Abstractions for Practical Virtual Machine Replay

Anton Burtsev David Johnson Mike Hibler Eric Eide John Regehr

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

Abstract

Efficient deterministic replay of whole operating systems is feasible and useful, so why isn't replay a default part of the software stack? While implementing deterministic replay is hard, we argue that the main reason is the lack of general abstractions for understanding and addressing the significant engineering challenges involved in the development of a replay engine for a modern VMM. We present a design blueprint—a set of abstractions, general principles, and low-level implementation details—for efficient deterministic replay in a modern hypervisor. We build and evaluate our architecture in Xen, a full-featured hypervisor. Our architecture can be readily followed and adopted, enabling replay as a ubiquitous part of a modern virtualization stack.

1. Introduction

In the last decade, deterministic replay went through a full cycle of a blooming research field—from rapid growth, to its peak, and arguably into decline. Numerous applications of deterministic replay were suggested: e.g., debugging and analysis of complex software systems [15, 26, 27, 32, 33, 35, 40, 41], fault-tolerant replication [9, 43, 44], performance analysis [4], and forensics [11, 19, 22]). A number of deterministic replay systems were developed along with advanced techniques for reconstructing execution of parallel [1, 12, 13, 38] and distributed systems. However, despite academic success, deterministic replay did not become a de facto part of systems and virtualization stacks.

As a default component of the modern VMM stack, ubiquitous deterministic replay could change the way we develop and analyze complex software 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, such as automatic debugging, explanation of cross-component performance anomalies, reconstruction of intrusion vectors, and more.

Why hasn't deterministic replay become a default part of the systems stack? Implementing deterministic replay is hard. We argue, however, that the main reason is the lack of general abstractions for developing replay mechanisms. In theory, system interfaces—OS system calls and VM hypercalls—are designed to provide a clean abstraction boundary and full encapsulation. In practice, abstractions are leaky due to a number of low-level optimizations aimed to deliver low-latency and high-throughput I/O for virtualized systems. In full-featured hypervisors like Xen and KVM, a replay interposition boundary built to capture the execution of a virtual machine cuts through multiple subsystems and layers of the software stack: hypervisor, host kernel, and host user-level. Without general abstractions for reasoning about nondeterminism, proper mechanisms for efficient recording, and tools for analyzing and debugging divergence, building replay into a full-featured hypervisor is impossible.

Abstractions for deterministic replay are badly needed. Existing replay prototypes either sidestep the complexity of a real environment and concentrate on a particular research topic, or develop an implementation that is challenging to generalize and reuse. It took the authors three person-years to implement a deterministic replay engine for uniprocessor guests running on Xen. Despite the existence of earlier replay implementations (one in Xen [23]), 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 collection of techniques and abstractions that provide a practical foundation for extending modern hypervisors with virtual machine replay. As our work is based on the experience of implementing deterministic replay in Xen, it fully reflects the complexity of a modern virtualization stack: parallel, asynchronous, and paravirtualized device I/O; a multi-layer device virtualization

Copyright © Anton Burtsev, David Johnson, Mike Hibler, Eric Eide, and John Regehr, 2016. This is the authors' version of the work. It is posted here for your personal use. Not for redistribution.

The definitive version was published in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Atlanta, GA, USA, Apr. 2016, http://dx.doi.org/10.1145/2892242.2892257

model; a fully preemptible, parallel hypervisor; and more. Our abstractions are clean and practical. We develop a general approach for capturing and coordinating the execution of a VM at multiple layers of the virtualization stack. We design general techniques for ensuring the determinism of larger nondeterministic components. Finally, our techniques keep recording mechanisms off the carefully optimized critical path of the hypervisor. We believe that deterministic replay is a useful part of the virtualization stack and that our blueprint can substantially simplify future replay implementations.

Although our work develops mechanisms essential for any replay engine, we focus on the replay of uniprocessor guest VMs. These are appropriate for many production use-cases where it is often acceptable to obtain scalability by running multiple guests on a powerful machine. A number of promising research efforts have addressed the problem of high overhead [13, 23] in recording multiprocessor guests. 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 [12], heavyweight execution-reconstruction techniques that are limited to several seconds of recording [1, 38], the inability to record a whole system due to strict limits on the amount of shared-memory nondeterminism [33], intrusive changes to the entire OS stack [7], or specialized hardware [29].

2. Deterministic VMMs Are Hard

The basis for deterministic replay is simple: the execution of a system is largely deterministic, and is only occasionally altered by external nondeterministic events. Being placed in identical initial conditions, and processing an identical instruction stream, the CPU deterministically generates the same values in registers and memory.[1] The determinism of execution holds until some external event, e.g., an external interrupt, or an I/O read from an external source, alters either the state of the CPU or the system's memory. Starting from the initial state, one can force the system to repeat its original execution by replaying external events.

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: the low-level state of the hypervisor (interrupt and exception handlers, timers, virtual CPUs and MMUs, and cross-VM shared memory), host device drivers (fully emulated and paravirtualized devices), a platform emulator (emulation of BIOS, legacy peripheral devices, and buses for unmodified guests), VM configuration and creation tools, and so on. Each of the parts contains some state of the VM and can affect its execution. Synchronized logging and orchestrated re-execution of the multi-level, multi-component system require abstractions providing a general approach to nondeterminism in a complex system, as well as mechanisms that can (1) record a complex decentralized system without loss of performance and (2) re-execute it in a controlled lock-step manner 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 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 [46]. 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.[2]

3. Abstractions and Mechanisms for Replay

The main challenge of implementing a replay platform in the complex, concurrent, multi-layer environment of a VMM is ensuring the determinism of execution: mediating all sources of nondeterminism and guaranteeing controlled execution of the system in a lock-step manner between pairs of nondeterministic events. Several abstractions are critical for addressing the complexity of this task: a three-part model of the environment, a general approach to implementing interposition functions, a simple locking and event atomicity model, and a general execution scheduler.

[1] There are anecdotal examples of nondeterministic CPU behavior [9].

[2] 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 [21].

3.1 A Three-Part Model

A three-part model provides a general view of possible nondeterministic updates that affect the execution of the system. The model represents the entire replayed system as three components (Figure 1): (1) the replayed system, (2) a deterministic execution environment, and (3) the nondeterministic external world. This representation simplifies the development of interposition and replay execution-scheduling layers by classifying all interactions between the replayed system and its environment into the following three categories: deterministic, synchronous, and asynchronous. Also, the three-part split reflects the fact that the seemingly rigid boundary of a virtual machine monitor in practice is pushed well outside the narrow hardware interface of a VMM for the following reasons: (1) the flexibility to choose the interposition boundary that reduces the amount of recorded nondeterminism, and (2) the possibility of reusing complex, low-level VMM code for injecting asynchronous events and implementing device I/O. Most of the hypervisor code—e.g., memory management, page-fault handling, and the hypercall interface—is deterministic and can be classified as the deterministic environment. This often allows ensuring determinism by recording and replaying an invocation of a high-level function from the VMM code (Figure 1(d)). 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, a *determinizing proxy*, that ensures the determinism of observed behavior (Section 3.2).

