A Recipe for Deterministic Replay

Anton Burtsev David Johnson Mike Hibler Eric Eide John Regehr

University of Utah
Salt Lake City, UT USA
{aburtsev, johnsond, mike, eeide, regehr}@cs.utah.edu

Abstract

Despite many implementations over the past decades, deterministic replay has not become a de facto part of the systems software stack. Prior work has shown that whole-system deterministic replay can impose little run-time overhead while enabling powerful analyses. What has not been shown, however, is a clear and general recipe for implementing VM recording and replay. We assert that this lack of published knowledge impedes the understanding, development, and adoption of deterministic replay as a component that should be ubiquitous in modern VMMs. We present a recipe for implementing deterministic replay. The recipe is based on a three-part model of VMM-based systems and describes the assembly of ingredients for an efficient replay engine. Our goal in presenting this recipe, which we used in our XenTT prototype, is to increase the deployment of deterministic replay by providing a software architecture that can be readily followed and adopted.

1. Introduction

In the past decade or so, many research projects have used deterministic replay for purposes such as debugging, performance analysis, and forensics [2, 5, 7, 9, 11]. Despite the utility of logging and replay, and the existence of several prototype implementations, mainstream operating systems and virtualization platforms have largely failed to support them. Why is this?

Historically, the deployment of deterministic replay has been hindered by several factors. First, general-purpose operating systems fail to provide a clean boundary between the replay mechanism and the replayed part of the system. The resulting process-level replay mechanisms have been hindered by the complexity of the system call interface, resulting in complex solutions that, in addition to being OS-specific, often support only a subset of applications. Second, recording and replay has had a high cost in terms of storage and CPU time.

Today, these problems are much less pressing. First, virtualization is becoming a default part of the system deployment stack. The virtual machine interface is narrow and supports a relatively clean separation between the replayed code and nondeterminism in the external world. Second, virtualization makes logging fast. Virtual machines are optimized to provide a low-overhead interposition layer: high-throughput asynchronous I/O, low-latency interrupts, and fast memory management. Third, the abundance of storage and processor power on modern systems reduces the effective penalties of recording and replay, making it feasible to record an entire virtual machine even if only a single process needs to be replayed. A commercial replay implementation from VMware can record and replay execution of enterprise workloads, e.g., Microsoft SQL and Exchange servers, 1 Gbps streams, a Hadoop cluster, etc., with an overhead of a few percent [14, 15].[1]

As a default component of the modern VMM stack, ubiquitous deterministic replay can change the way we reason about complex systems. The availability of complete system state, the guaranteed deterministic behavior of re-execution, and the absence of limitations on the run-time complexity of analysis algorithms collectively enable deep, iterative exploration of the run-time properties of whole systems, e.g., automatic debugging, explanation of cross-component performance anomalies, reconstruction of intrusion vectors, and more.

Despite the long history of VMM-based replay systems, deterministic replay is still difficult to implement.

[1] Starting with version 8, VMware Workstation dropped support for replay debugging functionality. This created a rumor that production-quality, VMM-based deterministic replay is infeasible. This is not true. VMware continues to use deterministic replay a basis for their fault-tolerant VM replication solution in vSphere.

Abstractions are badly needed. It took the authors three person-years to implement a deterministic replay engine for uniprocessor guests running on Xen 3.0.4. Despite the existence of earlier replay implementations (one in Xen [8]), the reuse of code and strategies for replay did not appear to be possible. Having no reference design, and no clear abstractions or principles from prior work to follow, we had to re-analyze sources of nondeterminism, reinvent debugging tools, and rediscover a way to split Xen into deterministic and nondeterministic parts such that recording and replay have good performance.

The contribution of this paper is an effective recipe—a collection of techniques and abstractions—that can serve as a practical guide for creating deterministic virtual machine replay. We believe that record/replay is a useful part of the virtual machine toolkit and that this recipe can substantially simplify future replay implementations, giving this technology a better chance of becoming part of the mainstream systems stack.

Our goal is to promote deterministic replay as a ubiquitous mechanism for the analysis of production systems. To this end, we focus on the replay of uniprocessor VMs, which is, at the moment, the most practical way to provide general, low-overhead replay of production workloads. A number of promising research efforts attempt to address the prohibitively large overhead [8] of recording multiprocessor systems. Still, these techniques require assumptions that are often unacceptable for production environments: e.g., the need to tolerate the overheads of whole-system binary translation [4], heavyweight execution-reconstruction techniques that are limited to several seconds of recording [1, 13], the inability to record a whole system due to strict limits on the amount of shared-memory nondeterminism [10], intrusive changes to the entire OS stack [3], or specialized hardware [12]. Fortunately, uniprocessor VMs are appropriate for many production systems, which can often use multiple VMs (horizontal scaling) to address large workloads. Our work develops mechanisms essential for any replay engine. Our replay recipe provides a general foundation for experimenting with both uniprocessor and multiprocessor architectures.

