Cooperative Applications. In *Proceedings of the 3rd Cabernet Plenary Workshop*, April 1997.

[49] J. H. Slye and E. N. Elnozahy. Support for Software Interrupts in Log-Based Rollback-Recovery. *IEEE Transactions on Computers*, pages 1113–1123, October 1998.

[50] E. T. Smith. Debugging tools for message-based, communicating processes. In *Proceedings of the 4th international conference on Distributed Computing Systems*, pages 303–310, May 1984.

[51] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A Capture/Replay Tool for Observation-Based Testing. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 158–167, August 2000.

[52] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[53] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June 2001.

[54] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM Press.

[55] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 2003 International Symposium on Computer Architecture*, June 2003.

[56] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

ABSTRACT

Execution replay for intrusion analysis

by

George Washington Dunlap III

Chair: Peter M. Chen

Computer intrusions are inevitable. When an intrusion happens, forensic analysis is critical to understanding the attack. An administrator needs to determine how the attacker broke in, what he changed, and what privileged information he may have seen. Unfortunately, current security logging systems are incomplete, leaving large gaps in the knowledge of what happened.

Execution replay is a practical way to add completeness to forensic logging. To show this, we describe ReVirt, a virtual machine execution replay system capable of security-grade logging. ReVirt can reconstruct the entire past state of the system at any point in time, including memory and disk, and can re-execute . This enables security tools that use ReVirt to gather arbitrarily detailed information about the system before, during, and after an attack. ReVirt adds 0-12% runtime overhead during logging. A single 100 GB disk can log continuously from weeks to years.

We also describe SMP-ReVirt, an execution replay system that can log and replay multiprocessor virtual machines. Races between the processors are detected using a concurrent-read, exclusive-write (CREW) protocol enforced with hardware page protections transparently to the virtual machine. This is the first execution replay system to log and replay a multiprocessor kernel outside of simulation. Performance depends heavily on the sharing rate of the workload. Some parallel applications run with overhead around 1%, while some run an order of magnitude slower with the logging enabled. Logging rates depend upon sharing rates. A 300GB disk can log workloads with low sharing rates for several years, and can even log workloads with very high sharing rates for several days.