Execution Replay for Multiprocessor Virtual Machines

George W. Dunlap, Dominic G. Lucchetti, Peter M. Chen
Electrical Engineering and Computer Science Dept.
University of Michigan
Ann Arbor, MI 48109-2122
{dunlapg,dluccet,pmchen}@umich.edu

Michael A. Fetterman
University of Cambridge Computer laboratory
15 JJ Thompson Avenue, Cambridge, UK, CB3 0FD
Michael.Fetterman@cl.cam.ac.uk

Abstract

Execution replay of virtual machines is a technique which has many important applications, including debugging, fault-tolerance, and security. Execution replay for single processor virtual machines is well-understood, and available commercially. With the advancement of multi-core architectures, however, multiprocessor virtual machines are becoming more important. Our system, SMP-ReVirt, is the first system to log and replay a multiprocessor virtual machine on commodity hardware. We use hardware page protection to detect and accurately replay sharing between virtual cpus of a multi-cpu virtual machine, allowing us to replay the entire operating system and all applications. We have tested our system on a variety of workloads, and find that although sharing under SMP-ReVirt is expensive, for many workloads and applications, including debugging, the overhead is acceptable.

Categories and Subject Descriptors C.4 [*Computer Systems Organization*]: Performance of Systems — Measurement Techniques; D.4.1 [*Operating Systems*]: Process Management — multiprocessing

General Terms Design, Measurement, Performance, Reliability, Security,

Keywords ReVirt, execution replay, multithreading, determinism, race recording, multiprocessors, virtual machines, Xen, direct memory access, SPLASH, page protections

1. Introduction

Execution replay gives the ability to reconstruct the past execution of a system. In conjunction with a checkpoint of the system state, it gives the ability to reconstruct the entire state at any point in time over the replay interval. This ability is useful for several different applications. For debugging, it allows a programmer to inspect the execution and state of a particular run of a system, even in the face of non-determinism[9, 14, 5, 8]. For security, it allows a system administrator to go back and inspect the entire state of the system before, during, and after an attack, allowing the system administrator to determine how the attack took place and observe the attacker's activities[6]. For fault tolerance, execution replay allows the state of a system just before a crash to be recovered without the need for frequent checkpoints[7, 4, 11]. Recent work has also used execution replay to efficiently collect and store software architectural traces[19].

A simple way to apply execution replay to a wide range of software is to implement execution replay for virtual machines[4]. Running softare in a virtual machine capable of being replayed allows a user to take advantage of execution replay without needing to modify software running inside the virtual machine. It also has the advantage of being able to use execution replay on an operating system kernel.

In order to implement execution replay in a virtual machine, any non-deterministic event that affects the virtual machine's state must be recorded. This state includes all memory allocated to the virtual machine, the processor registers, and the disk. For a single processor system, non-deterministic events include any external input (such as keyboard, mouse, or network), as well as the timing of non-deterministic events like interrupts. The techniques for replaying single processor systems are well understood, and are even available commercially[19].

With the increasing prevalence of multi-core processors, execution replay on multiprocessor systems has become more important. Implementing replay for multiprocessor systems is much more challenging than single processor systmes, however. Because writes on one processor can affect reads on another processor, the results of memory races must be recorded and replayed. Existing solutions require modification to software, or massive modifications to hardware.

We have built a system, SMP-ReVirt, which is the first system to log and replay multiprocessor virtual machines on commodity hardware. In order to detect and replay the results of memory races, we use hardware page protections, available on all modern desktop and server processors. This technique allows us to log and replay unmodified multiprocessor systems, including multiprocessor kernels running inside of a virtual machine.

Logging makes sharing more expensive, but the end-to-end impact on performance varies widely depending on the workload. For some applications it is prohibitively expensive, while for others there is little impact.

This paper explores execution replay for multiprocessor virtual machines. Section 2 introduces the basic concepts, terms, and requirements of execution replay for single processor virtual machines. It then discusses the complications that shared-memory systems introduce, and describes techniques to address them. Section 3 describes the research prototype we built using the Xen hypervisor, describing the implementation of the general principles in more detail, and describing some of the technical issues involved. In Section 4, we evaluate our research prototype, investigating the

VEE'08, March 5–7, 2008, Seattle, Washington, USA.

source of overhead and sharing. Finally, Section 5 discusses related work.

2. Execution Replay

Logging and replay is widely used for recovering state. The basic concept is straightforward: start from a checkpoint of a prior state, then roll forward, replaying events from the log to reach the desired state. The type of system being recovered determines the type of information that needs to be logged: database logs contain transaction records, file system logs contain file system data, and so on. Replaying a virtual machine requires logging the non-deterministic events that affect the virtual machine's computation. These log records guide the virtual machine as it re-executes (rolls forward) from a checkpoint. Most events are deterministic (e.g. arithmetic, memory, branch instructions) and do not need to be logged; the virtual machine will re-execute these events in the same way during replay as it did during the original execution.

In order to replay an execution, we simply log and replay any non-deterministic event that affects the state of the system. For virtual machines, this includes logging virtual interrupts, input from virtual devices such as the virtual keyboard, network, or real-time clock, and the results of non-deterministic instructions such as those that read the processor's time-stamp counter (TSC).

There are two aspects of an event which may be non-deterministic: data, and timing. We call an event which is non-deterministic in data an *input event*. An instruction which reads a processor's TSC is an example of an input event. The result of the read is non-deterministic; but the timing of it is *synchronous* – that is, it always happens at the same point in the instruction stream. To replay these events, the replay system needs to log and replay the data changed by the event.

