Debugging operating systems with time-traveling virtual machines

Samuel T. King, George W. Dunlap, and Peter M. Chen

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
http://www.eecs.umich.edu/CoVirt

Abstract

Operating systems are among the most difficult of software systems to debug with traditional cyclic debugging. They are non-deterministic; they run for long periods of time; their state and code is large and complex; and their state is easily perturbed by the act of debugging. This paper describes a time-traveling virtual machine that overcomes many of the difficulties associated with debugging operating systems. By time travel, we mean the ability to navigate backward and forward arbitrarily through the execution history of a particular run and to replay arbitrary segments of the past execution. We integrate time travel into a general-purpose debugger to enable a programmer to debug an OS in reverse, implementing commands such as reverse breakpoint, reverse watchpoint, and reverse single step. The space and time overheads needed to support time travel are reasonable for debugging, and movements in time are fast enough to support interactive debugging. We demonstrate the value of our time-traveling virtual machine by using it to understand and fix several OS bugs that are difficult to find with standard debugging tools.

1 Introduction

Computer programmers are all-too-familiar with the task of debugging complex software through a combination of detective work, guesswork, and systematic search. Tracking down a bug generally starts with running a program until an error in the program manifests as a fault. The programmer[1] then seeks to start from the fault (the manifestation of the error) and work backward to the cause of the fault (the programming error itself). Cyclic debugging is the classic way to work backward toward the error. In cyclic debugging, a programmer uses a debugger or output statements to examine the state of the program at a given point in its execution. Armed with this information, the programmer then re-runs the program, stops it at an earlier point in its execution history, examines the state at this point, then iterates.

Unfortunately, this classic approach to debugging is difficult to apply when debugging operating systems. Many aspects of operating systems make them among the most difficult of software systems to debug. Programmers cannot stand completely outside the OS to examine and control it without perturbing it; operating systems are non-deterministic; operating systems run for long periods of time; and operating systems are large and complex.

We now describe in more detail why operating systems are difficult to debug. First, programmers cannot stand completely outside the operating system when examining its state and controlling its execution. Because the operating system is the lowest level of software on a computer, programmers can only examine or control the OS by using some functionality in the OS (or by using specialized hardware such as an in-circuit emulator). Even if the programmer runs a remote kernel debugger on a second physical machine, the OS being debugged must support some basic functionality, such as reading and writing memory locations, setting and handling breakpoints, and communicating with the remote debugger (e.g., through the serial line). Using this basic functionality may be impossible on a sick OS. It may also perturb the state of the system being debugged. Consider how one would examine the file system on disk: one would need to navigate through the in-memory and on-disk file system structures and use the OS's device driver to read disk blocks into memory. Each of these steps would require some working functionality in the OS and would probably perturb the state of the file cache, the file system, and numerous kernel data structures.

Second, operating systems are non-deterministic.

[1] In this paper, the term "programmer" refers to the person debugging the system. The term "debugger" refers to the programming tool (e.g., `gdb`) used by the programmer to examine and control the program.

Their execution is affected by non-deterministic effects such as the interleaving of multiple threads, interrupts, user input, network input, and the perturbations of state caused by the programmer who is debugging the system. This non-determinism makes cyclic debugging infeasible, because the programmer cannot re-run the system to examine the state at an earlier point.

Third, operating systems run for long periods of time, such as weeks, months, or even years. Re-running the system in cyclic debugging is thus infeasible even if the OS were completely deterministic.

Finally, operating systems are large and complex. This is especially true if one takes into account that running an OS really means running all software on the machine, since application software affects the execution of the OS. The state of the OS is enormous because it includes the file system on disk, the virtual memory state of all processes, and all physical memory. This large state makes it unwieldy to roll back and re-run the system. One can choose to re-run the system without rolling back its state, but this introduces another source of non-determinism between runs. Another reason operating systems are complex to debug is they use self-modifying code when loading a new kernel module or a new application executable.

In this paper, we describe how to use *time-traveling virtual machines* to overcome many of the difficulties associated with debugging operating systems. By virtual machine, we mean a software-implemented abstraction of a physical machine that is at a low-enough level to run an operating system. Running the OS inside a virtual machine enables the programmer to stand outside the OS being debugged. From this vantage point, the programmer can use a general-purpose debugger to examine and control the execution of the OS without perturbing its state.

By time travel, we mean the ability to navigate backward and forward arbitrarily through the execution history of a particular run and to replay arbitrary segments of the past execution. For example, if the system crashed due to an errant pointer variable, time travel would allow the programmer to go back to the point when that pointer variable was corrupted; it would also allow the programmer to fast-forward again to the crash point. Time-traveling virtual machines allow a programmer to replay a prior point in the execution exactly as it was executed the first time. The past is immutable in our model of time travel; this ensures that there is only a single execution history, rather than a branching set of execution histories. As with cyclic debugging, the goal of time travel is to enable the programmer to examine the state of the OS at prior points in the execution. However, unlike cyclic debugging, time travel works in the presence of non-determinism. Time travel is also more convenient than classic cyclic debugging because it does not require the entire run to be repeated.

In this paper, we describe the design and implementation of a time-traveling virtual machine (TTVM) for debugging operating systems. We integrate time travel into a general-purpose debugger (`gdb`) for our virtual machine, implementing commands such as reverse step (go back to the last instruction that was executed), reverse breakpoint (go back to the last time an instruction was executed), and reverse watchpoint (go back to the last time a variable was modified).

The space and time overhead needed to support time travel is reasonable for debugging. For two workloads that exercise the OS intensively, the logging needed to support time travel adds 10-12% time overhead and 13-190 KB/sec space overhead. The speed at which one can move backward and forward in the execution history depends on the frequency of checkpoints in the time region of interest. Our system is able to insert additional checkpoints to speed up these movements or delete existing checkpoints to reduce space overhead. After adding checkpoints to a region of interest, our system allows a programmer to move to an arbitrary point within the region in about 12 seconds.

The following real-life example clarifies what we mean by debugging with time-traveling virtual machines and illustrates the value of debugging in this manner. The error we were attempting to debug was triggered when the guest kernel attempted to call a NULL function pointer. The error had corrupted the stack, so standard debugging tools were unable to traverse the call stack and determine where the invalid function call had originated. Using the TTVM reverse single step command, we were able easily to step back to where the function invocation was attempted and examine the state of the virtual machine at that point. We found the ability to debug in reverse tremendously helpful in finding this bug, as well as the other bugs described in Section 6.

2 Virtual machines

A virtual machine is a software abstraction of a physical machine [11]. The software layer that provides this abstraction is called a virtual machine monitor (VMM). An operating system can be installed and run on a virtual machine as if it were running on a physical machine. Such an OS is called a ”guest” OS to distinguish it from an OS that may be integrated into the VMM itself (which is called the ”host” OS).