Deterministic updates A replayed system and its execution environment evolve together by updating each others' state (Figure 1(a)). For example, a replayed virtual machine starts through the normal VM-creation protocol that instantiates the VMM with a new VM (creates virtual CPUs, memory, paravirtualized or emulated device drivers, hardware emulator, etc.). In practice, it is simpler to ensure that the VM-creation protocol and its components are deterministic than it is to implement a new set of tools instantiating a replayed VM in a more controlled environment. Later, during VM execution, the VM updates the state of the deterministic environment, and vice versa. 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 in the same way they did in the original run.

Synchronous updates A guest system periodically invokes functions that might return nondeterministic results. For instance, reading the timestamp counter accesses nondeterministic data from otherwise deterministic code (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.

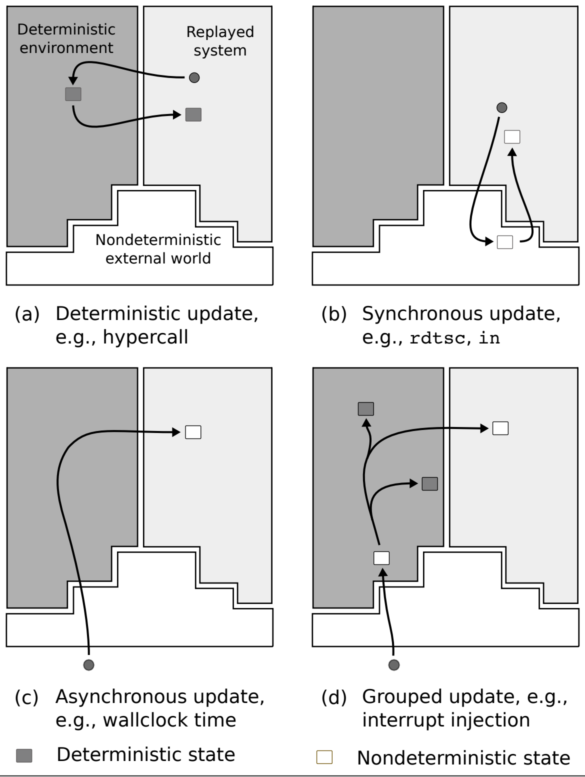
Deterministic environment
Replayed system
Nondeterministic external world
(a) Deterministic update, e.g., hypercall
(b) Synchronous update, e.g., rdtsc, in
(c) Asynchronous update, e.g., wallclock time
(d) Grouped update, e.g., interrupt injection
Deterministic state
Nondeterministic state

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

Asynchronous updates Asynchronous events represent external updates to the replayed system (Figure 1(c)). 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 carefully scheduling execution of a replayed VM.

Dependent updates 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 deterministic interrupt 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 ensure that interrupt handlers are atomic.

3.2 Interposition Functions and Determinizing Proxies

Interposition functions Interposition functions implement a general logging, replay, and filtering interface for nondeterministic events (Figure 2). During the original run, interposition functions record all nondeterministic updates. During

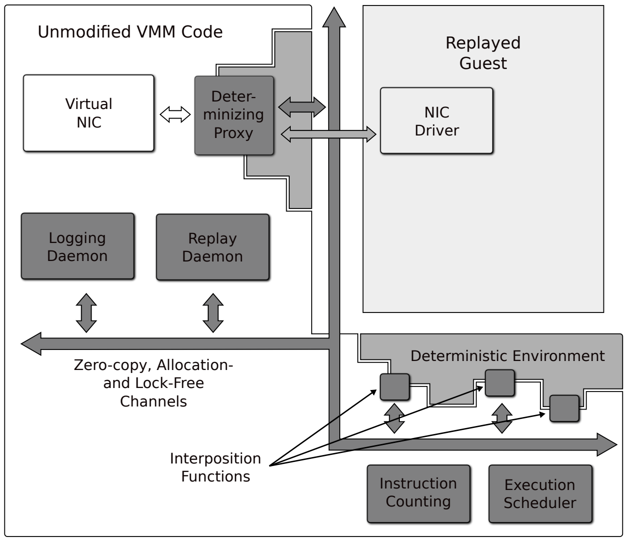

Figure 2. Components of the replay engine: pluggable interposition functions, logging and replay daemons, execution scheduler, instruction counting, and determinizing proxies.

replay, they serve two goals: (1) replay the original nondeterministic values and (2) prevent unscheduled nondeterministic events from updating the replayed system. Nondeterminism is generated by 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 prevent nondeterminism from leaking into the replayed system.

An interposition function follows the pattern shown in Listing 1. When the system is under replay and the current event is synchronous, it asks the replay scheduler to replay the current event. If the event is asynchronous, it is suppressed, but the replay scheduler can replay optional events (Section 3.5). During the original run, the function first pauses all the virtual CPUs of the VM, traces the event, and then emulates it by either emulating the original operation or invoking one of the original functions in the hypervisor code. Note that it is critical to trace the event before emulating it, as emulation might trigger more nondeterministic events. By following this pattern during replay, events will be replayed in order.

Different interposition functions can be invoked in different contexts of execution, e.g., hypervisor, host kernel, and host user-level. If the function is invoked from outside the hypervisor, it relies on fast communication primitives to implement atomic tracing and replay of events that are coordinated from inside the hypervisor.

Determinizing proxies Virtual devices are not part of 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, and order of replies. We use *determinizing proxies* to interpose on the communication protocol between the replayed

```
int trace_<function>(...) {
  event_t event = {<EVENT_NAME>, ...};
  if(replayed_guest()) {
    if(synchronous(&event)) {
      // request replay of a specific event
      replay_current_events(..., &event, &ret);
      return ret;
    }
    // asynchronous event: suppress the
    // update but replay "optional" events
    replay_current_events(...);
    return OK;
  }
  // Pause all virtual CPUs
  pause_vm();
  trace_event(<event_type>, ...);
  // Emulate original event
  ret = <function>(...);
  unpause_vm();
  return ret;
}
```

Listing 1. Generic example of an interposition function.

system and the device, ensuring that updates are propagated in a deterministic way. 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. The console device's proxy replays the console input itself. The more complex network device's proxy avoids emulation of the full device protocol. Instead, it replays the network payload into the guest device by using the device driver functions of the host kernel.