An event which is non-deterministic in timing is called an *asynchronous* event. A virtual interrupt is an example of an asynchronous event. The state change caused by an interrupt is deterministic (writing certain values on the processor's stack and changing certain registers), but the point in the instruction stream where the interrupt is delivered is non-deterministic.

To replay asynchronous events, an execution replay system needs to be able to identify the exact point in the instruction stream where the event occurred, and replay the event at the same point in the instruction stream during replay. In order to do this, we utilize the hardware branch counter available on several architectures, in conjunction with the instruction address. The observation is that if a given virtual address is executed twice, there must be a branch between them. Using branch counters allows us to identify a particular instruction in the instruction stream at which the asynchronous event occurred, so that we can re-deliver the event at the same point during replay.

Note that an event may be both asynchronous and an input event. An example of such an event is DMA from a virtual device, where both the timing of the DMA, and the data written by the DMA, must be logged and replayed for the system work correctly.

Input from devices, such as keyboard and network, must be logged and replayed; however, output, such as writes to a console or sending network packets, do not affect state and do not need to be logged or replayed. The data sent will be re-generated by the replaying system. This data may be discarded without affecting the reconstruction of the state of the virtual machine. However, it is frequently useful to involve those devices in replay.

Other devices, such as the disk, allow us a choice. We could simply log all reads from the disk; but this typically generates a prohibitive amount of data, even for a moderately short run. Instead, we can avoid logging input from the disk by including it in the replaying system. If we checkpoint and restore the disk along with the rest of the state of the system, writes to the disk will be re-generated, which causes reads to return the same data as during logging. Thus we can make reads from disk deterministic without logging them.

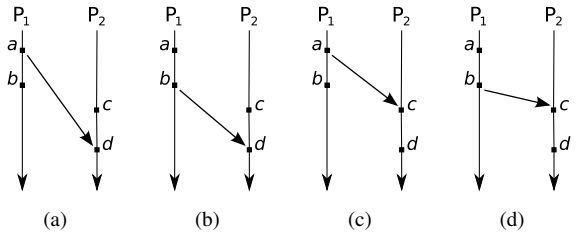


Figure 1. Constraints sufficient to guarantee the order $a \rightarrow d$

2.1 Replaying shared-memory systems

When replaying shared-memory systems, reads from memory by one processor are affected by writes of another processor. Since these reads and writes may happen in any arbitrary interleaving, this introduces fine-grained non-determinism into any shared memory operation.

In order to reconstruct the state of shared memory, each processor must view writes to shared memory by other processors as asynchronous events[1]. We therefore need to preserve the *order* of execution between the processors. We do not need a strict instruction-by-instruction ordering, however. Only reads and writes to shared memory need to be ordered with respect to each other. More specifically, two instructions need to be ordered only if both of the following are true:

- They both access the same memory.
- At least one of them is a write.

This is the *ordering requirement*. Any interleaving of instructions during replay that satisfies the ordering requirement will result in the same execution[2].

We indicate that instruction a is ordered before instruction b by $a \rightarrow b$. This is read, "a happens-before b". In order to enforce an order between two processors, we introduce *constraints* between instructions. A constraint $a \rightarrow b$ indicates that the replay system will ensure that b does not execute until a has executed.

Two points on the instruction stream may be ordered even if there is no direct constraint from one to the other. Within a single processor, there is an implicit ordering, based on the order the instructions were executed. Furthermore, ordering is transitive: $a \rightarrow b$ and $b \rightarrow c$ implies $a \rightarrow c$.

Consider Figure 1. Suppose that a and d are writes to the same memory, but that b and c are unrelated—$a \rightarrow d$ is the only ordering necessary. One constraint sufficient to guarantee the ordering is $a \rightarrow d$. But any of the following constraints would imply the order $a \rightarrow b$ as well:

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

If b and c are unrelated, we say that the constraints above are *over-constrained*, because they cause the replay system to run more strictly than necessary: either P_2 must wait until P_1 reaches b

[1] We can instead view reads from shared memory as synchronous data input events. This is the method taken by BugNet[12], discussed in Section 5.

[2] See [10] for a more complete exploration of the concept of order and state in a distributed system.

(although the data was ready at a), or P_2 stops and waits at c (although it is not necessary to stop and wait until d), or both. Over-constraining reduces the potential parallelism during a replay run, but can be taken advantage of to reduce the number of constraints or simplify logging.

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

A logging system for a shared memory system must generate a set of constraints that will satisfy the ordering requirement, but is free to choose any set of constraints that will meet that ordering.

To detect which memory operations need to be ordered, we implement a *concurrent-read, exclusive-write* (CREW) protocol between *virtual cpus* in a multiprocessor virtual machine. This technique for detecting constraints was first introduced by [9]. The CREW protocol stipulates that each shared object may be in one of the following two states:

- *concurrent-read*: All cpus have read permission, but none have write permission.
- *exclusive-write*: One cpu (called the *owner*) has both read and write permission; all virtual cpus have no permission.

Each read or write operation to shared memory is checked for access before executing. If a virtual cpu attempts a memory operation for which it has insufficient access, the CREW system must make requests to the other processors to decrease their permissions, so that it may increase its own. We call these increases and decreases in permissions *CREW events*.

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

In order to check for access of shared memory reads and writes, we use hardware page protections, available on all modern desktop processors. Hardware page protections are enforced by the *memory management unit*(MMU), which will check each read and write as the instruction executes, and cause a fault to the hypervisor on any violation. Because the checks are done in hardware, the common case is very fast. It also allows us to interpose on reads and writes without modifying the software running on the guest.