Several features of virtual machines make them attractive for our purposes. First, because the VMM adds a layer of software below the guest OS, it provides a convenient substrate on which one can add new features. Unlike remote kernel debugging, these new features need no

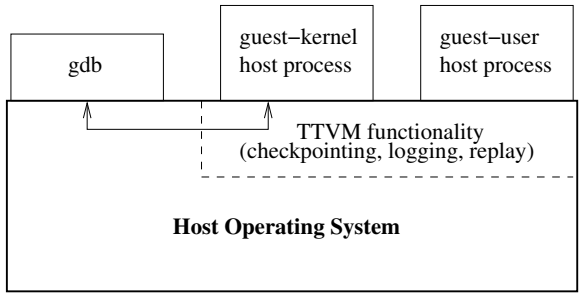

Figure 1: System structure: UML runs as two user processes on the host Linux OS, the *guest-kernel host process* and the *guest-user host process*. TTVM's ability to travel forward and back in time is implemented by modifying the host OS. We extend `gdb` to make use of this time traveling functionality. `gdb` communicates with the guest-kernel host process via a remote serial protocol.

support or functionality in the guest OS. We use this substrate to add traditional debugging capabilities such as setting breakpoints and reading and writing memory locations. We also add non-traditional debugging features such as logging and replaying non-deterministic inputs and saving and restoring the state of the virtual machine.

Second, a VMM allows us to run a general-purpose, full-featured debugger on the same physical machine as the OS being debugged without perturbing the debugged OS. Compared to traditional kernel debuggers, virtual machines enable more powerful debugging capabilities (e.g., one can read the virtual disk) with no perturbation of or dependency on the OS being debugged. It is also more convenient to use than a remote debugger because it does not require a second physical machine.

Finally, a VMM offers a narrow and well-defined interface: the interface of a physical machine. This interface makes it easier to implement the checkpointing and replay features we add in this paper, especially compared to the relatively wide and complex interface offered by an operating system to its application processes. The state of a virtual machine is easily identified as the virtual machine's memory, disk, and registers and can thus be saved and restored easily. Replay is easier to implement in a VMM than an operating system because the VMM exports an abstraction of a uniprocessor virtual machine (assuming a uniprocessor physical machine), whereas an OS exports an abstraction of a virtual multiprocessor to its application processes.

The VMM used in this paper is User-Mode Linux (UML) [8]. UML is implemented as a kernel modification to a host Linux OS (Figure 1) [2]. The virtual machine runs as two user processes on the host OS: one host process (the *guest-kernel host process*) runs all guest kernel code, and one host process (the *guest-user host process*) runs all guest user code. The guest-kernel host process uses the Linux `ptrace` facility to intercept system calls and signals generated by the guest-user host process. The guest-user host process uses UML's skas-extension to the host Linux kernel to switch quickly between address spaces of different guest user processes.

UML's VMM exports a virtual architecture that is similar but not identical to the host hardware. The guest OS in UML (also Linux) must be ported to run on top of this virtual architecture. Each piece of virtual hardware in UML is emulated with a host service. The guest disk is emulated by a raw disk partition on the host; the guest memory is emulated by a memory-mapped file on the host; the guest network card is emulated by a host TUN/TAP virtual Ethernet driver; the guest MMU is emulated by calls to the host `mmap` and `mprotect` system calls; guest timer and device interrupts are emulated by host SIGALRM and SIGIO signals; the guest console is emulated by standard output. The guest Linux's architecture-dependent layer uses these host services to interact with the virtual hardware. In addition to UML's standard virtual devices, we provide the ability to use unmodified real device drivers in the guest OS to drive devices on the host platform (Section 3.2).

Running an OS inside a virtual machine incurs overhead. We measured UML's virtualization overhead as 76% for a build of the Linux kernel (a system-call intensive workload which is expensive to virtualize [13]). This overhead is acceptable for debugging (in fact, UML is used in production web hosting environments). If lower overhead is needed, the ideas in this paper can be applied to faster virtual machines such as Xen [4] (3% overhead for a Linux kernel build), UML-Linux/FAUmachine [13] (35% overhead for a Linux kernel build), or a hardware-supported virtual machine such as Intel's upcoming Vanderpool Technology.

3 Time-traveling virtual machines

A time-traveling virtual machine should have two capabilities. First, it should be able to reconstruct the complete state of the virtual machine at any point in a run, where a run is defined as the time from when the virtual machine was powered on to the last instruction it executed. Second, it should be able to start from any point in a run and from that point replay the same instruction stream that was executed during the original run from that point. This section describes how TTVM achieves these capabilities through a combination of logging, replay, and checkpointing.

[2] We use the skas (separate kernel address space) version of UML.

3.1 Logging and replaying a virtual machine

The foundational capability in TTVM is the ability to replay a run from a given point in a way that matches the original run instruction for instruction. Replay causes the virtual machine to transition through the same states as it went through during the original run; hence replay enables one to reconstruct the complete state of the virtual machine at any point in the run. TTVM uses the ReVirt logging/replay system to provide this capability [9]. This section briefly summarizes how ReVirt logs and replays the execution of a virtual machine.

A virtual machine can be replayed by starting from a checkpoint, then replaying all sources of nondeterminism [6, 9]. For UML, the sources of nondeterminism are external input from the network, keyboard, and real-time clock and the timing of virtual interrupts. The VMM replays network and keyboard input by logging the calls that read these devices during the original run and regenerating the same data during the replay run. Likewise, we configure the CPU to cause reads of the real-time clock to trap to the VMM, where they can be logged or regenerated.

To replay a virtual interrupt, ReVirt logs the instruction in the run at which it was delivered and re-delivers the interrupt at this instruction during replay. This point is identified uniquely by the number of branches since the start of the run and the address of the interrupted instruction [16]. ReVirt uses a performance counter on the Intel Pentium 4 CPU to count the number of branches during logging, and it uses the same performance counter and instruction breakpoints to stop at the interrupted instruction during replay [1]. Replaying interrupts enables ReVirt to replay the scheduling order of multi-threaded guest operating systems and applications, as long as the VMM exports the abstraction of a uniprocessor virtual machine [18]. Replaying a virtual shared-memory multiprocessor will likely incur significantly higher overhead because events from multiple threads can be interleaved on very fine granularity, and we defer this for future work.