2. Deterministic VMMs Are Hard

The basis for deterministic replay is simple: the execution of a system is mostly deterministic by default, with only occasional interruption by external nondeterministic events such as interrupts, values read from I/O registers, nondeterministic instructions, and DMA updates.[2] In other words, replay can be implemented by replicating the system's initial state and then letting the CPU execute in its normal, deterministic fashion, interrupting it only to replay the stream of previously recorded external events. Although the basic idea is simple, the devil is in the details.

Complex interposition boundary. Replay requires that all nondeterministic input be available for interposition during logging and replay. Although the virtual machine is designed to have a rigid isolation boundary, in a real system it has a number of architectural dependencies on multiple parts of the virtual machine monitor: peripheral device and bus emulation code, an emulated BIOS, timer and interrupt controllers, virtual CPUs and MMUs, device configuration and VM creation tools, and so on. Each of the parts contains some state of the VM and can affect its execution. This creates two problems for the replay framework. First, for a full-featured VMM, it is impractical to detect all sources of nondeterminism by statically analyzing the VMM's code. Replay needs automated tools designed to identify undetected sources of nondeterminism at run time. Second, replay requires a general mechanism that can implement efficient, synchronized logging at every level, and then provide controlled, orchestrated execution of the multi-level system between adjacent pairs of nondeterministic events during replay.

Concurrent, reentrant environment. Modern hypervisors are designed to provide low-latency virtualization of interrupts and device I/O. They run with minimal locking to ensure preemptive, concurrent, and parallel processing of high-priority interrupts and signals. Most components are reentrant, and under high load may create interleaving of low-level updates to the state of the replayed system in an order that is still acceptable for the system, but is impossible to replay at the level of recorded events. Deterministic replay must ensure the atomicity of recording across the entire VMM stack without introducing a "big lock" into a highly concurrent system.

Complex instruction-counting logic. Despite having a long development history, precise instruction

[2] A paravirtualized VM interface eliminates the need to emulate the BIOS, legacy BIOS boot protocol, PCI bus, DMA engine, and peripheral devices. Instead, VMs rely on explicit shared-memory communication, a system-wide device configuration database, software timers, and interrupt-like event channels.

counting—required for replaying asynchronous events—is still challenging on modern CPUs. Precise instruction counting requires tracking every exit from the replayed system. This is especially challenging in the face of the System Management Mode (SMM) interrupts, which exit straight into the BIOS firmware and are transparent to the hypervisor [16]. Instruction-accurate injection of asynchronous events requires support for emulating repeated string instructions, which do not change the instruction pointer or branch counter across multiple iterations, and careful emulation of the trace flag, which is used to single-step the replayed system.

Subtle divergence bugs. A change of a single bit in the state of the replayed system can potentially alter its execution path. An analysis of the divergence is further complicated by a period of execution that is common (unchanged) between the altering change and the observed divergence. In practice, without special debugging tools aimed at recording and comparing execution traces across original and replay runs, it is impossible to implement a replay engine that scales to replay enterprise workloads.[3]

The above challenges are common to all virtualization platforms and present a replay engine with the same set of problems. Replay needs general, reusable mechanisms that can help its implementation.

3. A Replay Recipe

The first challenge for implementing deterministic replay lies in a design of the interposition boundary that cuts through the complex code of the VMM in such a way that the replay engine can ensure controlled VM execution during replay. To scale the replay engine to the complexity of realistic systems and workloads, it is critical that the interposition boundary is built on a set of concise, well-understood principles.