Generating constraints from CREW events is straightforward. Each CREW fault will cause two CREW events: a privilege increase on one processor, and a privilege decrease on another processor. If a is the point of privilege decrease, and b is the point of privilege increase, then the constraint $a \rightarrow b$ will be sufficient to order any reads and writes associated with this CREW event.

To see why this is so, consider a particular interaction between two processors, P_1 and P_2. Suppose that instruction b at P_2 writes to a page which is in concurrent-read mode. This instruction will cause a fault into the CREW subsystem. The CREW system will then reduce the privileges of P_1, and increase the privileges of P_2. Let us call the instruction this privilege-decrease happens instruction a. Processor P_1 has had read permission from the time it received permission through point a. Any instruction during that time may have read the page that b is about to write. We cannot tell when the last access was using page protections, but we know that it will be before a by program order, so constraining $a \rightarrow b$ will give us the ordering we need.

Constraining on privilege-reduction events rather than on the last read does mean that our replay will be over-constrained. This is because we do not have access to when the last read or write to the page actually occurred; any instruction executed before the privilege-reduction event could have accessed the page.

2.2 Direct Memory Access

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

Replaying DMA presents some difficulties. In DMA, a device acts as another processor with respect to memory transactions. A single processor system with DMA-enabled devices is effectively a multiprocessor system from replay's perspective. However, unlike peer processors in an SMP system, the devices do not have an MMU that we can use to interpose on accesses[3]. How are we to involve devices in the CREW protocol?

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

If the device follows this type of transaction model, where the device will only access memory between certain well-defined boundaries, and the cpu does not need to access memory to the device until a transaction is completed, and if the hypervisor can interpose and understand the commands from the guest to the device and the device's responses, we can model the device as a *non-preemptible actor* in the CREW protocol. A non-preemptible actor does explicit acquire and release of pages before and after a transaction, rather than acquiring them on demand and having them preempted, as preemptible actors such as virtual cpus do. When a cpu issues a DMA command to the device, the hypervisor informs the CREW protocol, which acquires the appropriate privileges on behalf of the device (either concurrent-read or exclusive-write, depending on the transaction). When the device informs the cpu that the transaction is done, the hypervisor informs the CREW protocol, which will release access on behalf of the device.

If any virtual cpu tries to access a page in a way that is incompatible with the CREW privileges of some device on a system, the CREW system must block its execution until the device has finished the transaction associated with that page. In this way, we do not need to rely on the correctness of kernels or device drivers inside the virtual machine; only on the correctness of the hardware.

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

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

3. Prototype system

3.1 The Xen Hypervisor

In order to test the effectiveness of using hardware page protections to detect sharing, we modified the Xen hypervisor[2]. Xen supports *paravirtualized* guests: that is, the guest kernel is modified to use the interface provided by the hypervisor for privileged instructions, rather than the hypervisor emulating the interface of native hardware. Xen's interface was designed from the ground up with speed of paravirtualization in mind, which allows Xen to run kernel-intensive benchmarks (usually the worst case for virtual machines) at speeds near native[2].

Xen guests use *hypercalls* to perform privileged operations. These operations include memory operations, setting trap tables and interrupt gates, accessing debugging hardware, switching stack pointers and segment registers, and so on. Xen uses a shared page for passing certain kinds of information between the guest and Xen; this is called the *shared-info* page. Some of this is information that requires privileges to access directly from the hardware, and may be read often by the guest: for instance, the speed of the CPU, what kinds of memory and hardware functionality are available, and the system time. The shared-info page also has information about pending and blocked virtual interrupts.

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

Unprivileged domains do not have direct access to hardware devices. Instead, they use virtualized devices provided by domain 0. Access to these virtual devices is paravirtualized as well. The driver running inside the unprivileged guest is called the *front end*. The corresponding part which provides the paravirtualized device from domain 0 is called the *back end*. The front end marshals requests to the virtual device from the unprivileged domain and sends them to the back end. The back end then satisfies these requests through domain 0's kernel and its access to physical devices. The front end also grants access to the back end for any memory associated with the transaction. This granting is the Xen analog of DMA. Standard devices include virtual console, disk, and network devices.

3.2 Replaying Xen

The hardware interface of Xen has few surprises, in regards to non-determinism. The results of hypercalls are deterministic; these results do not need to be logged and replayed. The results of instructions which read the processor's time-stamp counter (TSC) must be logged and replayed, and the timing of virtual interrupt deliveries must be logged. Each virtual cpu has its own log, out of which it replays non-deterministic events.

Under normal circumstances, Xen gives a guest direct read access to the pagetables. This is problematic for replay for several reasons, the most important of which is that during logging, the permission in the hardware pagetables for a given page may be less than what the guest has set for that page. Thus when the guest kernel walks its own pagetables, it may get confusing results.

Instead, we use a feature of Xen called *shadow pagetables* for replaying guests. The guest's pagetables are virtualized and not used directly by the MMU. Instead, the hypervisor creates copies, or *shadows*, of the guest's pagetables, which are used by the actual hardware. By introducing this level of indirection we lose some performance, but we gain an abstraction that is much easier to work with. Each virtual cpu on a multi-virtual-cpu guest domain can have its own shadow of a shared pagetable, and any reductions in protections due to the CREW protocol are not visible to the guest.

All memory allocated to the guest is visible by all guest processors; so any changes to this memory must be involved in the CREW protocol. Each virtual cpu and device is a different actor in the CREW protocol. Each actor has a *CREW event count* that is incremented on every CREW event (any increase or decrease in permissions). A particular CREW event is represented as a tuple of the actor and the CREW event count.