3.2 Supporting real device drivers in the guest OS

In general, VMMs export a limited set of virtual devices. Some VMMs export virtual devices that exist in hardware (e.g., VMware Workstation exports an emulated AMD Lance Ethernet card); others (like UML) export virtual devices that have no hardware equivalent. Exporting a limited set of virtual devices to the guest OS is usually considered a benefit of virtual-machine systems, because it frees guest OSs from needing device drivers for myriad host devices [21]. However, when using virtual machines to debug operating systems, the limited set of virtual devices prevents programmers from using and debugging drivers for real devices; programmers can only debug the architecture-independent portion of the guest OS. There are two ways to address this limitation and enable the programmer to run and debug real device drivers in a guest OS. With both strategies, real device drivers can be included in the guest OS without being modified or re-compiled.

The first way to run a real device driver in the guest OS is for the VMM to provide a software emulator for that device. The device driver issues the normal set of I/O instructions: `IN/OUT` instructions, memory-mapped I/O, DMA commands, and interrupts. The VMM traps these privileged instructions and forwards them to/from the software device emulator. With this strategy, ReVirt can log and replay device driver code in the same way it logs and replays the rest of the guest OS. If one runs the VMM's software device emulator above ReVirt's logging system (and above the checkpoint system described in Section 3.3), ReVirt will guide the emulator and device driver code through the same instruction sequence during replay as they executed during logging. While this first strategy fits in well with the existing ReVirt system, it only works if one has an accurate software emulator for the device whose driver one wishes to debug.

We modified UML to provide a second way to run real device drivers in the guest OS, which works even when no software emulator exists for the device of interest. With this strategy, the VMM traps and forwards the privileged I/O instructions issued by the guest OS device driver to the actual hardware device. The programmer specifies the I/O port space and the range of memory-mapped I/O space for the device; the VMM uses this knowledge to allow or disallow accesses to privileged ports/memory. The programmer also specifies the DMA command set for the device, so the VMM knows when the guest OS initiates a DMA transfer and when the device finishes a DMA transfer.[3]

This second strategy requires extensions to enable ReVirt to log and replay the execution of the device driver. Whereas the first strategy placed the device emulator above the ReVirt logging layer, the second strategy forwards driver actions to the actual hardware device. Because this device may not be deterministic, ReVirt must log any information sent from the device to the driver. Specifically, ReVirt must log and replay the data returned by `IN` instructions, memory-mapped I/O instructions, and completed DMA reads. To avoid confusing the device, ReVirt suppresses output to the device during replay.

[3] A programmer trying to write/debug a device driver already needs this information.

TTVM is able to trap and forward all types of I/O instructions (`IN/OUT` instructions, memory-mapped I/O, DMA commands, interrupts), and ReVirt logs and replays all device interactions that return data to the driver. Our prototype supports logging and replay for device drivers for a subset of our test platform's devices (16550A serial port and Intel ICH4 soundcard); one can add a new device by informing the VMM of the device's I/O port space, range of memory-mapped I/O space, and DMA commands. Trapping, forwarding, and logging I/O instructions slows I/O bound workloads by a moderate amount. For the serial port, each I/O instruction is slowed by 50% (0.6 μs). For the soundcard, the guest OS device driver can play an MP3 music clip and record audio in real-time.

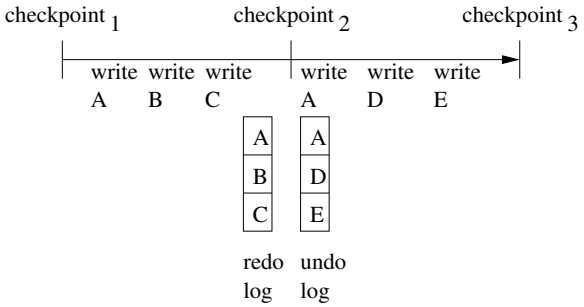

Figure 2: Checkpoints of the disk are represented as undo and redo logs. The figure shows the redo and undo logs that would results for checkpoint$_2$ for the given sequence of disk writes.

3.3 Checkpointing for faster time travel

Logging and replaying a virtual machine from a single checkpoint at the beginning of the run is sufficient to recreate the state at any point in the run from any other point in the run. However, one cannot recreate this state *quickly* by using logging and replay alone because the virtual machine must re-execute each instruction from the beginning to the desired point, and this period may span many days. To accelerate time travel over long periods, TTVM takes periodic checkpoints while the virtual machine is running [19] (ReVirt started only from a disk checkpoint of a powered-off virtual machine).

The simplest way to checkpoint the virtual machine is to save a complete copy of the state of the virtual machine. This state is comprised of the CPU registers, the virtual machine's physical memory, the virtual disk, and any state in the VMM or host kernel that affects the execution of the virtual machine. For UML, this host kernel state includes the address space mappings for the guest-user host process and the guest-kernel host process, the state of open host file descriptors, and the registration of various signal handlers (analogous to the interrupt descriptor table on real hardware).

The state of the virtual machine may also be contained in other host kernel state, such as the data in the host kernel's stack while the virtual machine is blocked in a host system call. We simplify the checkpointing and restoration of host kernel state by taking a checkpoint only when the guest-user host process initiates a system call to the guest OS. At this point, the host kernel stacks for the guest-user and guest-kernel host processes are small and well-defined. We simplify the act of restoring the system to a checkpoint by restoring the virtual machine's state only from a similar state, i.e. when the guest-user host process initiates a system call. At this point, the host kernel stacks for the guest-user and guest-kernel host processes already match the values they had at the time of the checkpoint, so TTVM does not need to explicitly save or restore this portion of the host kernel state.

Saving a complete copy of the virtual-machine state is simple but inefficient. In particular, saving the entire state of the virtual disk adds enormous space and time overhead. Several methods have been used in the past to reduce this overhead. One way is to structure the virtual disk as a log [20]; this method can keep all versions of disk blocks that are needed to restore to specific checkpoints without extra disk I/O. To keep our prototype simple, we currently maintain the in-place structure of the virtual disk and supplement it with undo and redo logs. The undo and redo logs use copy-on-write to store only those disk blocks that have been modified since the last checkpoint. Starting with the disk contents at a current point, the complete state of the disk can be restored back to a prior checkpoint by starting from the current disk contents and restoring the disk blocks in the undo log. The undo log at checkpoint$_n$ contains the set of disk blocks that have been modified between checkpoint$_n$ and checkpoint$_{n+1}$, with the values of the blocks in the undo log being those at checkpoint$_n$ (Figure 2).

Analogously, TTVM enables a programmer to move forward in time to a future checkpoint by using the disk blocks in the redo log. The redo log at checkpoint$_n$ contains the set of disk blocks that have been modified between checkpoint$_{n-1}$ and checkpoint$_n$, with the values of these disk blocks again being those at checkpoint$_n$ (Figure 2).

If a disk block is modified during two successive checkpoint intervals, the undo and redo logs for the checkpoint between these two intervals will contain the same values for that disk block. TTVM detects this case and shares such data between the undo and redo logs. E.g., in Figure 2, block A's data is shared between checkpoint$_2$'s undo and redo logs.