3.1 A Three-Part Model

Our model, shown in Figure 1, serves as a general abstraction for reasoning about nondeterminism in the complex, multi-layered, multi-component environment of a VMM. The replay system is divided into three parts: the replayed system, a deterministic execution environment, and a nondeterministic external world. The three-part split reflects the fact that the seemingly

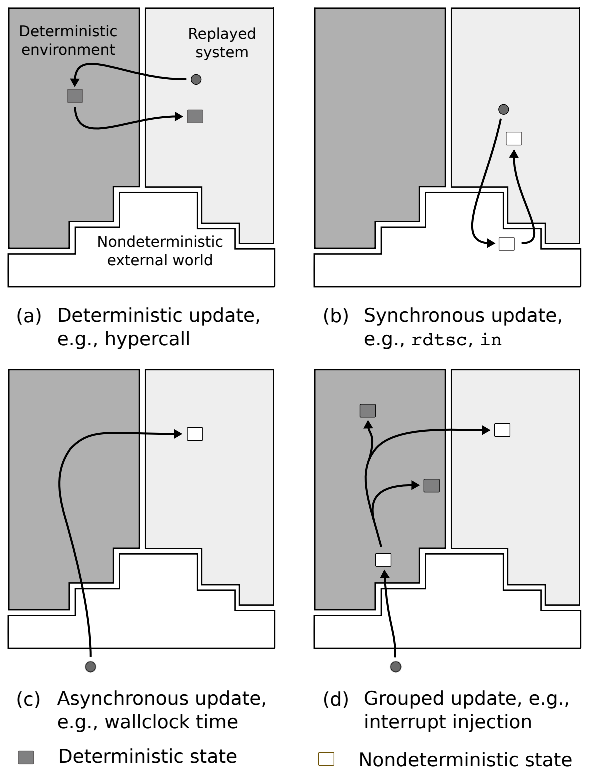
Deterministic environment
Replayed system
Nondeterministic external world
(a) Deterministic update, e.g., hypercall
(b) Synchronous update, e.g., rdtsc, in
(c) Asynchronous update, e.g., wallclock time
(d) Grouped update, e.g., interrupt injection
Deterministic state
Nondeterministic state

Figure 1. External world, deterministic environment, and replayed system.

rigid boundary of a VMM extends well outside of the virtual machine itself. Most of the hypervisor code—e.g., memory management, page-fault handling, and the hypercall interface—is deterministic and is classified as the deterministic environment. Device code, e.g., for disk, network, and console, can be forced to look deterministic to the replayed system with the help of a small layer of code, *determinizing proxies*, which ensure the determinism of observed behavior.

Our model serves the practical goals of (1) choosing the interposition boundary, which reduces the amount of recorded nondeterminism, and simplifies implementation of the replay engine; (2) providing general events types for implementing execution scheduling loop; and (3) creating a general mechanism for pluggable interposition functions. To realize these goals, the three-part model classifies all interactions between the replayed system and its environment into the following three categories.

Deterministic updates. The state of the replayed system and its execution environment evolve together by updating each other. Figure 1(a) shows an interaction where the update originates inside the replayed system

[3] The ReVirt team analyzed a replay-divergence bug caused by the order of page fault exceptions, which were required to emulate the "dirty" page bits, and external interrupts for two months [6].

and changes the state of the deterministic environment. A hypercall invocation updates the state of the hypervisor, while its return parameters update the state for the replayed system. Deterministic interactions do not need to be logged, but they do need to be re-executed during replay to ensure that both parts of the system evolve their state in the same way that they did in the original recorded run. Deterministic interactions can also serve as useful points for comparison between the original and replay executions in order to debug the replay engine.

Synchronous events. Reading a time stamp counter is an example of accessing nondeterministic data from otherwise deterministic code; this is shown in Figure 1(b). To ensure that the replayed machine follows the original execution path, synchronous events are replayed "in place." Replay interposition primitives query the replay engine and return to the system the value of the nondeterministic variable that was observed during the original run.

Asynchronous events. Asynchronous events, depicted in Figure 1(c), represent external updates to the replayed system. These include interrupts and updates to shared memory from virtual device drivers running in parallel with the replayed system. In contrast to a synchronous event—where the replay machine effectively schedules the state update itself—an asynchronous event must be replayed in an instruction-accurate fashion by the execution scheduling loop.

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

Grouping dependent events. Figure 1(d) illustrates the common case where an asynchronous event triggers the execution of a function that performs multiple deterministic and synchronous updates. For example, an interrupt event updates the flags, registers, and stack of the guest system. While it is possible to record all these updates as asynchronous events, it is easier and more efficient to record a single asynchronous update, and treat the remaining updates as synchronous events originating from the code of the handler. Of course, the replay system must ensure the atomicity of the entire handler. In many cases this is easy, since the hypervisor is already designed to make sure that the code of the interrupt handlers is atomic.