There are three sources of reads and writes that must be involved in the CREW protocol: guest read and write instructions, virtual device DMA, and hypervisor access to guest state. For each type of access, we must determine how we will detect accesses, enforce CREW permissions, and replay constraints. Note that we modify the hypervisor and virtual devices in order to enforce CREW permissions and enable correct logging and replay. We do not, however, require any cooperation from a guest. We discuss each of these sources of reads and writes in turn in the following sections.

3.3 Guest access to shared state

Guest read and write instructions are detected using the hardware page protections. Constraints are generated and logged at privilege-increase events. Privilege-decrease events generate *CREW count increment* events in the log. These events allow the system to reproduce the CREW event count at the proper point during replay, which is necessary to determine when a constraint has been satisfied. Both constraint events and CREW count increments are asynchronous events. Using the MMU allows us to interpose on all guest accesses without any assumptions about the software running in the guest.

During logging, before a privilege-increase event can happen, corresponding privilege decreases and the resulting CREW count increment logs must happen on other processors. Doing this properly requires some care. The log must be taken on the other processor, because certain information (such as the registers and hardware performance counters) are only directly available on that physical cpu while it is running. Furthermore, the TLB must be flushed there as well, to make sure that modifications made to the pagetables are actually reflected in the TLB of the processor.

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

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

3.4 Virtual device access to shared state

Virtual devices are implemented as non-preemptible actors. In order to log and replay virtual devices, we instrument the back-end of

the device in domain 0. The back-end does an explicit acquire and release of read or write permission from the CREW system for any pages involved in the transaction before fulfilling the request. The "acquire" corresponds to the privilege increase, and the "release" corresponds to the privilege decrease. Each device has its own log, in which it records CREW count increases, constraints, any information necessary about the interaction with the guest to be able to replay properly, such as the order in which block requests were satisfied. During replay, the back-end will make calls into the CREW system to inform it of CREW count increments and to wait for constraints to be satisfied. It will also replay any appropriate device interaction with the guest.

The kinds of information logged and replayed depends on the device. For the console driver, the data from console input is logged and replayed. Output is re-executed; the guest device's reads from memory are ordered by constraints generated by the CREW protocol, so we are guaranteed that at the point of read, the memory is in the same state.

When logging the block device, we do not log the data read from the disk. Instead, we restore the disk to its original state, and re-execute both read and write requests during replay. During replay, a request is not passed to the back-end until the constraint has been met. The constraints guarantee that the same data goes from the virtual machine to the disk.

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

Logging and replaying the network driver has yet not been implemented.

3.5 Hypervisor accesses to guest state

Hypervisor access to guest state requires special consideration, for several reasons. First, the hypervisor's execution changes between logging and replay. Secondly, the hypervisor makes private mappings of guest memory which are not subject to the CREW protocol. Finally, because the hypervisor does not save its stack on a context switch, it cannot effectively block. This section discuss these issues, and how we solved them.

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

- When performing hypercalls on behalf of the guest
- The shadow code reads guest page table entries in order to generate shadow tables. It also sets dirty and accessed bits of page-table entries.
- Updating data on the shared-info page.
- Virtual interrupt delivery. The hypervisor needs to read the shared-info page to determine pending and blocked interrupts. The hypervisor writes the interrupt frame on to the guest stack.

There are two distinct problems to be solved: interposing on access to guest state, and replaying the ordering of multiple accesses within the hypervisor.

Interposing on hypervisor access to guest state is fairly straightforward. Hypercalls and writes to the guest stack use the guest's virtual address space. If it reads or writes a page for which the CREW has reduced permissions, it will fault and go through the CREW handler just like any guest access. We therefore need to do nothing to interpose on these accesses. Other hypervisor accesses, such as reads and writes to the guest pagetables and accesses to the shared-info page, use the hypervisor's private mapping. For these accesses, we instrument the code to acquire the page required on behalf of the virtual cpu that caused the access.

Because virtual cpus are preemptible actors, we need to ensure that access is not preempted before the hypervisor access finishes. We therefore require that the shadow lock be held continuously from the time the page is acquired until the access to the page is complete. Since the shadow lock must be acquired to gain permission, this guarantees that no other cpu can remove the code's permission before its operation is complete.

Replaying the ordering of accesses within the hypervisor poses a more difficult problem. Consider the following potential races:

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

Unfortunately, it is difficult to log and replay asynchronous events (including constraints) within the hypervisor. The main reason is that since the hypervisor is doing the logging and replaying, execution within the hypervisor is necessarily different between logging and replay. Determining where to deliver the interrupt using instruction counters becomes impossible in this case. Furthermore, unlike the Linux kernel, the hypervisor has no per-vcpu stack; if it calls `schedule()`, it loses all its context. So the only acceptable form of blocking while waiting for a constraint to be satisfied is to spin, an action which could lead to deadlocks.

We considered many potential solutions to this problem, but in the end they added a lot of complication to an already complicated system, with the only gain being more parallelism between hypercalls. What we would like is to avoid hypervisor races entirely, treating all hypervisor operations as atomic operations, so that if a hypervisor operation (such as a hypercall) accesses the same memory as an instruction or another hypervisor operation (and at least one of them is a write), the one will happen either entirely before or entirely after the other.

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

The hypervisor lock allows us to treat a hypercall as an atomic unit for ordering purposes. Any constraints generated during the hypercall are pushed logically to the beginning of the hypercall. Any CREW events will be pushed until after the end of the hypercall. (These modifications are consistent with the rules for over-constraining, discussed in Section 2.1.)