Similar copy-on-write techniques could be applied to

the virtual machine's physical memory, but this is more difficult. Intercepting memory store instructions adds more overhead and complexity than intercepting writes to disk because writes to disk are performed in larger units and are done through explicit calls to VMM service routines. In addition, a significant fraction of the virtual machine's physical memory may be modified over a checkpoint interval of minutes or hours, and this fraction limits the degree to which copy-on-write can reduce the number of memory pages that must be saved in a checkpoint.

3.4 Time traveling between points of a run

TTVM enables a programmer to travel arbitrarily backward and forward in time through a run. Time traveling between points in a run requires a combination of restoring to a checkpoint and replay. To travel from point A to point B, TTVM first restores to the checkpoint that is prior to point B (call this checkpoint$_n$). TTVM then replays the execution of the virtual machine from checkpoint$_n$ to point B. The more frequently checkpoints are taken, the smaller the expected duration of the replay phase of time travel.

Restoring to checkpoint$_n$ requires several steps. TTVM first restores the copy saved at checkpoint$_n$ of the virtual machine's registers, physical memory, and any state in the VMM or host kernel that affects the execution of the virtual machine. Restoring the disk to the values it had at checkpoint$_n$ makes use of the data stored in the undo logs (if moving backward in time) or redo logs (if moving forward in time). Consider an example of moving from a point after checkpoint$_{n+2}$ backward to checkpoint$_n$ (restoring to a checkpoint in the future uses the redo log in an analogous manner). TTVM first restores the disk blocks from the undo log at checkpoint$_n$. It then examines the undo log at checkpoint$_{n+1}$ and restores any disk blocks that were not restored by the undo log at checkpoint$_n$. Finally, TTVM examines the undo log at checkpoint$_{n+2}$ and restores any disk blocks that were not restored by the undo logs at checkpoint$_n$ or checkpoint$_{n+1}$. Applying the logs in this order ensures that each disk block is written at most once.

3.5 Adding and deleting checkpoints

An initial set of checkpoints are taken during the original, logged run. TTVM supports the ability to add or delete checkpoints from this original set. At any time, the user may choose to delete existing checkpoints to free up space. While replaying a portion of a run, a programmer may choose to supplement the initial set of checkpoints to speed up anticipated time-travel operations. The non-disk portions of a checkpoint (e.g., memory and registers) are stored as a complete copy, so adding or deleting those portions of a checkpoint is straightforward—adding a new checkpoint does not affect this portion of existing checkpoints, and deleting a checkpoint can simply throw away this portion of the deleted checkpoint. Adding or deleting the undo and redo log portions of a checkpoint is less straightforward, and we next describe how to manipulate these data structures in these cases.

Adding a new checkpoint can be done when the programmer is replaying a portion of a run from a checkpoint (say, checkpoint$_1$). TTVM goes through four steps to add a new checkpoint$_2$ at the current point of replay (between existing checkpoint$_1$ and checkpoint$_3$). First, TTVM removes any disk block from checkpoint$_1$'s undo log that was not written between checkpoint$_1$ and checkpoint$_2$. TTVM identifies these disk blocks by maintaining a list of the disk blocks that are modified since the system started replaying at checkpoint$_1$, just as it does during logging to support the copy-on-write undo log. This removal is optional, since extra blocks in the undo log are inefficient but not incorrect. Second, TTVM creates the redo log for checkpoint$_2$, which consists of the same set of disk blocks in checkpoint$_1$'s new undo log, but with the values at the current point of replay. Third, TTVM removes from checkpoint$_3$'s redo log any disk block that was not written between checkpoint$_2$ and checkpoint$_3$. To identify these disk blocks, TTVM stores a timestamp for each block in a checkpoint's redo log, which is the latest time that block was written before the checkpoint. TTVM compares the current timestamp at checkpoint$_2$ with the timestamp at checkpoint$_3$ to identify which blocks to remove from checkpoint$_3$'s redo log. Fourth, TTVM creates the undo log for checkpoint$_2$, which consists of the same set of disk blocks in checkpoint$_3$'s new redo log, but with the values at the current point of replay.

Deleting an existing checkpoint (presumably to free up space for a new checkpoint) can be done during the original logging run or when the programmer is replaying a portion of a run. TTVM goes through two steps to delete checkpoint$_2$ (between checkpoint$_1$ and checkpoint$_3$). TTVM first moves the blocks in checkpoint$_2$'s undo log to checkpoint$_1$'s undo log. A block that already exists in checkpoint$_1$'s undo log takes precedence over a block from checkpoint$_2$'s undo log. Similarly, TTVM moves the blocks in checkpoint$_2$'s redo log to checkpoint$_3$'s redo log. A block that already exists in checkpoint$_3$'s redo log takes precedence over a block from checkpoint$_2$'s redo log.

3.6 Expected usage model

We expect programmers to use TTVM in three phases. In phase 1, the programmer runs a test to trigger an er-

ror. This phase may last a long time (hours or days), so TTVM should minimize the time and space overhead during this phase. Programmers should therefore specify a long checkpoint interval during this phase (e.g., 10 minutes). For long runs, the system may not be able to save all of the checkpoints taken, so TTVM may need to delete some prior checkpoints. TTVM's default policy keeps more frequent checkpoints for periods near the current time than for periods farther in the past; this policy assumes that periods in the near past are likely to be the ones of interest during debugging. TTVM chooses checkpoints to delete by fitting them to a distribution where the distance between checkpoints increases exponentially as one goes farther back in time [5].

In phase 2, the programmer attaches the debugger, switches the system from logging to replay, and prepares to debug the error. To speed up time-travel operations, programmers should start usually by adding more frequent checkpoints to the portion of the run they expect to debug. This can be accomplished by replaying the interesting portion of the run (say, a 10 minute interval) while taking frequent checkpoints (say, every 10 seconds). Section 5 will show that these frequent checkpoints add moderate overhead, but this overhead is added only to the portion of the run that is about to be debugged rather than to the entire run.

In phase 3, the programmer uses TTVM to time-travel forward and backward through the run and debug the error. We next describe new debugging commands that allow a programmer to navigate conveniently through the run.

4 TTVM-aware gdb

In this section, we discuss how to integrate the time traveling capability of TTVM into a debugger (`gdb`). We first introduce the new reverse debugging commands and discuss how they are implemented. We then describe how to manage the interaction of time traveling with the state changes generated by `gdb`. Finally, we describe how our prototype implements communication between `gdb` and TTVM.

4.1 Time travel within gdb

In addition to the standard set of commands available to debuggers, TTVM allows `gdb` to restore prior checkpoints, replay portions of the execution, and examine arbitrary past states. A promising application of these technique is providing the illusion of virtual-machine reverse execution.