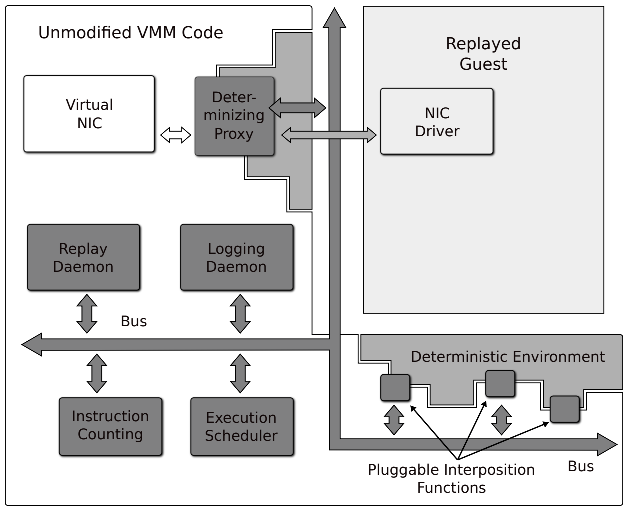

Figure 2. Components of the replay engine.

3.2 Replay Building Blocks

The parts of the replay engine are designed to utilize a coarse-grained plan of action inspired by the three-part model. For synchronous events, interposition functions record and replay events "in place"; for asynchronous events, they record and firewall events "in place," but replay is done from the execution scheduling loop.

Virtual devices are not inside the deterministic environment. However, since their execution during replay is driven by requests from the replayed system, they are "nearly deterministic"—the only nondeterministic aspect of their execution is the time at which they respond. We use *determinizing proxies* to interpose on the communication protocol between the replayed system and the device, ensuring that updates are propagated in a deterministic way. The following parts constitute the main building blocks of the replay engine (Figure 2).

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

Data and control bus. Implemented as multiple lock-free shared memory queues, the data bus provides a communication primitive that connects all components of the replay engine at all levels of the system. The interposition primitives attach to the data bus to relay the stream of nondeterministic events to and from the logger and replay daemons. The control logic is used to request the replay of a specific event from the determinizing proxies. When a proxy receives a request, it replays communication with the proxied device up to the requested point. The bus also ensures the ordering and atomicity of tracing events. We use a concept of active messages: instead of locking the system at every level of the VMM stack, the interposition functions send a message on the bus. The message encodes the update to the state of the replayed system. It is atomically processed by the bus, which preempts the replayed system, records its branch counter, and performs the update.

Determinizing proxies. Determinizing proxies ensure the determinism of the communication protocol between the replayed system and virtual devices. A virtual device accesses the state of the guest system through two mechanisms: (1) memory remapping, and interrupt signaling hypervisor calls, and (2) a region of shared memory. The determinism of hypervisor calls is ensured by the interposition layer inside the hypervisor. To ensure the determinism of direct memory updates, the determinizing proxy inserts itself between the guest system and the virtual device, and mirrors all updates to and from the guest system in a deterministic way.

Some devices, e.g., network and console, require replay of the device I/O payload. For the console device, its proxy replays the console input itself. For the more complex network device, the proxy avoids emulation of the full device protocol. Instead, it replays the network payload into the device by using the device functions.

Logging and replay daemons. The logging daemon is responsible for flushing the data bus to a permanent store. The logger runs on a separate CPU, attaches to the data bus, and makes sure that all processing of the nondeterministic stream is done off the critical path. The interposition functions, which are on the critical path, resume processing immediately after putting events on the data bus. The replay daemon provides the stream of nondeterministic events during replay.

Instruction-counting logic. Instruction-counting logic implements precise instruction counters on top of hardware performance counters, which were originally intended to support performance analysis of the compiler and low-level system optimizations, and are only statistically correct.

Execution scheduler. The replay engine induces a VM to reproduce a recorded execution path by injecting each nondeterministic event at its instruction-accurate position in the instruction stream. The execution scheduler implements controlled execution of the system between nondeterministic events during replay. For synchronous events, the scheduling engine lets the system run until it reaches the point in execution at which it asks to replay that specific event.

To replay asynchronous events, the execution scheduler configures branch counters to raise an overflow interrupt before the original event takes place, and then continues execution of the system in a single-step mode. This is done to address a hardware delay in receiving the interrupt. Upon reaching the target place in the execution, the replay engine replays the asynchronous event and continues execution by scheduling execution of the system to the next nondeterministic event.