The hypervisor lock solves both hypercall-hypercall races and hypercall-guest races. Consider the hypercall-guest race mentioned in the list above. Before the virtual cpu can read the modified version of page A, it must get read access; but to get read access, it

has to complete a shadow fault. The fault cannot execute until after the hypercall has completed. So any virtual cpu reading A or B will either see the state before the hypercall or the state after, but not in the middle.

Using the hypervisor lock for hypercalls reduces the potential parallelism if two hypercalls happen at the same time, but this is an unusual case. In our measurements, the vast majority of the overhead measured at this point is related to the CREW subsystem (including waiting for the shadow lock); very little is attributable to a hypercall waiting for a hypercall lock.

4. Evaluation of multiprocessor ReVirt

4.1 Workloads

To evaluate how well traditional SMP workloads run under ReVirt, we ran the SPLASH2 benchmark suite from Stanford University[16]. This is a well-studied suite of computationally intensive parallel applications designed to evaluate the design of parallel processors. Most of the tests have parameters or input values that can be set. The test comes with a set of default parameters and input; however, it was tuned to state-of-the-art systems of over ten years ago. Using those parameters on modern processor, most applications finished in a small fraction of a second. This is not enough time to distinguish the actual workload from start-up effects. We chose input parameters such that the tests ran for around 60 seconds. The tests we ran were FMM, LU, ocean, radiosity, radix, and water-spatial[4].

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

- *kernel-build*: parallel build of the Linux kernel. This is a build of the stock Linux kernel with the default configuration. We use `gcc` version 3.4.5, and Linux kernel version 2.6.17. In order to make this a parallel workload we used the `-j` option of `make`, which tells `make` how many outstanding child processes to try to keep at one time (maintaining build dependencies). Our experience indicates that `-j n+1` produces the optimum throughput on a domain with `n` virtual cpus. The extra process allows efficient overlap of computation and I/O.
- *dbench*: a filesystem benchmark for Linux that's meant to emulate a workload that a Linux Samba server might generate under the NetBench Windows file server benchmark. As one might expect, the workload is almost entirely in the kernel.

4.2 Workload characteristics

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

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

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

Woo et al. [16] did a comprehensive study of the SPLASH2 benchmarks on idealized hardware to learn how different hardware parameters, such as cacheline size and latency, affected their runtime. The key workload characteristics include concurrency, working set size, communication to computation ratio, and spatial locality.

Using hardware page protections effectively changes two parameters. The first is the granularity of sharing, which goes from a 16-byte cacheline to the 4096-byte page size. The second is the increased latency from a miss. Our early tests indicated that a remote cacheline miss on our hardware was around 400 cycles; the average time for a CREW fault is over 40,000 cycles. Therefore we can expect that workloads that are prone to false sharing for large cacheline sizes will have unnaturally high amounts of communication, causing performance to suffer. We also expect that workloads with a naturally high communication rate regardless of cacheline size will suffer from the increased latency.

Increasing the granularity of sharing can have two effects, depending on the particular workload. In workloads with lower spatial locality, increasing the granularity of sharing can increase the false sharing, thus increasing the overall communication required. In workloads with high spatial locality, on the other hand, increasing the granularity of sharing can "coalesce" what would be individual faults or misses into one large one, reducing the amount of communication required.

This has similar but subtly different effects on caches than on execution replay. On caches, an increase in false sharing increases both data and communication overhead (thus reducing performance), while the "coalescing" effect decreases the number of cacheline misses by effectively prefetching data.

SMP-ReVirt is not responsible for actual data transfer, but instead for permission and generating constraints. So although false sharing will raise the amount of communication, it will not increase the cache data transferred between cpus, nor will it reduce cache misses due to spatial locality. There are analogs, however. Increased granularity can cause increased false sharing, and thus more time overhead and an increased log size. But the "coalescing" effect for workloads with high spatial locality may reduce the number of constraints needed, compared to cacheline-based logging.

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

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

4.3 Results and analysis

The main purpose of this section is to investigate the properties of using page protections to detect sharing. We want to know the following:

- For each workload, how much overhead does logging with page protections generate? This overhead includes time overhead (how much it slows down compared to a non-logging system), and space overhead (how much disk space is required to hold the log).
- Is it worth adding virtual processors to try to increase performance? Does it increase performance, and if so, how much?

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

Workload	Logging rate (compressed)	Fills 300GB disk in
FMM	.234 GB/day	1280 days
LU	.238 GB/day	1261 days
ocean	.232 GB/day	1295 days
radix	.292 GB/day	1025 days
water-spatial	.231 GB/day	1296 days
kernel-build	.562 GB/day	534 days
radiosity	.232 GB/day	1295 days
dbench	.557 GB/day	1280 days

Table 1. Space overhead of logging a single processor guest.

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

Table 2. Space overhead of logging a two processor guest.

- Where is the sharing? Is it in the kernel, which we may be able mitigate by paravirtualizing in future implementations, or in the application, where we are unlikely to be able to make many changes?

To distinguish logging in general from logging with page protections, we first consider the overhead of logging and replaying uniprocessor guests. Figure 2 shows the normalized runtime for ReVirt for uniprocessor workloads in Xen. All workloads are run with only one thread. Kernel build has approximately 12% overhead. Some of the workloads of the SPLASH2 suite have approximately 5% overhead, and most have negligible overhead. Table 1 shows the space overhead. We show the log size in compressed gigabytes per day, and the time to fill a dedicated 300GB logging disk. These numbers are similar to those found in [6].