Reverse execution, when applied to debugging, provides the functionality standard debuggers are often trying to approximate. For example, a kernel may follow

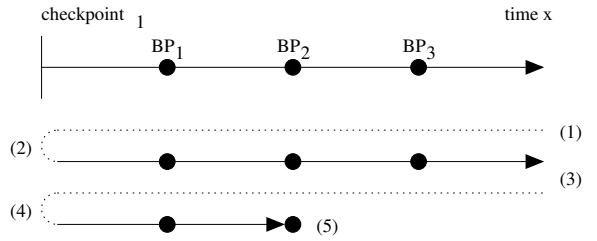

Figure 3: Reverse continue uses two execution passes. The programmer calls `reverse continue` at time x. In the first pass, (1) TTVM restores checkpoint$_1$, then (2) replays execution until time x. Along the way, TTVM makes note of breakpoints BP$_1$, BP$_2$, and BP$_3$. When time x is reached, programmer sees a list of these breakpoints and selects one to go back to. In the example show here, the programmer selects BP$_2$. In the second pass, TTVM again (3) restores checkpoint$_1$ and (4) replays execution, but this time TTVM stops at breakpoint BP$_2$ and returns control to the programmer (5).

an errant pointer, read an unintended data structure, and crash. Using a standard debugger, the programmer can gain control when the crash occurs. A common approach at this point is to traverse up the call stack. This approximates reverse execution because it allows the programmer to see the partial state of function invocations that occurred before the crash. However, it only allows the programmer to see variables stored on the stack, and it only shows the values for those variables at the time of each function invocation. Another approach is to re-run the system with a watchpoint set on the pointer variable. However, this approach works only if the bug is deterministic. Also, the programmer may have to step through many watchpoints to get to the modification of interest. Ideally, the programmer would like to go to the *last* time the pointer was modified. However, current debugging commands only allow the programmer to go to the *next* modification of the pointer.

To overcome this deficiency, we add a new command to `gdb` called `reverse continue`. `reverse continue` takes the virtual machine back to a *previous* point, where the point is identified by the reverse equivalents of forward breakpoints, watchpoints, and steps. In the example above, the programmer could set a watchpoint on the pointer variable and issue the `reverse continue` command. After executing this command, the debugger would return control to the programmer at the *last* time the variable was modified. This jump backward in time restores all virtual-machine state, so the programmer could then use standard `gdb` commands to gather further information.

The `reverse continue` command is imple-

mented using two execution passes (Figure 3). In the first pass, TTVM restores a checkpoint that is earlier in the execution and replays the virtual machine until the current location is reached again. During the replay of the first pass, `gdb` receives control on each trap caused by `gdb` commands issued by the programmer (e.g., breakpoints, watchpoints, steps). `gdb` keeps a list of these traps and, when the first pass is over, allows the programmer to choose a trap to time travel back to. During the second pass, `gdb` again restores the same checkpoint and replays. When the selected trap is encountered during the second pass, `gdb` returns control to the programmer.

This approach is general enough that it provides reverse versions to *all* `gdb` commands. For example, the programmer can set instruction breakpoints, conditional breakpoints, data watchpoints, or single steps (or combinations thereof), and the `reverse continue` command keeps track of all resulting traps and allows the programmer to go back to any of them. We have found each of these reverse commands useful in our kernel debugging (Section 6).

We found `reverse step` to be a particularly useful command (`reverse step` goes back a specified number of instructions). This command is particularly useful because it tracks instructions executed in guest kernel mode regardless of the kernel entry point. For example, if `gdb` has control inside a guest interrupt handler, and the interrupt occurred while the guest kernel was running, `reverse step` can go backward to determine which guest kernel instruction was preempted. We implemented an optimized version of the `reverse step` command because it is used so frequently and because the unoptimized version generates an inordinate number of traps. On x86, `gdb` uses the CPU's `trap` flag to single step forward. `reverse step` also uses the `trap` flag, but doing so naively would generate a trap to `gdb` on each instruction replayed from the checkpoint. To reduce the number of traps caused by `reverse step`, we wait to set the `trap` flag during each pass's replay until the system is near the current point. Our current implementation defines "near" to be within one system call of the current point, but one could easily define "near" to be within a certain number of branches.

Finally, we implemented a `goto` command that a programmer can use to jump to an arbitrary time in the execution, either behind or ahead of the current point. Our current prototype defines time in a coarse-grained manner by counting guest system calls, but it is possible to define time by logging the real-time clock, or by counting branches. `goto` is most useful when the programmer is trying to find a time (possibly far from the current point) when an error condition is present.

4.2 TTVM/debugger interactions

Time traveling must affect debugging state (e.g., breakpoints that are set) differently from how it affects other virtual-machine state. Time-travel operations change the virtual-machine state, but they should preserve debugging state. For example, if the programmer sets a breakpoint and executes `reverse continue`, the breakpoint must be unperturbed by the checkpoint restoration so that it can trap to `gdb` during the first and second pass of replay. Unfortunately, `gdb` mingles debugging state and virtual-machine state. For example, `gdb` implements software breakpoints by inserting x86 `breakpoint` instructions directly into the code page of the process being debugged.

To enable special treatment of debugging state, TTVM tracks all modifications `gdb` makes to the virtual state. This allows TTVM to make debugging state persistent across checkpoint restores by manually restoring the debugging state after the checkpoint is restored. In addition, TTVM removes any modifications caused by the debugger before taking a checkpoint, so that the checkpoint includes only the original virtual-machine state.

4.3 Reverse gdb implementation

`gdb` and TTVM communicate via the `gdb` remote serial protocol (Figure 1). The remote serial protocol between `gdb` and TTVM is implemented in a host kernel device driver. `gdb` already understands the remote serial protocol and so does not need to be modified. The host kernel device driver receives the low-level remote protocol commands and reads/writes the state of the virtual machine on behalf of the debugger. These reads and writes are transparent to the virtual machine: neither the execution or replay of the virtual machine is affected (unless the guest kernel reads state that has been modified by `gdb`).

Although `gdb` did not have to be modified to understand the remote serial protocol, it did have to be extended to implement the new reverse commands. This provided complete integration of the new reverse commands inside the familiar `gdb` environment.

5 Performance

In this section, we measure the time and space overhead of TTVM and the time to execute time-travel operations. Since debugging is dominated by human think time, our main goal in this section is only to verify that the overhead of TTVM is reasonable.