3.3 Precise Instruction Counting

A replay platform forces the replayed system to repeat its original execution by reproducing all nondeterministic updates to the state of the system and deterministic environment at the exact points of execution at which they happened during the original run. The position of each nondeterministic event is uniquely determined by the number of instructions executed by the replayed system since its start. While requiring compiler or binary translation support if done in software [36, 42], on modern CPUs, instruction counting can be implemented by utilizing the hardware performance monitoring interface [3, 31]. On the Intel architecture, two hardware performance counters can be utilized to implement an accurate instruction-counting algorithm: the branch instruction retired counter (BR_INST_RETIRED.ALL_BRANCHES) and the instruction retired counter (`INST_RETIRED.ALL`) [31]. A tuple of *{instruction pointer, branch counter}* is sufficient to identify the exact position in the instruction stream [24, 42]. On x86 systems, the tuple must be extended with the value of the `RCX` register, to account for cases when a string-copy instruction is preempted by an interrupt in the middle of the long copying loop. (The `RCX` register contains the last iteration of the loop.) Unfortunately, two problems complicate the implementation of a precise instruction counting: delay of the counter overflow interrupt, and nondeterminism.

Delay of the counter overflow interrupt The hardware performance monitoring interface of Intel CPUs provides support for preempting the execution of the system when a certain number of performance events is reached. In other words, it is possible to configure a performance counter to signal a nonmaskable interrupt when the counter overflows. Unfortunately, a nonmaskable counter overflow interrupt can be delayed for many cycles.[3] To address the interrupt delay problem, we configure the replay engine to preempt the execution of the system long enough in advance to account for the delay of the interrupt. After the interrupt is received, the system is single-stepped until the proper point in the instruction stream is reached.

Counter nondeterminism Both branch and instruction counters can become nondeterministic in the face of interrupts and exceptions. Specifically, certain instruction sequences change the behavior of the counter if preempted with an interrupt. In practice, this is a problem when the system is single-stepped for long periods during replay to compensate for the delay of the counter-overflow interrupt. During single-stepping, we implement a precise instruction-counting algorithm in software. If a counting anomaly occurs while the system is not in the single-step execution mode, we try to guess the correct value of the counter based on the value of the instruction pointer register. If the counter is only several instructions apart from the recorded value, we adjust it to match the value recorded during the original run.

3.4 Lightweight Interposition and Logging

Interposition code resides on the critical path of the guest system: inside interrupt and exception handlers, I/O paths of device drivers, and exit paths from the guest to the hypervisor. The main principle for implementing a fast interposition layer is to offload all tracing, processing, and saving of the trace data from the critical path. We implement this principle by utilizing a three-stage logging pipeline: pluggable interposition functions, ring channels, and a logging daemon (Figure 3).

The lightweight interposition layer must be designed to avoid memory allocations, data copying, and locking on the critical path. We implement our tracing mechanisms on top of a producer-consumer ring buffer. We allocate the memory

[3] On our hardware, the counter interrupt is typically delayed by only several instructions. We have seen cases, however, in which the delay is more than a hundred cycles.

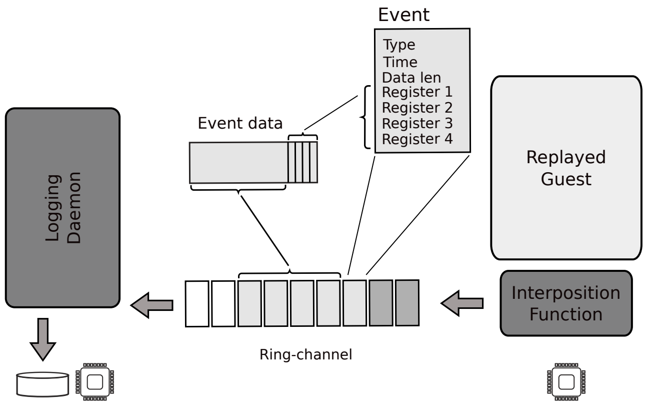

Figure 3. Lightweight interposition pipeline. Events and their payloads are allocated inside the ring.

for a new event record straight in the communication channel and log the event data into that memory. In the ring buffer, the pointer to the next element in the ring always points to the next available record, and thus it can be allocated by simply incrementing the pointer. Ring buffers are both lock-free and nonblocking; allocation, send, and receive operations are done with a single update of the producer and consumer pointers. To avoid blocking, a ring buffer provides flow control and tries to notify the receiver via an out-of-band mechanism when new records are available. For channels in which delay does not matter, ring channels notify the receiver only if the channel becomes critically full.

Atomicity of cross-CPU events To ensure the atomicity of recording nondeterministic events in a multi-CPU environment, we use *active messages*. We preempt and suspend the execution of the guest system. To record an asynchronous event between two physical CPUs—the CPU on which the event originates, and the CPU on which the recorded system is running—our primitives migrate the execution of the recording function between the CPUs. We request invocation on another CPU with an interprocessor interrupt (IPI). An IPI preempts execution of the guest system and invokes the requested function in the context of the IPI handler. Active messages allow us to avoid multiple cross-core round-trips required for suspending a VM.

Branch counter caching Frequent accesses to the relatively slow hardware branch-counter register introduce additional overhead when recording nondeterministic events. To minimize this cost, like ReVirt [22], our system accesses the hardware counter only once for each exit from the guest into the hypervisor.

3.5 Execution Scheduling

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 from one nondeterministic event to the next.

Proper design of the event scheduler, and careful choice of the event scheduling types, ensures the extensibility of the replay infrastructure—allowing it to be modified to record additional information about the system: e.g., debugging information, the state of hardware performance counters, and branch-tracing store events. A general execution scheduler implements support for the following types of events.

Synchronous and asynchronous events To replay synchronous events, the scheduling engine lets the system run until it reaches the execution point at which it must 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 [24, 37]. Upon reaching the target location in the execution, the replay engine replays the asynchronous event and continues execution by scheduling the execution of the system to the next nondeterministic event.

Optional events Optional events are useful to implement best-effort service. A scheduler will replay an optional event if it observes that the recorded guest is at a position in the instruction stream at which the event occurred in the original run; otherwise, it will discard the event. For example, we rely on the optional event type to replay performance information (Section 4.5) and turn on and off heavyweight debugging features like hardware branch tracing (Section 3.6).

Nonreplayable events Finally, nonreplayable events can record arbitrary information, e.g., debugging and performance analysis primitives like `printf`, tracing of real-time performance information during original and replay runs, and collecting BTS logs (Section 3.6).

Retyping asynchronous events as synchronous The replay engine benefits from recording as many synchronous events as possible. The replay of synchronous events does not require slow single-stepping—the system can just run to a point at which it exits into the hypervisor. Typically, several asynchronous events (e.g., device ring buffer updates) will be recorded while the guest system is inside the hypervisor that is servicing this synchronous exit. Although the synchronous exit itself does not need to be recorded since it is deterministic, it is possible to use the information about the synchronous exit to relabel all asynchronous events inside this exit as synchronous.

Replay on exit to guest It is reasonable to assume that the timestamp of the guest system changes only while it is running. In practice, this assumption is not true. The logic of instruction emulation implemented inside the hypervisor can change the instruction pointer of the guest system, moving its timestamp forward. Therefore, a scheduler should check if more events are ready to be replayed right before exiting into the guest system.