Figure 3 shows the normalized runtime of SMP-ReVirt for a two processor system, compared to the same test running on unmodified two processor systems. We also include a single processor logging system, as a reference to determine if adding a second processor gains any performance advantage. In general, the number of processes for each test is equal to the number of processors in the virtual machine. The exception is kernel build on multiprocessor guests, where the number of processes is one plus the number of processors, to make more efficient overlap of processing and disk I/O. Table 2 shows the space overhead.

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

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

The majority of the overhead is traceable directly to sharing, measured in the number of faults. Figure 4 presents the sharing rate for the two processor guest, broken down by kernel- and process-level, in faults per second. We calculate the sharing rate by dividing the total number of faults in the logging run by the time of the unmodified Xen guest run. (It may seem more natural to divide the logging fault count by the logging runtime. However, dividing by logging runtime gives misleading results. Large fault counts are made to look less severe by the very overhead they produce. Dividing by the unmodified guest runtime treats the fault rate as a property of the workload.)

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

Table 3. Space overhead of logging a four processor guest.

We can see from Figure 4 that LU and ocean have very little sharing at all; the little sharing that occurs is in the kernel. This is consistent with what we expected from our knowledge of the workload. Water-spatial and FMM have a large amount of process-level sharing. Because FMM and water-spatial have minimal kernel sharing, their total sharing is low enough to beat a single processor system. Radiosity has large amount of process-level sharing, and an even larger amount of kernel-level sharing. Nearly all of the sharing in dbench and kernel-build comes from the kernel.

Figure 5 shows the overhead on a 4-cpu system. The system included two dual-core Xeons (four cores in all). The graph shows the unmodified domain, as well as 4-cpu and 1-cpu logging runs on the system. The graph also includes 2-cpu runtimes from the other hardware, normalized to the 1-cpu logging run on this machine, so that we can see the impact on runtime as we add virtual cpus. We did not run radiosity and dbench. Table 3 summarizes the space overhead.

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

Figure 6 shows the sharing rate, broken down by kernel- and process-levels. We can compare the results qualitatively to Figure 4, keeping in mind that they were run on different systems. FMM has considerably more sharing on the four processor guest; about six times as much for process-level and fourteen times for kernel-level. Radix's process-level sharing is comparable to the two cpu system, but its kernel-level sharing is a more than four times longer. Water-spatial's process-level sharing is about the same for two and four cpus, and although there is twice as much kernel-level sharing, its overall effect is still small.

We have verified the validity of our logging system by replaying all of these workloads as well. Execution replay is extraordinarily sensitive in the presence of asynchronous events: the slightest deviation in state or execution path will cause the next asynchronous event delivery to fail. Furthermore, in the process of developing the replay system, we developed many tools to verify the state and execution of the system. We are therefore confident that our system accurately duplicates the state and execution of the original run.

5. Related work

The idea of using a virtual machine to achieve the benefits of execution replay without needing to modify the software running on it was first proposed by Bressoud, et al[3, 4]. His system uses execution replay to enable a high-availability *primary-backup* system. The main system (or primary) is logged, and the logs fed to one

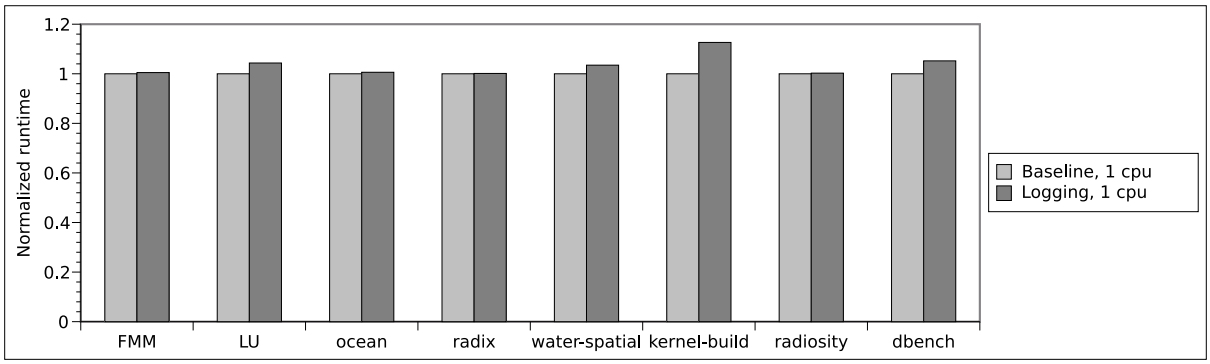
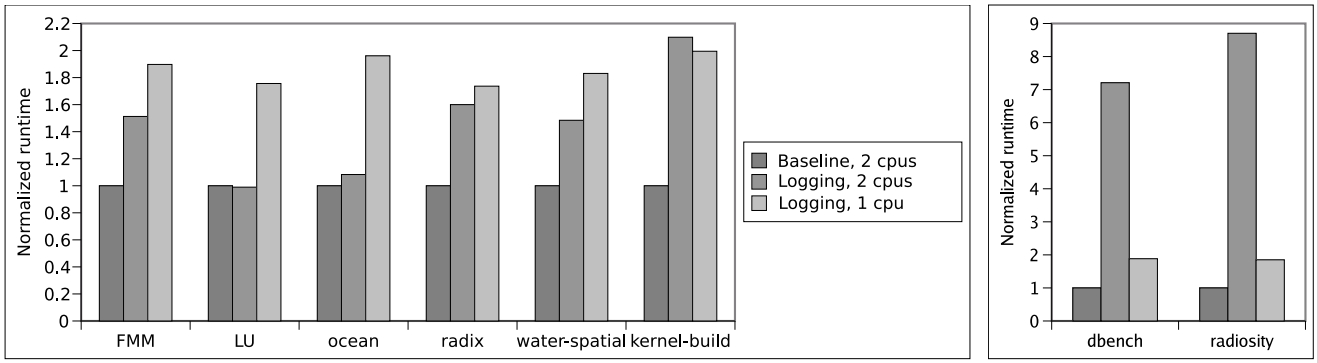