All measurements are carried out on a uniprocessor 3 GHz Pentium 4 with 1 GB of memory and two disks, a 40 GB Samsung SP4004H and a 120 GB Hitachi Deskstar

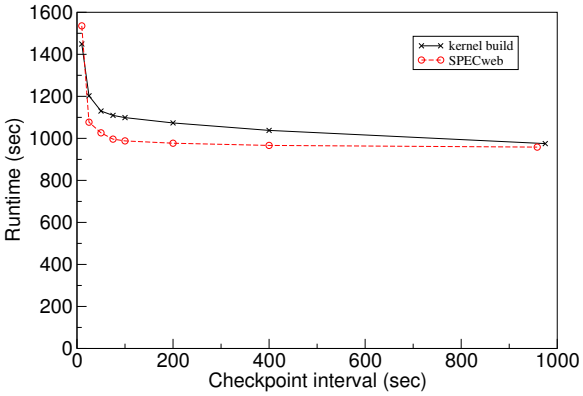

Figure 4: The effect of checkpointing on running time. For long runs, we expect programmers to use long checkpoint intervals (e.g., 10 minutes) to minimize overhead. For short, intensive debugging sessions, we expect programmers to accept higher overhead to enable faster time-travel operations during debugging.

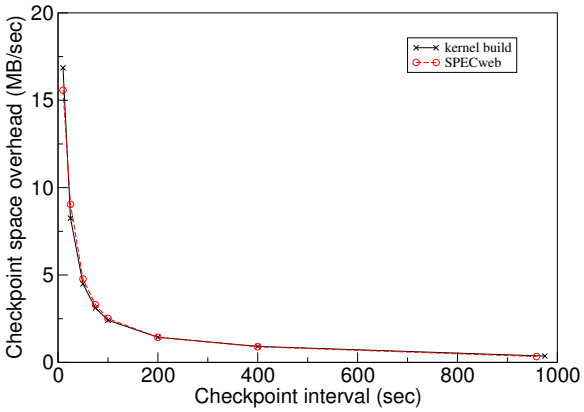

Figure 5: Space overhead of checkpoints. For long runs, programmers will use long checkpoint intervals to minimize overhead and will cap the maximum space used by checkpoints by deleting selected checkpoints.

GXP. The host file system is on the Hitachi disk, and the guest file system is on the Samsung disk. The host OS is Linux 2.4.18 with the skas extensions for UML and TTVM modifications. The guest OS is the UML port of Linux 2.4.20. We configure the guest to have 256 MB of memory and a 5 GB disk, which is stored on a host raw disk partition. Both host and guest file systems are initialized from a RedHat 9 distribution. All results represent the average of at least 5 trials.

We measure two workloads running on the guest OS: SPECweb99 using the Apache web server and three successive builds of the Linux 2.4 kernel (each of the three builds executes `make clean; make dep; make bzImage`),

We first measure the time and space overhead of the logging needed to support replay. Running these workloads on TTVM with logging (with a single checkpoint at the beginning of the run) adds 10% time overhead for kernel build and 12% overhead for SPECweb99, relative to running the same workload in UML on standard Linux (with skas). The space overhead of TTVM needed to support logging is 13 KB/sec for kernel build and 190 KB/sec for SPECweb99. These time and space overheads are very acceptable, especially for debugging.

Replay on TTVM occurs at approximately the same speed as the logged run. For the kernel-build workload, TTVM takes 2% longer to replay than it did to log; for SPECweb99, TTVM takes 1% longer to replay than it did to log. Replay is sometimes much faster than logging because TTVM skips over idle periods during replay.

We next measure the cost of checkpointing. Figures 4 and 5 show how the time and space overheads of checkpointing vary with the interval between checkpoints. We expect checkpoints to be taken at different frequencies for phase 1 and phase 2 of the usage model described in Section 3.6. In phase 1, we expect the programmer to minimize overhead over long runs by using a relatively long checkpoint interval. Taking checkpoints every 10 minutes adds less than 4% time overhead and less than 1 MB/sec of space overhead. Space overhead for long runs can also be capped at a maximum size, which causes TTVM to delete selected checkpoints.

In phase 2, we expect programmers to optimize for the speed of time-travel operations by taking frequent checkpoints (e.g., every 10 seconds) over a short period of interest (e.g., 10 minutes). For this short period, we expect programmers to tolerate much higher overhead than they would for the entire run. Time and space overheads remain quite moderate for checkpoint intervals as short as 25 seconds (less than 25% time overhead and less than 10 MB/sec space overhead), and they are tolerable even for intervals as short as 10 seconds (less than 60% time overhead and less than 17 MB/sec space overhead). For short checkpoint intervals, time and space overhead are dominated by the cost of saving the contents of the guest's physical memory.

Next we consider the speed of moving forward and backward through the execution of a run. As described in Section 3.4, time travel takes two steps: (1) restoring to the checkpoint prior to the target point and (2) replaying the execution from this checkpoint to the target point. Figure 6 shows the time to restore a checkpoint as a function of the distance from the current point to a prior or future checkpoint. We used a checkpoint interval of 45 seconds and spanned the run with 20 checkpoints. Moving to a checkpoint farther away takes more

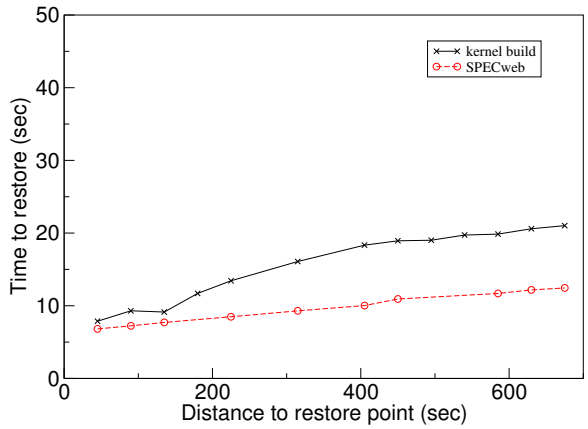

Figure 6: Time to restore to a checkpoint.

time because TTVM must restore more disk blocks. Recall that each unique disk block is written at most once, even when restoring to a point that is many checkpoints away. Hence the maximum time of a restore operation approaches the time to restore all unique disk blocks changed in the workload. The time for the second step depends on the distance from the checkpoint reached in step one to the target point. Since replay on TTVM occurs at approximately the same speed as the logged run, the average time of this step for a random point is half the checkpoint interval.

6 Experience and lessons learned

In this section, we describe our experience using TTVM to track down three kernel bugs and show how using reverse `gdb` commands simplified the process. Next, we talk about our experiences using TTVM, how the reverse commands helped, and why standard `gdb` fell short. Finally, we discuss the interactivity of our reverse debugging commands.