Debugging tools. A replay engine can aid manual analysis of nondeterminism with a run-time *page guarding* mechanism that write-protects pages of the replayed system on transitions to hypervisor. To further improve the analysis of divergent executions, the replay infrastructure needs to implement an efficient offline execution comparison tool using the branch tracing facilities provided by Intel and AMD CPUs. Both architectures provide a way to record all branches taken by the CPU in a memory buffer. Along with offline symbol resolution and trace-comparison tools, branch tracing provides a good mechanism for analyzing diverging executions. Finally, to detect divergent state in the hypervisor, the replay engine needs to be extended with a run-time mechanism that can record and compare the state of the VMM across original and replay runs.

4. Experiences with Deterministic Replay

We have implemented XenTT, a working deterministic replay prototype for the Xen 3.0.4 VMM. Careful design choices, as articulated in this paper's recipe, ensure that our implementation is efficient, maintainable, and extensible. XenTT can replay the execution of 32-bit, paravirtualized, uniprocessor Linux guests.

Performance summary.[4] Under send and receive network throughput tests, XenTT stays within 8%

[4] Our testing machine is a quad-core 2.4 GHz Intel Xeon E5530 "Nehalem" server with hyperthreading support, 12 GB of 1066 MHz DDR2 RAM, Broadcom NetXtreme II BCM5709 rev C 1 Gbps NICs, and several Western Digital WD1501FASS 1.5 GB SATA

and 14% of unmodified Xen, achieving throughput of 104 MB/s and 97 MB/s respectively. On serial disk write and read tests, XenTT stays within 21% and 23%, achieving throughput of 80 MB/s and 101 MB/s respectively. On an idle machine, XenTT's nondeterministic log grows at a rate of 4 GB/day (or 1 GB/day if compressed with the default gzip settings).

We have evaluated XenTT on systems workloads from the Phoronix benchmark suite. For the CPU-intensive benchmarks (LAME and GnuPG) XenTT remains within 2% of unmodified Xen; on the web-server workloads the recorded system stays within 5–31% of unmodified Xen (31% for Apache, 29% for Nginx, 9% for pybench, 5% for phpbench, and 23% pgbench); on the workloads with much random I/O XenTT's performance remains within 3% of the unmodified system for dbench, and 27% for postmark.

5. Related Work

Facing complexity of implementing precise hardware branch counting, most of existing process-level replay frameworks abandon support for replay of asynchronous events [10]. A general branch-counting module can reintroduce asynchronous events to process-level replay. QEMU-based replay solutions [4] simplify their implementation by relying on the inherently synchronized execution model of the QEMU emulator, which runs all structural parts, e.g., device and CPU emulation code, under the "big lock." As QEMU tries to depart from coarse-grained locking, those replay engines will require logging, locking, and orchestrating mechanisms similar to the ones suggested by our recipe. Facing challenges of the highly preemptive Xen environment, ReVirt [7] implemented many concepts similar to ours, but lacked a high-level abstract model and mechanisms that could help to translate its implementation to other VMMs.

6. Conclusion

We have presented a process and a software architecture—a recipe—for implementing deterministic record and replay facilities within VMMs. The contribution of this paper lies not in the implementation of a particular VMM replay engine, but rather in laying out the issues that are common to all VMM replay systems and presenting a solution that we believe can be followed to create efficient replay engines for practical hypervisors. Our recipe is the result of three person-years of effort spent implementing XenTT. We believe that if we had had a recipe to follow at the start of our XenTT project, our implementation effort would have been reduced. By sharing our battle-won experience, we hope to encourage other implementations and thereby promote deterministic replay as standard equipment for VMMs.

disks with a 64 MB buffer, 7200 RPM and a sustained data transfer rate of 138 MB/s. Replayed VMs run with 1GB RAM.

References

[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] M. Attariyan, M. Chow, and J. Flinn. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[3] T. Bergan et al. Deterministic process groups in dos. In *OSDI*, 2010.

[4] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *PPoPP*, 2013.

[5] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC*, 2008.

[6] G. Dunlap. Personal communication, 2012.

[7] G. W. Dunlap et al. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[8] G. W. Dunlap et al. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.

[9] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC*, 2005.

[10] O. Laadan et al. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS*, 2010.

[11] G. Lefebvre et al. Execution mining. In *VEE*, 2012.

[12] P. Montesinos et al. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS*, 2009.

[13] S. Park et al. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[14] VMware, Inc. VMware vSphere 4 Fault Tolerance: Architecture and Performance. White Paper, 2009.

[15] VMware, Inc. Protecting Hadoop with VMware vSphere 5 Fault Tolerance. White Paper, 2012.

[16] B. Weissman et al. Precise branch counting in virtualization systems. U.S. pat. 20090249049 A1. VMware, Inc., 2009.