Figure 2. Overhead of ReVirt for a single processor Xen guest



(a) (b)

Figure 3. Overhead of ReVirt for a two processor Xen guest

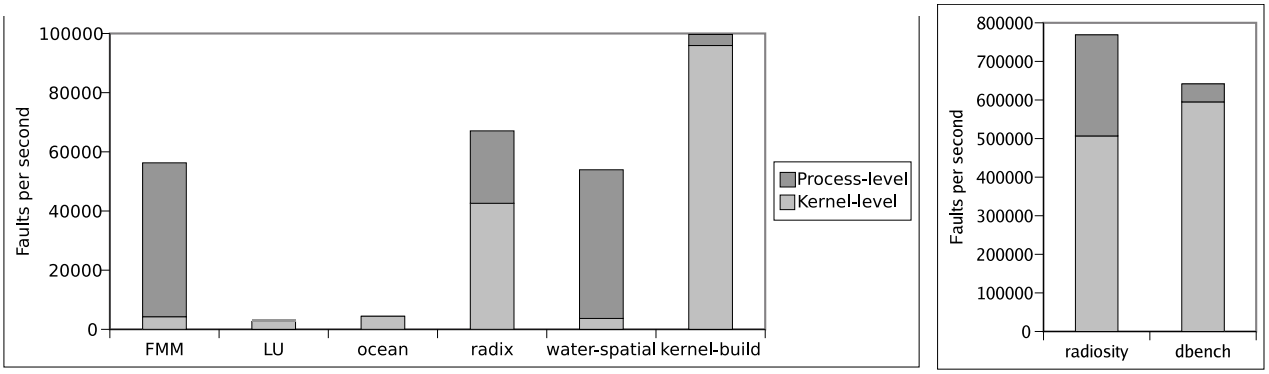


Figure 4. Kernel- and process-level sharing rates, in faults per second, for a two processor guest

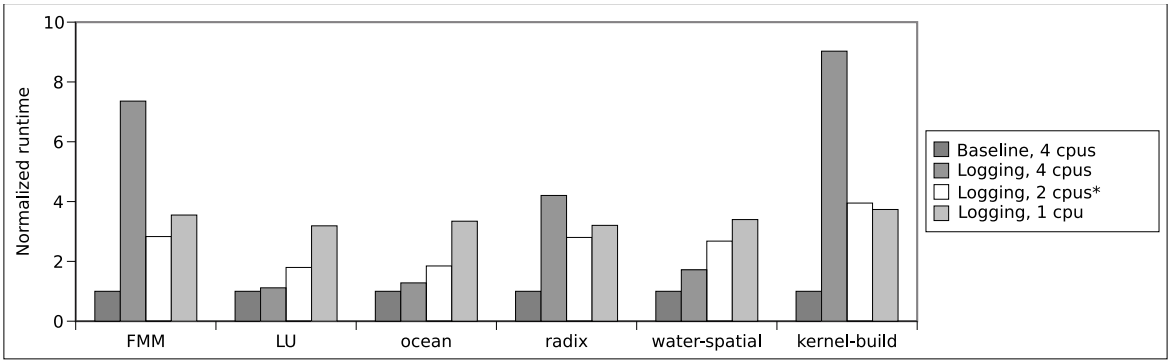


Figure 5. Overhead of ReVirt for a four processor Xen guest. The two processor logging was run on different hardware, so has been normalized to the single processor logging case on the same hardware.

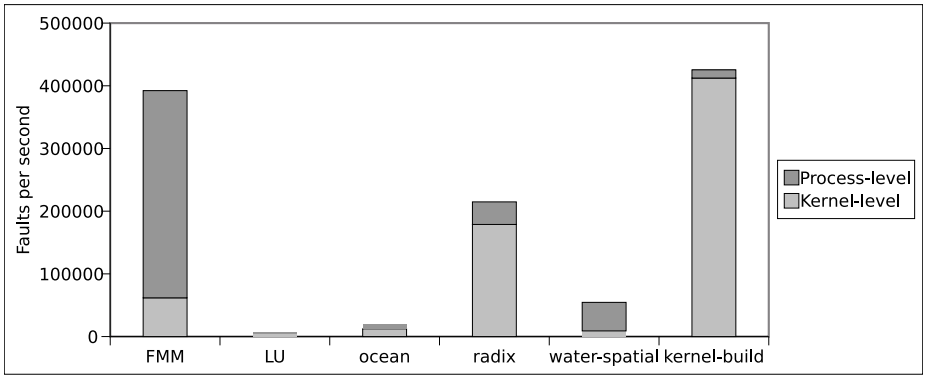


Figure 6. Kernel- and process-level sharing rates, in faults per second, for a four processor guest

or more backup systems, which replay the logs immediately. This guarantees that the backup system is in the same state as the logging system, ready to take over in the event of a failure. Since his system is targeted at single processor virtual machines, he does not address the issue of sharing.

Execution replay has had a long history in the debugging and parallel computing fields. The idea of using a CREW protocol to simplify recording was first discussed in [9]. Subsequent research using this technique reduced redundant constraints and used other techniques to optimize logging[13, 15]. The literature in the parallel computing fields is targeted at replaying parallel applications, and the solutions require the software being replayed to be modified. (We consider binary instrumentation to be a form of software modification.) Furthermore, only a single application is logged and replayed. By using hardware page protections and logging a virtual machine, SMP-ReVirt can replay an entire system of unmodified software.