6.1 System call bug

While developing TTVM, we encountered a guest kernel panic. We first tried to debug this error using traditional cyclic debugging techniques and standard `gdb`, i.e. not using time travel. First, we set a breakpoint in the guest kernel `panic` function that is invoked when the kernel encounters an unrecoverable error. We then re-ran the virtual machine, hoping for the guest kernel panic to reoccur. Fortunately, the bug re-occurred and `gdb` gained control when a memory exception caused by guest kernel code triggered a panic. The fault occurred after the guest kernel attempted to execute an instruction at address 0. We tried to understand how the kernel reached address 0 by traversing up the call stack of the guest kernel. However, `gdb` was unable to traverse up the call stack because the most recent call frame had been corrupted when the kernel called the "function" at address 0. Since `gdb` was unable to find the prior function, we next looked at the data on the stack manually to try to find a valid return address. We found a few candidate addresses, but we eventually gave up after disassembling the guest kernel and searching through various assembly code segments.

We next used reverse commands to debug the guest kernel. We started by attaching `gdb` to the guest-kernel host process at the time of the panic. We then performed several reverse single steps which took us to the point at which address 0 had been executed. We performed another reverse single step and found that this address had been reached from the system call handler. At this point we used a number of standard `gdb` commands to inspect the state of the virtual machine and determine the cause of the error. The bug was an incorrect entry in the system call table, which caused a function call to address 0.

6.2 Kernel race condition bug

We next tried debugging a guest kernel bug that had been posted on the UML kernel mailing list. The error we found was triggered by executing the user-mode command `ltrace strace ls`, which caused the guest kernel to panic.

First, we tried to debug the error using traditional cyclic debugging techniques and standard `gdb`, i.e. not using time travel. We set a breakpoint in the kernel `panic` function and waited for the error. After the `panic` function was called, we traversed up the call stack to learn more about how the error occurred. According to our initial detective work, the guest kernel received a debug exception while in guest kernel mode. However, debug exceptions generated during guest kernel execution get trapped by the debugger prior to delivery. Since `gdb` had not received notification of an extraneous debugging exception, we deemed a guest kernel-mode debugging exception unlikely.

By performing additional call stack traversals, we determined that the current execution path originated from a function responsible for redirecting debugging exceptions to guest user-mode processes. This indicated that the debugging exception occurred in guest user mode, rather than in guest kernel mode as indicated by the virtual CPU mode variable. Based on that information, we concluded that either the call stack was corrupted, or the virtual mode variable was corrupted.

We sought to track changes to the virtual mode variable in two ways, both of which failed. First, we set a

forward watchpoint on the mode variable and re-ran the test. This failed because the mode variable was modified legitimately too often to examine each change. Second, we set a number of conditional breakpoints to try to narrow down the time of the corruption. With the conditional breakpoints in place, we re-ran the test case but it executed without any errors. We then gave up our vain attempt to track down this non-deterministic bug with cyclic debugging and switched to using our reverse debugging tools.

Our first step when using the reverse debugging tools was to set a reverse watchpoint on the virtual CPU mode variable. After trapping on the guest kernel panic, we were taken back to an exception handler where the variable was being changed intentionally. The new value indicated that the virtual machine was in virtual kernel mode when this exception was delivered. We reverse stepped to confirm that this was in fact the case, and then went forward to examine the subsequent execution. Since the virtual CPU mode variable is global, and the nested exception handler did not reset the value when it returned, the original exception handler (the user mode debugging exception) incorrectly determined that the debugging exception occurred while in virtual kernel mode. At this point it was clear that the exception handler should have included this variable as part of the context that is restored upon return.

6.3 mremap bug

Next, we debugged a bug in the mremap system call (CVE CAN-2003-0985), which occurs in the architecture-independent portion of Linux. This bug corrupts a process's address map when the process calls `mremap` with invalid arguments; it manifests later as a kernel panic when that process exits.

First, we tried to debug the error using traditional cyclic debugging and standard `gdb`, i.e. not using time travel. We attached `gdb` when the kernel called `panic`. We traversed up the call stack and discovered that the cause of the panic was a corrupted (zero-length) address map. Unfortunately, the kernel panic occurred long after the process's address map was corrupted, and we were unable to discern the point of the initial corruption. We thought to re-run the workload with watchpoints set on the memory locations of the variables denoting the start and end of the address map. However, these memory locations changed each run because they were allocated dynamically. Thus, while the bug crashed the system each time the program was run, the details of how the bug manifested were non-deterministic, and this prevented us from using traditional watchpoints. Even if the bug were completely deterministic, using forward watchpoints would require the programmer to step laboriously through each resulting trap during the entire run to see if the values at that trap were correct.

Reverse debugging provided a way to go easily from the kernel panic to the point at which the corruption initially occurred. After attaching `gdb` at the kernel panic, we set a reverse watchpoint on the memory locations of the variables denoting the start and end of the address map. This watchpoint took us to when the guest OS was executing the `mremap` that had been passed invalid arguments, and at this point it was obvious that mremap had failed to validate its arguments properly.

6.4 Soundcard device driver

Finally, we used TTVM to do reverse debugging for the Intel ICH4 soundcard driver. This driver uses `IN/OUT` instructions, interrupts, DMA reads and writes, and memory mapped I/O. We knew of no way to trigger a bug in this driver, so instead we used TTVM with reverse breakpoints and reverse single steps to navigate through the execution of the driver as though we were debugging.

Device drivers pose special problems for traditional debuggers. Traditional debuggers pause the execution of the device driver. However, since the device itself continues to run, it may generate interrupts whose timings are different enough to change the nature of the bug. The device may also require real-time responses that cannot be met by a paused driver.

TTVM avoids these difficulties during debugging because it does not need to use the device in order to replay and debug the driver. TTVM logs all interactions with the device, including I/O, interrupts, and DMA. During replay, the driver transitions through the same sequence of states as it went through during logging (i.e. while it was driving the device), regardless of timing or the state of the device. As a result, debugging can pause the driver during replay without altering its execution.

6.5 Lessons learned

We learned three main lessons from our experience debugging kernel bugs with standard `gdb` and TTVM.

First, we learned that many bugs are too fragile to find with cyclic debugging. Classic Heisenbugs [12] such as race conditions (Section 6.2) thwart cyclic debugging because they manifest only occasionally, depending on transient conditions such as timing. However, cyclic debugging often fails even for bugs that manifest each time a program runs, because the details of how they manifest change from run to run. Details like the internal state of an OS depend on numerous, hard-to-control variables, such as the sequence and scheduling order of all processes that have been run since boot. In the case of the mremap bug, minor changes to the internal OS state (the

address of dynamically allocated kernel memory) prevented us from using watchpoints during cyclic debugging.

In contrast, TTVM's reverse debugging makes even the most fragile of bugs perfectly repeatable. TTVM's deterministic replay ensures that the details of the internal OS state will remain consistent from run to run and thus enables the use of debugging commands that depend on fragile information.