3.6 Scaling Development with Replay Analysis Tools

In our experience, the scalable development of deterministic replay is impossible without a range of debugging and analysis tools to aid the analysis of nondeterminism and the debugging of subtle replay divergence cases. Three mechanisms aid the development of replay: page guarding, hardware branch tracing, and run-time state comparison.

Page guarding Our system's run-time *page guarding* mechanism enables the detection of unrecorded updates to the guest system. Page guarding write-protects guest pages when the guest system exits into the hypervisor. To implement the guard, we extended the hypervisor to automatically protect and unprotect pages on every transition between the guest system and the hypervisor. Page guarding enables us to detect a large subset of all nondeterministic events at the processor and memory boundaries of the virtual machine interface.

Hardware branch tracing Any undetected nondeterminism or error in the replay implementation might *diverge* the execution of a guest system from its original run. Without additional information, subtle replay failures are difficult to debug. We implemented an efficient execution comparison tool using the Branch Tracing Store (BTS) facility provided by Intel CPUs [31]. The BTS interface allows us to configure a region of memory as a linear array in which the CPU records all taken branches. The BTS can be configured to send a nonmaskable interrupt when the array reaches a certain length. We flush the contents of the BTS buffer every time it traces a nondeterministic event. This way, we are able to see what code the system was executing between nondeterministic updates, and compare BTS traces across original and replay runs. We further built a set of tools that resolve raw branch addresses into human-readable symbol names. We rely on the GDB debugger to perform the symbol lookup. The BTS mechanism, coupled with automatic symbol resolution and trace-comparison tools, has proved to be a good mechanism for analyzing diverging executions.

Run-time state comparison Finally, to detect divergent state and nondeterminism in the hypervisor, the replay engine contains a run-time mechanism that can record and compare the state of the hypervisor across original and replay runs. We implement state comparison by extending our replay engine with a new optional event. This event carries the state information between original and replayed runs, and triggers a state comparison when it is replayed.

4. Deterministic Replay in Xen

We implemented our ideas in XenTT, a full-system deterministic replay engine for the Xen virtual machine monitor. Multiple reasons motivated the choice of Xen as the basis for XenTT. Xen is a full-featured, open-source virtualization platform [5]. It offers excellent virtualization performance; a fast, fully asynchronous, paravirtualized device driver architecture; and support for a wide variety of guest systems

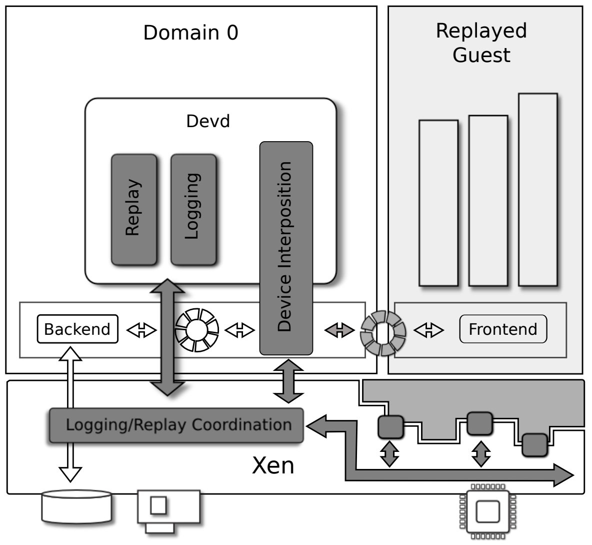

Figure 4. Architecture of the XenTT replay engine.

and hardware platforms. It is widely used as a virtualization provider in commercial datacenters [2] and large-scale academic research facilities [47].

4.1 XenTT Architecture

XenTT implements our replay architecture in Xen (Figure 4). Its replay engine consists of four main components and several high-bandwidth communication channels that connect them. The event-interposition layer utilizes pluggable interposition functions to implement logging and replay of the low-level VM interface exported by the hypervisor. The coordination layer relays events between the interposition functions and the logging and replay daemons. Logging and replay daemons run as user processes inside the privileged Xen domain; they process the log of recorded events and save it to a persistent store. The device-interposition layer implements determinizing proxies for each logged and replayed Xen device.

4.2 Hardware-level Virtual Machine Interface

Several Xen components require modification to record and replay nondeterministic events; we discuss them here.

Start info page The `start_info` page is shared between the guest system and the hypervisor at boot time. XenTT leverages the determinism of the domain-creation protocol, which recreates values in the start info page during replay.

Shared info page The `shared_info` page contains information required for initialization of the guest, delivery of interrupts (event channels), nanosecond and wall-clock time, etc. Shared info is updated asynchronously by the hypervisor. XenTT records and replays nondeterministic updates to the shared info page as simple memory-page updates.

Grant tables The grant tables store information for memory access permissions and in-flight sharing of pages between domains. Grant tables are typically updated asynchronously by the backend drivers. A grant table update has the form of a compare-and-exchange or a clear-flag operation. For compare-and-exchange, XenTT records a grant table operation as an index into the grant table array, the old value, the new value, and the result of the compare-and-exchange.

Event channels Event channels are Xen's analog to hardware interrupts. They are one-bit communication primitives used to send immediate notifications between virtual machines. To deliver an event, Xen preempts the execution of the guest, creates a special interrupt stack, and forces the execution of the interrupt handler. Although the event-delivery protocol requires several updates to the `shared_info` page, and the injection of an interrupt frame into the guest, its execution is deterministic. XenTT records and replays event notifications by simply invoking the event delivery function (`evtchn_set_pending`).

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

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

EFLAGS register To single-step the guest during replay, XenTT uses the trace flag (TF) in the EFLAGS register of the guest. Obviously, TF can change the execution of the system during replay, if it is "leaked" into the guest. For example, TF changes the execution of the Linux kernel on the system-call entry path. To preserve the determinism of the guest, XenTT virtualizes the EFLAGS register. During replay, when the guest is single-stepped, XenTT parses the guest's instruction stream and detects instructions that save the EFLAGS register.

Optional events Some exits from the guest are in-place events (i.e., exceptions, hypercalls, and int 0x80 system calls) that are deterministically re-executed by the guest, if the determinism of all other events is preserved. To reduce interposition overhead, XenTT does not record these events.

Time Xen exports wall-clock time, system time since boot, and run-time state statistics to the guest system through a shared memory region that Xen updates periodically. To obtain the most recent time values, the guest system interpolates time values with nanosecond precision by reading a hardware timestamp (TSC) register that is used to compute the time passed since the last memory update. To ensure the determinism of time values, XenTT records updates of time values in the shared page, and implements emulation of the `rdtsc` instruction that accesses the timestamp counter. XenTT records and replays the guest's run-time state statistics that reflect the

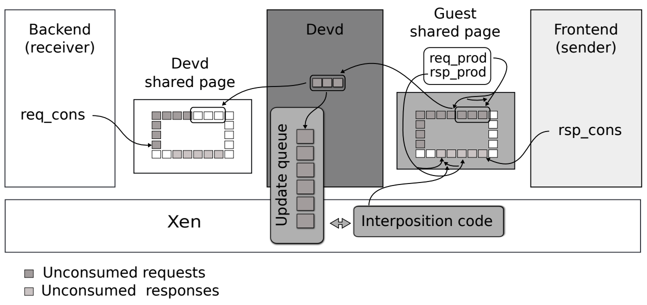

Figure 5. Details of ring interposition.

amount of time a guest system spends in each of four states: running, runnable, blocked, and idle. Finally, XenTT records the execution of periodic, polling, and single-shot timers.