Bacon et. al[1] first proposed supporting multiprocessor replay in hardware by snooping the cache-coherence protocol. Their simulated system uses a hardware instruction counter piggy-backed to cache-coherence messages to identify sharing.

Flight Data Recorder (FDR)[17] explores the idea in much more detail, using modern commercial workloads. Rather than replaying the entire execution, FDR focuses on replaying the last one second of execution before a crash. They instrument the cache-coherence hardware to detect memory races and generate constraints, using a modified version of Netzer's algorithm[13] to reduce redundant constraints. Input and interrupts are also logged. Checkpoints are implemented by logging old values of memory as it is modified. On a crash, the entire memory is dumped, which in conjunction with the old memory log, allow the state of one second ago to be re-constructed. The rest of the log allows FDR to replay the entire state of the system as it executed for the last second before the crash. Afterwords, the FDR team explored various optimizations to the system, including techniques to reduce the number of constraints logged and efficiently compress them[18]. These optimizations reduced the constraint log size (around 2 MB/s in [17]) by a factor of 25 on average.

The memory checkpoint data made up the majority of FDR's log. However, to analyze a crash, the entire state of the system may not be necessary. Rather than attempting to checkpoint the whole state, BugNet[12] logs the first read from shared memory in each replay interval, or when a data race is detected. Only the state of the register file (including the instruction pointer) is regenerated by execution replay. The memory state that affects execution can be reconstructed from the log. During replay, each processor is capable of replaying independently of the other processors. Constraints are recorded only as an aid to debugging, to help correlate the interleaving of processor execution, and are not necessary for correct execution.

The authors of BugNet went on to develop a different kind of constraint. Rather than logging constraints between individual instructions (which they term a *point-to-point* method), they use barrier-like logs of memory operations called *strata*. This technique lends itself naturally to instrumenting the cache-coherence protocol, but more investigation is necessary to see whether it can be implemented with hardware page protections.

Using the page protections causes two distinctions between SMP-ReVirt and both architecture-based and software-based systems. The first is that the granularity of sharing is limited to the

size of the page, which on x86 processors is 4096 bytes. This granularity is much larger than object and cacheline granularity. As a result, SMP-ReVirt is likely to be more prone to false sharing, which causes unnecessary run-time overhead and larger log file sizes.

Secondly, both architecture-based and software-based systems are able to store information about the last memory access, while SMP-ReVirt is only being able to store information about the point of privilege-reduction. As a result, hardware- and software-based systems can be less over-constrained than SMP-ReVirt, and can perform certain kinds of optimizations like Netzer's transitive reduction[13], while SMP-ReVirt cannot. Over-constraining may have an impact on the efficiency of replay, but is unlikely to have an effect on the efficiency of logging or the number of constraints. Missing opportunities for optimizations will make SMP-ReVirt's log larger, but it's not clear exactly how much; [18] only reports the results after all optimizations, not one-by-one, and some of the optimizations may be applicable to SMP-ReVirt.

There are also several distinctions worth noting between SMP-ReVirt and existing hardware cache-based approaches. First, SMP-ReVirt runs on commodity hardware, while cache-based approaches currently exist only in simulation. Secondly, SMP-ReVirt's goal is to log an entire execution of a virtual machine onto disk, while FDR and BugNet log just the last seconds before a crash into memory.

In part, FDR's one-second limitation is due to the fact that it logs only to memory. This is in part a result of the fact that FDR is logging *all* software running on the system. Everything must be done in hardware, and certain things are more difficult to do in hardware, like marshaling the logs onto disk or over the network, or involving the disk in replay to avoid logging all data read from disk. SMP-ReVirt allows some software to be involved in logging and replay, such as the hypervisor and domain 0. This gives SMP-ReVirt more flexibility for what to do with the log, including putting it on long-term storage on disk or sending it over a network.

We believe it would be ideal to take a mixed approach: add hardware support for detecting memory sharing and generating constraints, which the hypervisor could use instead of implementing the CREW protocol for normal pages. Hardware support would allow us to detect sharing on a cache-line granularity, which has the potential to greatly reduce the overhead of sharing, while maintaining both the ability to log unmodified software. Logging guests from the hypervisor would give us the flexibility of sending the logs to disk or network, giving us enough storage to log the entire execution of a virtual machine, rather than only the last few seconds.

6. Conclusion

We have presented our work on SMP-ReVirt, the first system to log and replay multiprocessor virtual machines on commodity hardware. We use hardware page protections to detect races between virtual cpus in a multiprocessor virtual machine. This allows us to log and replay an entire virtual machine, including the kernel and all applications, without modifying the software.

Using the hardware page protections avoids the overhead of instrumenting every read and write in software, but necessitates a large granularity of sharing and incurs higher overhead when sharing occurs. Some workloads perform poorly, but others perform surprisingly well. The main factors that influence performance include how much the workload involves the guest kernel, and how sensitive the workload is to false sharing at larger sharing granularities.

References

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

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

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

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

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

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

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

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

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

[10] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, 1993. Chapter 6.

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

[12] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Internationals Symposium on Computer Architecture (ISCA)*, June 2005.

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

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

[15] M. W. Shapiro. RDB: A System for Incremental Replay Debugging. Technical Report CS-97-12, Brown University, July 1997.

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

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

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

[19] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS, San Diego, CA, June*, volume 3, pages 4–2, 2007.