Second, we experienced the poor match between standard debugging commands and most debugging scenarios. Standard watchpoints and breakpoints are best suited to go to a future point of possible interest. In contrast, programmers usually want to go to a prior point of possible interest, because they are following in reverse the chain of events between the execution of the buggy code and the ensuing detection of that error. Trying to go backward by re-running the workload with forward watchpoints and breakpoints is very clumsy without TTVM. If the bug is fragile, the bug may not manifest (or may not manifest in the same way) during each run. Even if the bug manifests in exactly the same way during each run, cyclic debugging forces a programmer to step manually through all spurious traps since the beginning of the run, or to run the program numerous times searching manually for the period of interest.

In contrast, TTVM's reverse debugging commands provided exactly the semantics we needed to find each of the bugs we encountered. For the kernel race bug and the mremap bug, the point of interest was the last time a variable changed before the error was detected. For the system call bug, the point of interest was a few instructions before the error was detected.

Third, we learned that, while standard debuggers seek to approximate time travel by traversing up the call stack, this form of time travel is neither complete nor reliable. Stack traversal is incomplete because it shows only the values of variables on the stack and because it shows those variables only at the time of their function's last invocation. For the mremap bug, the code that contained the error was executed during a prior system call and was not on the stack when the error was detected. Stack traversal is unreliable because it works only if the stack is intact. For the system call bug, the stack had been corrupted by an erroneous function call. Similarly, common buffer overflow attacks corrupt the stack and render stack traversal difficult. It is ironic that one of the most powerful techniques of standard debuggers depends on the partial correctness of the program being debugged.

In contrast, TTVM provides complete and reliable time travel. It is complete in that it can show the state of any variable at any time in the past. It is reliable in that it works without depending on the correctness of the program being debugged.

6.6 Interactivity of reverse debugging

To debug the bugs described in this section we triggered the error while logging, then replayed the virtual machine to diagnose the error. When replaying, we set the checkpoint interval to ten seconds. This checkpoint interval added reasonable runtime overhead for debugging (in fact, it added less overhead than some forward debugging commands, such as conditional breakpoints) and was short enough to support interactive performance for reverse commands.

We found the reverse commands to be quite interactive. Usually we used the reverse commands to step back a couple instructions or to go back to a recent breakpoint within the current checkpoint interval. This caused most of our checkpoint state to remain in the host file cache, which further sped up subsequent reverse commands. Restoring to the nearest checkpoint took under 1 second; replaying to the point of interest took five seconds on average (given the ten second checkpoint interval). Taking a reverse single step took about 12 seconds on average; this includes the time for both passes (Figure 3), i.e. restoring the checkpoint twice and replaying the remainder of the checkpoint interval twice. Overall, we found the speed of our reverse debugging commands fast enough to support interactive usage comfortably.

7 Related work

Our work draws on techniques from several areas, including replay of non-deterministic programs, virtual-machine replay, and reverse debugging. Our unique contribution is combining these techniques in a way that enables powerful debugging capabilities that have not been available previously for operating systems.

Researchers have worked to replay non-deterministic programs through various approaches. The events of different threads can be replayed at different levels, including logging accesses to shared objects [14], logging the scheduling order of multi-threaded programs on a uniprocessor [18], or logging physical memory accesses in hardware [3]. Other researchers have worked to optimize the amount of data logged [17].

Virtual-machine replay has been used for non-debugging purposes. Hypervisor used virtual-machine replay to synchronize the state of a backup machine to provide fault tolerance [6]. ReVirt used virtual-machine replay to enable detailed intrusion analysis [9]. Our work applies virtual-machine replay to achieve a new capability, which is reverse debugging of operating systems. TTVM also supports additional features over prior virtual-machine replay systems. TTVM supports the ability to run, log, and replay real device drivers in the guest OS, and TTVM can travel quickly forward and

backward in time through its use of checkpoints and undo and redo logs. In contrast, ReVirt supported only a single checkpoint of a powered-off virtual machine, and Hypervisor did not need to support time travel at all (it only supported replay within an epoch).

Reverse execution has been discussed in the programming community for many years [22, 10, 2], and some have integrated this capability into a debugger [5, 7]. However, no prior system has attempted to support reverse debugging for operating systems. Most prior systems focused on reversing deterministic programs rather than replaying non-deterministic effects such as interrupts and thread scheduling [10, 5, 2, 7]. Some systems logged and replayed data by instrumenting programs at the source code or assembly language level [2, 5, 7]. Unfortunately, this approach is difficult to apply for operating systems, which use self-modifying code when loading new application programs or kernel modules. Operating systems are perhaps the most difficult type of software system to reverse and replay because of their many sources of non-determinism, their large and complex state, their need to work for code that is loaded on the fly, and their long running time.

The system that is closest in capability to ours is Simics [15]. Simics is a machine simulator on which one can run operating systems and applications. Simics supports replay of non-deterministic inputs and has an interface to a debugger, although it is unclear whether Simics supports higher-level commands such as reverse breakpoint. One feature of Simics is that it can run unmodified operating systems, whereas current open-source VMMs such as UML require modifications to the hardware-dependent layer of their guest OSs. However, while Simics has similar capabilities to TTVM, it is drastically slower, and this makes debugging long runs impractical. On a 750 MHz Ultrasparc III, Simics executes 2-6 million x86 instructions per second (several hundred times slower) [15], whereas virtual machines typically incur a slowdown of less than 2x.

8 Conclusions

We have described the design and implementation of a time-traveling virtual machine and shown how to use TTVM to add powerful capabilities for debugging operating systems. We integrated TTVM with a general-purpose debugger, implementing commands such as reverse breakpoint, reverse watchpoint, and reverse step.

TTVM added reasonable overhead in the context of debugging. The logging needed to support time travel for two OS-intensive workloads added 10-12% in running time and 13-190 KB/sec in log space. Taking checkpoints at a rate appropriate for long runs added less than 4% overhead. Taking frequent checkpoints to prepare for debugging a portion of a run added less than 60% overhead and enabled reverse debugging commands to complete in about 12 seconds.

We used TTVM and our new reverse debugging commands to fix three OS bugs that were difficult to find with standard debugging tools. We found the reverse debugging commands to be intuitive to understand, fast and easy to use, and extremely helpful in finding and fixing real OS bugs.

References

[1] The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. Technical report, Intel Corporation, 2004.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, 8(3), May 1991.

[3] 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.

[4] 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.

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

[6] 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.

[7] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible Debugging Using Program Instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, August 2001.

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

[9] 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.

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

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

[12] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the 1986 Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.

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

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

[15] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

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

[17] R. H. B. Netzer and M. H. Weaver. Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs. In *Proceedings of the 1994 Conference on Programming Language Design and Implementation (PLDI)*, June 1994.

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

[19] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

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

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

[22] M. V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566, September 1973.