4.3 Device Drivers

The major source of nondeterminism in any system is communication with external devices. Under Xen, a guest system accesses its virtual devices via a backend-frontend split device pair [14, 25]. To notify each other about new I/O requests, backend and frontend device drivers rely on a fast, lock-free, producer-consumer ring, which they share in a memory page. The backend and frontend devices add new requests to the ring by simply advancing the pointers in the shared ring, and sometimes notifying the other end via an event channel.

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

There are two challenges in logging device driver communication. First, the overhead of logging every update to memory shared between virtual machines is prohibitive. XenTT leverages the semantics of the shared ring and logs only shared-ring pointer updates. (In theory, a guest could access the data in the shared memory before the pointer update; in practice, guest frontend device drivers do not.)

The second challenge is a result of the fact that a shared ring is updated inside the kernel of a device driver domain. XenTT must record the exact state of the guest at the update, but this state is only available inside the hypervisor. To avoid multiple context switches needed to read guest state, XenTT implements a new technique that ensures atomicity of the memory update *and* reading the guest state. Instead of updating a pointer in the ring, the device driver domain sends the hypervisor an active message (Section 3.4) describing the update. The hypervisor atomically performs the update and records guest state.

4.4 Determinizing Proxies

To preserve the determinism of replay, XenTT ensures the determinism of updates to the shared-ring buffers. To control all shared-ring updates from the backend devices, XenTT implements a device-driver interposition component: `Devd`, which is inserted between each pair of backend and frontend device drivers to mediate their communication (Figure 5). `Devd` implements determinizing proxies for the four most critical Xen devices: console, XenStore, disk, and network.

`Devd` is implemented as a user-level application that runs inside the device driver domain, i.e., Domain 0 in a typical Xen setup. `Devd` implements a device bus: by monitoring the XenStore database, `Devd`'s bus driver discovers new devices connected to the guest system. For each newly discovered device, `Devd` walks through the list of registered drivers and tries to find a match for the device.

Instead of connecting to the event channels and shared rings of the frontend domain, backend devices connect to the shared rings created by `Devd`. `Devd` is transparent to the communication between backend and frontend devices: backend and frontend devices update their ring pointers as they do in case of noninterposed execution, and `Devd` reflects all updates between the two rings it mediates. In Figure 5, `Devd` mediates the update of the request producer pointer.

In-kernel payload logging Replay of the console, XenStore, and network devices requires logging of the data entering the guest during the original run. To shorten the datapath for high-throughput network devices, XenTT implements a payload logging mechanism, which allows it to save the payload of a backend device straight into a file without leaving the context of the device driver domain kernel.

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

4.5 Extending Replay with Accurate Performance Information

Deterministic replay ensures that the replayed copy of execution is identical to the original run; i.e., during replay, the system repeats its original execution instruction by instruction. However, the performance information about the original execution is lost. Compared to the original run, the replayed system progresses at different speeds during replay. The two main factors that affect the performance of the replayed system are (1) eliminated periods of I/O waits and CPU inactivity and (2) frequent periods of single-stepping required for replay of asynchronous events.

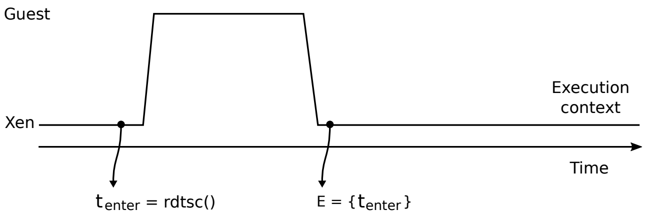

Figure 6. Saving and piggybacking the TSC information on exit to guest.

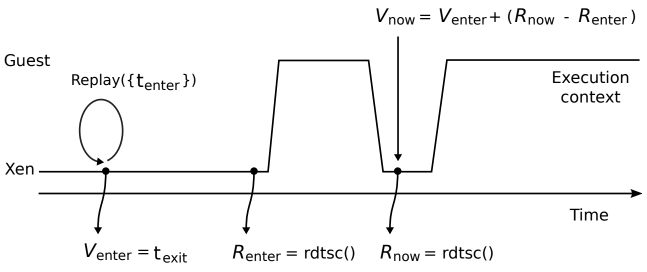

Figure 7. Replaying the TSC information.

XenTT extends a traditional replay protocol with the ability to record and recreate a faithful view of performance of the original run during replay. During original execution, XenTT periodically records the value of the timestamp counter register (TSC). During replay, recorded values provide a basis for accurate approximation of performance at any moment of replayed execution with the help of a simple linear model.

XenTT records the value of the TSC on every transition from the hypervisor into the recorded system. The following optimization allows XenTT to make time recording more accurate. Every time the hypervisor is about to enter the guest, XenTT saves the current value of the TSC. The recording of this value is delayed until the guest returns to the hypervisor (Figure 6). During replay, the value of the original timestamp counter is replayed on every transition from the hypervisor into the guest (Figure 7). The current values of the TSC right at the entry point (R_{entry}) and at the moment the performance information is queried (R_{now}) allow XenTT to recreate the original value of the timestamp counter (V_{now}) as $V_{now} = V_{enter} + (R_{now} - R_{enter})$.

5. Evaluation

Can our replay abstractions and mechanisms be applied for recording complex software systems? In this section, we present several evaluation scenarios that demonstrate XenTT's ability to perform deterministic recording of real systems, on a variety of workloads, with overheads that are non-prohibitive for use in production environments.

Hardware setup We conducted our evaluation on a hardware platform that is representative of a production cloud environment. We performed experiments on a Dell PowerEdge R710 server equipped with a quad-core 2.4 GHz Intel Xeon E5530 "Nehalem" processor with hyperthreading support, 12 GB of 1066 MHz DDR2 RAM, and four Broadcom NetXtreme II BCM5709 rev C 1 Gbps NICs. The machine is configured with four Western Digital WD1501FASS 1.5 GB SATA disks with a 64 MB buffer, 7200 RPM, and a sustained data transfer rate of 138 MB/s. One of these disks is the root disk. XenTT is based on a development version of the 32-bit Xen hypervisor (the closest corresponding Xen release is 3.0.4) and a 32-bit Linux 2.6.18 kernel.

To minimize test variability, we configured the system with the minimal set of resources required to fulfill the test task. For all experiments, we configured the Xen hypervisor to recognize only three CPU cores: two cores are allocated for Domain 0, and one core runs the XenTT guest VM. To reduce caching and buffering effects, we reduce the memory allocation for Domain 0 and the guest VM to be 2 GB and 192 MB, respectively.

5.1 Is Replay Faithful?

How do we know if the replayed system repeats the original execution? Deterministic replay naturally implements a self-checking mechanism through the replay of synchronous events. During replay, the system periodically asks for the replay of a synchronous event. At this point, the replay platform must provide the original value for a specific event. Replay detects divergence if the log does not contain the requested event, or if the timestamp of the event differs from the current position in the execution of the replayed system. In practice, synchronous events are frequent enough to provide a high degree of confidence that replay is faithful. A malicious system could possibly confuse the replay engine about its state (e.g., if replay is used for malware analysis or virtual machine accountability [28]). In these cases, hardware branch tracing can be used to compare executions at the level of taken branches.

5.2 Logging and Replay Overheads

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

Figure 8 presents our results from running Freebench benchmarks on a Linux guest and on a XenTT guest with recording enabled. For all but one test, the recording infrastructure introduced only a small overhead of 5.6%.

To evaluate the performance of a recorded system on a set of systems workloads, we configured XenTT to record the execution of the Phoronix benchmarking suite. The Phoronix suite provides a large library of benchmarks; we use nine that characterize whole-system performance and stress specific hardware components.

Figure 9 shows the performance of XenTT relative to the reference, non-XenTT performance, for each benchmark. LAME, GnuPG, and Stream remain within 2% of the perfor-

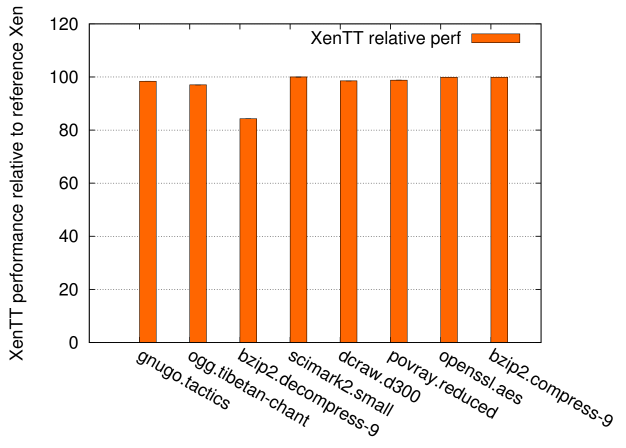

Figure 8. Freebench benchmarks.

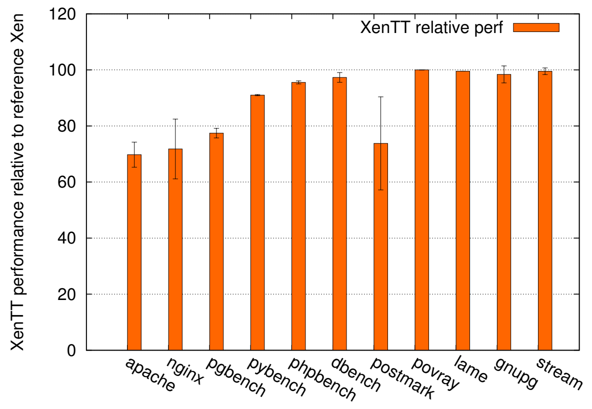

Figure 9. Phoronix benchmarks.

mance of an unmodified Xen. Under XenTT, `apache` serves approximately 69% as many requests per second as the reference implementation. The `apache` benchmark incurs a large number of accesses to the timestamp counter register (TSC), via the `rdtsc` instruction. In an unmodified Xen, `rdtsc` is a non-privileged instruction. XenTT, however, forces the guest system to exit on `rdtsc` to record the returned TSC value. It is unclear if the large number of TSC events is inherent to the Apache workload or simply a side-effect of the benchmark. DBench and Postmark perform a large number of random disk operations. Higher delays of the disk interposition code penalize their performance.

Log sizes We evaluated the space required to store deterministic logs by running XenTT on several representative tasks. A Linux kernel boot incurs a large number of exits to the hypervisor and nondeterministic events. XenTT logs performance information on every VM exit. An idle XenTT system generates a raw log at a rate of 167 MB/hr (4 GB/day), or 44 MB/hr (1 GB/day) if compressed with gzip. For the TCP network receive test, we report both nondeterministic event and payload logs. We do not report the compressed size of the payload log, since it is payload-dependent.

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

Table 1. Log size for various workloads.

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

Disk-intensive workloads To stress sequential disk I/O, we used `dd` to read and write a 4 GB file (Figure 10). We used a 1 MB block size and averaged results over three runs. The Xen disk drivers provide little buffering on the I/O path and are therefore sensitive to the delays that XenTT introduces by routing all disk requests through a user-level device-interposition daemon. On a disk with a sustained data transfer rate of 138 MB/s, a vanilla Xen system achieves write and read throughputs of 101 MB/s and 130 MB/s, respectively. The XenTT interposition layer stays within 21% and 22% of the performance of unmodified Xen.

Network-intensive workloads We evaluated network logging overhead by recording the execution of the `netperf` network benchmark (Figure 11). We set a TCP window size of 128 KB, ran `netperf` for 60 s, and averaged results over three runs. Compared to disk I/O, Xen's network drivers are more highly optimized to support high-bandwidth workloads and tolerate I/O delays. On send and receive operations, XenTT is able to stay within 8% and 14% of unmodified Xen.

To measure network delay, we used XenTT to record the execution of the `ping` tool (Figure 12). The tests report the overhead of interposition for both idle and loaded network paths. To load the network path, the guest system runs a `netperf` TCP stream test in the background, concurrently to the ping test. On a loaded connection, the measured delays for XenTT are within 5% of those measured for Xen. On an idle link, XenTT's interposition code adds an 80 μs delay.

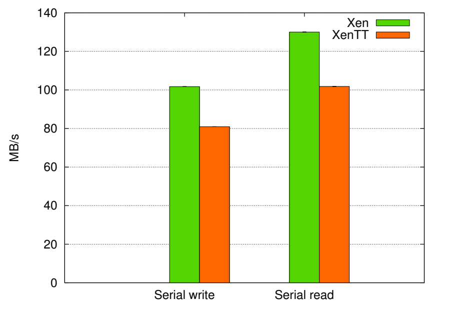

Figure 10. Disk throughput.

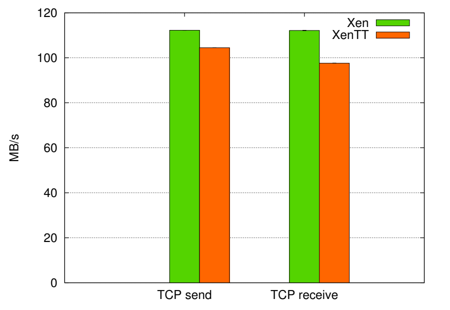

Figure 11. Network throughput.

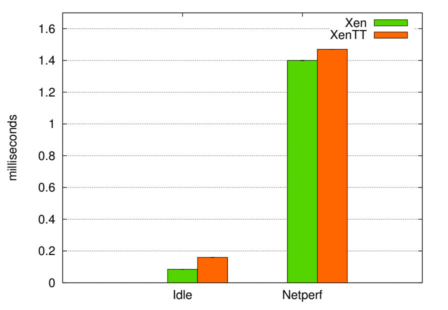

Figure 12. Network delay.

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

Table 2. Precision of the performance model.

5.3 Precision of the Performance Model

To characterize the precision of XenTT's performance model, we measured simple sequences of instructions. Table 2 lists our results. The "nopX" tests consist of *X* `nop` instructions; the "realX" tests consist of *X* simple instructions (e.g., `pushl`, `inc`, `addl`, etc.). To improve measurement fidelity, we removed the possibility of page faults while loading the test code pages. In the table, the "Native" cycle counts are obtained by directly instrumenting the test using the `rdtsc` instruction; the "VMI/Perf Model" values are obtained from the XenTT performance model via a virtual machine introspection (VMI) interface exported by XenTT at the beginning and end of the test sequence. The "Error" values are the ratios of VMI to Native cycle counts.

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

6. Discussion

Although we implement deterministic replay for paravirtualized Xen guests, we argue that our ideas generalize to other hypervisors and types of virtualization. Paravirtualization and hardware-supported CPU virtualization require inherently similar replay interposition boundaries. A paravirtualized interface is designed to follow the shape of the hardware interface of the CPU—a pragmatic choice aimed at minimizing the changes necessary in the guest kernel. Compared to paravirtualization, hardware-supported virtualization defines a much cleaner and simpler protocol for injecting asynchronous interrupts into the guest system. As a result, the replay interposition boundary can rely on the same principles for injecting guest interrupts (i.e., determinism of low-level hypervisor functions), but does not require assembly programming to invoke interposition functions from an interrupt return path.

Full virtualization extends hardware-supported virtualization of the CPU with full emulation of a platform. This requires extending the interposition boundary into a hardware emulator: QEMU is a de facto standard emulator used by both Xen and KVM. Until recently, QEMU had a single-threaded, serialized device-emulation architecture that enabled relatively simple interposition and replay [12]. Components of a hardware emulator that are not performance-critical, such as BIOS emulation, can leverage this simple architecture, and thus can be trivially extended with replay. On the other hand, the emulation of high-throughput hardware devices would require fully asynchronous, parallel devices. Such devices will benefit from the techniques developed by our work: determinizing proxies, device interposition, fast communication primitives, centralized recording, active messages, and general replay scheduling. These same techniques can be applied to the replay of virtualization platforms that support direct device assignment and SR-IOV. In the case of direct assignment of a device, instead of full emulation of the device interface, a replay platform must implement a simpler interposition logic for the low-level device interface (the PCI configuration space, BAR regions, DMA engine, etc.).

KVM implements a hypervisor as part of the Linux kernel. While different than Xen, KVM is built on architectural ideas similar to those in Xen, and thus can benefit from the general mechanisms and principles presented here. Furthermore, we argue that our replay debugging and analysis techniques—page guarding of the VMCS region, BTS tracing, and runtime state comparison—can significantly aid in the analysis of nondeterminism. While originally dependent on QEMU-based device emulation, recent versions of KVM leverage virtio [39], a fast, paravirtualized device stack. Similarly to split-device drivers in Xen, virtio relies on shared-memory ring buffers for high-throughput communication. While having a different ring interface, virtio devices can leverage our ideas for implementing general device interposition.

7. Related Work

Techniques to log and replay state date back to the earliest computing systems. In 1948, the ENIAC relied on system checkpointing to recover computations interrupted by frequent component failures [34]. An excellent overview of the early work in the area of reversible execution is given by Leeman [34]. Several surveys provide taxonomies of early [20] and more recent work [16, 30] in the area of replay debugging. Chen et al. provide a good overview of recent approaches to multiprocessor replay [13]. Contemporary notions of system replay originated as parts of distributed checkpoint protocols [6], replay debuggers for parallel systems [18], and fault-tolerant replication approaches [8].

A major drawback of all early replay systems is their inability to handle asynchronous events. Mellor-Crummey and LeBlanc were the first to implement an instruction-counting algorithm in software [36]. Cargill and Locanthi were the first to advocate the implementation of a simple instruction-counting mechanism in hardware [10]. Bressoud and Schneider relied on one of the first hardware branch-counting implementations in their hypervisor-based, full-system replication solution [9]. Precise instruction counting on modern hardware is still a problem [37]. VMware has articulated the details of a precise branch-counting algorithm that operates in the face of System Management Mode (SMM) interrupts [46].

A process-level replay solution can be implemented inside [17, 41] or outside [26, 27, 40] of an operating system kernel, i.e., as an extension to a kernel or as a library within a process. Facing the complexity of implementing precise hardware branch counting, most existing process-level replay frameworks do not provide support for replaying asynchronous events [33]. Mozilla's rr is a notable exception that implements instruction counting and replay of asynchronous events for a process-level replay solution [37].

In contrast to process-level solutions, full-system replay can provide a complete implementation of replay that requires no simplifying assumptions about the replayed system, except those imposed by the virtualization platform [12, 19, 22, 48]. A commercial replay implementation from VMware can record and replay the 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 [43, 44].[4] QEMU-based replay solutions [12] 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 orchestration mechanisms similar to the ones suggested by our recipe. Facing the challenges of the highly preemptive Xen environment, we implemented many concepts similar to those found in SMP-ReVirt [23, 24], which however lacked high-level abstractions and mechanisms that could help programmers to translate its implementation to other VMMs.

8. Conclusion

We have presented principles and mechanisms for implementing deterministic replay in a modern VMM. The contribution of this paper lies not in the implementation of a particular replay engine, but rather in laying out the issues that are common to all virtual machine replay systems and presenting a solution that we believe can be followed. Our architecture is the result of three person-years of effort spent implementing XenTT. We believe that if we had had a reference architecture to follow at the start of our XenTT project, our implementation effort would have been much smaller. By sharing our battle-won experience, we hope to encourage other implementations and thereby promote deterministic replay as standard equipment for VMMs.

Acknowledgments

We thank the anonymous VEE '16 reviewers for their helpful comments. We performed our experiments on machines provided by the Utah Emulab testbed [47]. This work was supported in part by the Air Force Research Laboratory and DARPA under Contract No. FA8750–10–C–0242. This material is also based upon work supported in part by the National Science Foundation under Grant Number 0524096.

References

[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proc. SOSP*, pages 193–206, Oct. 2009. doi:10.1145/1629575.1629594.

[2] Amazon Web Services, Inc. Amazon EC2 – virtual server hosting, 2016. URL https://aws.amazon.com/ec2/.

[3] AMD Corporation. AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2007.

[4] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production soft-

[4] Starting with version 8, VMware Workstation dropped support for replay debugging functionality. For a while, VMware continued using deterministic replay as a basis for their fault-tolerant VM replication solution in vSphere. However, vSphere 6.0 abandoned deterministic replay in favor of fast checkpointing [45].

ware. In *Proc. OSDI*, Oct. 2012. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/attariyan.

[5] 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 *Proc. SOSP*, pages 164–177, Oct. 2003. doi:10.1145/945445.945462.

[6] J. F. Bartlett. A nonstop kernel. In *Proc. SOSP*, pages 22–29, Dec. 1981. doi:10.1145/800216.806587.

[7] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proc. OSDI*, pages 177–192, Oct. 2010. URL https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Bergan.pdf.

[8] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proc. SOSP*, pages 90–99, Oct. 1983. doi:10.1145/773379.806617.

[9] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. In *Proc. SOSP*, pages 1–11, Dec. 1995. doi:10.1145/224056.224058.

[10] T. A. Cargill and B. N. Locanthi. Cheap hardware support for software debugging and profiling. In *Proc. ASPLOS*, pages 82–83, Oct. 1987. doi:10.1145/36177.36187.

[11] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *Proc. OSDI*, pages 541–554, Oct. 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chen_ang.

[12] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proc. PPoPP*, pages 207–218, Feb. 2013. doi:10.1145/2442516.2442537.

[13] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen. Deterministic replay: A survey. *ACM Comput. Surv.*, 48(2), Nov. 2015. doi:10.1145/2790077.

[14] D. Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, first edition, 2007. ISBN 978-0132349710.

[15] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. USENIX ATC*, pages 1–14, June 2008. URL https://www.usenix.org/legacy/event/usenix08/tech/full_papers/chow/chow.pdf.

[16] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. D. Bosschere. A taxonomy of execution replay systems. In *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[17] F. Cornelis, M. Ronsse, and K. De Bosschere. TORNADO: A novel input replay tool. In *Proc. PDPTA*, 2003.

[18] R. Curtis and L. D. Wittie. BUGNET: A debugging system for parallel programming environments. In *Proc. ICDCS*, pages 394–400, Oct. 1982.

[19] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In *Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 66–71, Oct. 2006. doi:10.1145/1181309.1181320.

[20] C. Dionne, M. Feeley, and J. Desbiens. A taxonomy of distributed debuggers based on execution replay. In *Proc. PDPTA*, Aug. 1996.

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

[22] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI*, pages 211–224, Dec. 2002. URL https://www.usenix.org/legacy/event/osdi02/tech/dunlap.html.

[23] G. W. Dunlap, D. G. Lucchetti, P. M. Chen, and M. A. Fetterman. Execution replay for multiprocessor virtual machines. In *Proc. VEE*, Mar. 2008. doi:10.1145/1346256.1346273.

[24] G. W. Dunlap III. *Execution Replay for Intrusion Analysis*. PhD thesis, University of Michigan, 2006.

[25] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. 1st Workshop on Operating System and Architectural Support for the On Demand IT Infrastructure (OASIS)*, Oct. 2004. URL https://www.cl.cam.ac.uk/research/srg/netos/papers/2004-safehw-oasis.pdf.

[26] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proc. USENIX ATC*, pages 289–300, May–June 2006. URL https://www.usenix.org/legacy/events/usenix06/tech/geels.html.

[27] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proc. OSDI*, pages 193–208, Dec. 2008. URL https://www.usenix.org/legacy/events/osdi08/tech/full_papers/guo/guo.pdf.

[28] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proc. OSDI*, pages 119–134, Oct. 2010. URL https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Haeberlen.pdf.

[29] N. Honarmand and J. Torrellas. RelaxReplay: Record and replay for relaxed-consistency multiprocessors. In *Proc. ASPLOS*, Mar. 2014. doi:10.1145/2541940.2541979.

[30] J. Huselius. Debugging parallel systems: A state of the art report. MTRC Report 63, Mälardalens University, Västerås, Sweden, Sept. 2002. URL http://www.es.mdh.se/publications/366-Debugging_Parallel_Systems__A_State_of_the_Art_Report.

[31] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C, and 3D): System Programming Guide, 2015.

[32] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX ATC*, pages 1–15, Apr. 2005. URL https://www.usenix.org/legacy/events/usenix05/tech/general/king.html.

[33] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. SIGMETRICS*, pages 155–166, June 2010. doi:10.1145/1811039.1811057.

[34] G. B. Leeman, Jr. A formal approach to undo operations

in programming languages. *ACM TOPLAS*, 8(1):50–87, Jan. 1986. doi:10.1145/5001.5005.

[35] G. Lefebvre, B. Cully, C. Head, M. Spear, N. Hutchinson, M. Feeley, and A. Warfield. Execution mining. In *Proc. VEE*, pages 145–158, Mar. 2012. doi:10.1145/2151024.2151044.

[36] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proc. ASPLOS*, pages 78–86, Apr. 1989. doi:10.1145/70082.68189.

[37] Mozilla Foundation. rr: lightweight recording & deterministic debugging, Feb. 2016. URL http://rr-project.org/.

[38] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proc. SOSP*, pages 177–192, Oct. 2009. doi:10.1145/1629575.1629593.

[39] R. Russell. virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS OSR*, 42(5):95–103, July 2008. doi:10.1145/1400097.1400108.

[40] Y. Saito. Jockey: A user-space library for record-replay debugging. In *Proc. AADEBUG*, pages 69–76, Sept. 2005. doi:10.1145/1085130.1085139.

[41] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proc. USENIX ATC*, pages 29–44, June–July 2004. URL https://www.usenix.org/legacy/event/usenix04/tech/general/srinivasan.html.

[42] G. Venkitachalam, M. Nelson, B. Weissman, M. Xu, and V. V. Malyugin. Using branch instruction counts to facilitate replay of virtual machine instruction execution. U.S. patent 7,844,954, Nov. 2010.

[43] VMware. VMware vSphere 4 Fault Tolerance: Architecture and performance. White paper, Aug. 2009. URL https://www.vmware.com/resources/techresources/10058.

[44] VMware. Protecting Hadoop with VMware vSphere 5 Fault Tolerance. Technical white paper, Aug. 2012. URL https://www.vmware.com/resources/techresources/10301.

[45] VMware. VMware vSphere 6 Fault Tolerance: Architecture and performance. Technical white paper, Dec. 2015. URL https://www.vmware.com/resources/techresources/10514.

[46] B. Weissman, V. V. Malyugin, P. Vandrovec, G. Venkitachalam, and M. Xu. Precise branch counting in virtualization systems. U.S. patent 9,027,003, May 2015.

[47] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002. URL https://www.usenix.org/legacy/event/osdi02/tech/white.html.

[48] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, June 2007. URL https://labs.vmware.com/academic/publications/retrace.