USENIX Association

Proceedings of the General Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

Operating System Support for Virtual Machines

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

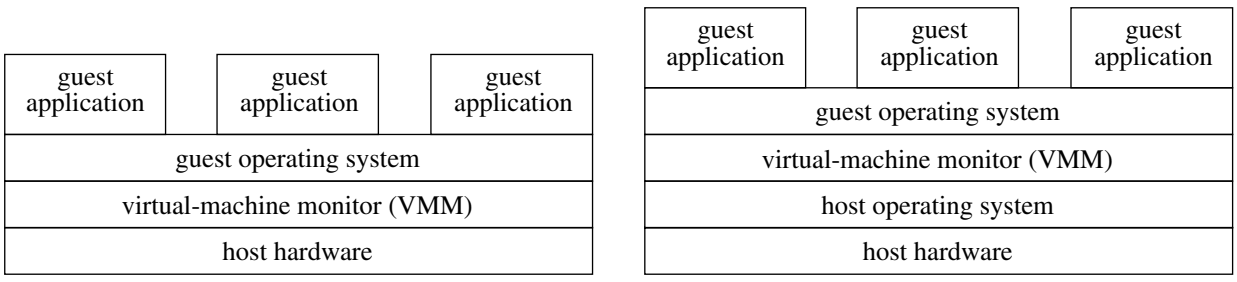
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
http://www.eecs.umich.edu/CoVirt

Abstract: A virtual-machine monitor (VMM) is a useful technique for adding functionality below existing operating system and application software. One class of VMMs (called Type II VMMs) builds on the abstractions provided by a host operating system. Type II VMMs are elegant and convenient, but their performance is currently an order of magnitude slower than that achieved when running outside a virtual machine (a standalone system). In this paper, we examine the reasons for this large overhead for Type II VMMs. We find that a few simple extensions to a host operating system can make it a much faster platform for running a VMM. Taking advantage of these extensions reduces virtualization overhead for a Type II VMM to 14-35% overhead, even for workloads that exercise the virtual machine intensively.

1. Introduction

A virtual-machine monitor (VMM) is a layer of software that emulates the hardware of a complete computer system (Figure 1). The abstraction created by the VMM is called a virtual machine. The hardware emulated by the VMM typically is similar or identical to the hardware on which the VMM is running.

Virtual machines were first developed and used in the 1960s, with the best-known example being IBM's VM/370 [Goldberg74]. Several properties of virtual machines have made them helpful for a wide variety of uses. First, they can create the illusion of multiple virtual machines on a single physical machine. These multiple virtual machines can be used to run applications on different operating systems, to allow students to experiment conveniently with building their own operating system [Nieh00], to enable existing operating systems to run on shared-memory multiprocessors [Bugnion97], and to simulate a network of independent computers. Second, virtual machines can provide a software environment for debugging operating systems that is more convenient than using a physical machine. Third, virtual machines provide a convenient interface for adding functionality, such as fault injection [Buchacker01], primary-backup replication [Bressoud96], and undoable disks. Finally, a VMM provides strong isolation



Type I VMM Type II VMM

Figure 1: Virtual-machine structures. A virtual-machine monitor is a software layer that runs on a host platform and provides an abstraction of a complete computer system to higher-level software. The host platform may be the bare hardware (Type I VMM) or a host operating system (Type II VMM). The software running above the virtual-machine abstraction is called guest software (operating system and applications).

between virtual-machine instances. This isolation allows a single server to run multiple, untrusted applications safely [Whitaker02, Meushaw00] and to provide security services such as monitoring systems for intrusions [Chen01, Dunlap02, Barnett02].

As a layer of software, VMMs build on a lower-level hardware or software platform and provide an interface to higher-level software (Figure 1). In this paper, we are concerned with the lower-level platform that supports the VMM. This platform may be the bare hardware, or it may be a host operating system. Building the VMM directly on the hardware lowers overhead by reducing the number of software layers and enabling the VMM to take full advantage of the hardware capabilities. On the other hand, building the VMM on a host operating system simplifies the VMM by allowing it to use the host operating system's abstractions.

Our goal for this paper is to examine and reduce the performance overhead associated with running a VMM on a host operating system. Building it on a standard Linux host operating system leads to an order of magnitude performance degradation compared to running outside a virtual machine (a *standalone system*). However, we find that a few simple extensions to the host operating system reduces virtualization overhead to 14-35% overhead, which is comparable to the speed of virtual machines that run directly on the hardware.

The speed of a virtual machine plays a large part in determining the domains for which virtual machines can be used. Using virtual machines for debugging, student projects, and fault-injection experiments can be done even if virtualization overhead is quite high (e.g. 10x slowdown). However, using virtual machine in production environments requires virtualization overhead to be much lower. Our CoVirt project on computer security depends on running all applications inside a virtual machine [Chen01]. To keep the system usable in a production environment, we would like the speed of our virtual machine to be within a factor of 2 of a standalone system.

The paper is organized as follows. Section 2 describes two ways to classify virtual machines, focusing on the higher-level interface provided by the VMM and the lower-level platform upon which the VMM is built. Section 3 describes UMLinux, which is the VMM we use in this paper. Section 4 describes a series of extensions to the host operating system that enable virtual machines built on the host operating system to approach the speed of those that run directly on the hardware. Section 5 evaluates the performance benefits achieved by each host OS extension. Section 6 describes related work, and Section 7 concludes.

2. Virtual machines

Virtual-machine monitors can be classified along many dimensions. This section classifies VMMs along two dimensions: the higher-level interface they provide and the lower-level platform they build upon.

The first way we can classify VMMs is according to how closely the higher-level interface they provide matches the interface of the physical hardware. VMMs such as VM/370 [Goldberg74] for IBM mainframes and VMware ESX Server [Waldspurger02] and VMware Workstation [Sugerman01] for x86 processors provide an abstraction that is identical to the hardware underneath the VMM. Simulators such as Bochs [Boc] and Virtutech Simics [Magnusson95] also provide an abstraction that is identical to physical hardware, although the hardware they simulate may differ from the hardware on which they are running.

Several aspects of virtualization make it difficult or slow for a VMM to provide an interface that is identical to the physical hardware. Some architectures include instructions whose behavior depends on whether the CPU is running in privileged or user mode (sensitive instructions), yet which can execute in user mode without causing a trap to the VMM [Robin00]. Virtualizing these sensitive-but-unprivileged instructions generally requires binary instrumentation, which adds significant complexity and may add significant overhead. In addition, emulating I/O devices at the low-level hardware interface (e.g. memory-mapped I/O) causes execution to switch frequently between the guest operating system accessing the device and the VMM code emulating the device. To avoid the overhead associated with emulating a low-level device interface, most VMMs encourage or require the user to run a modified version of the guest operating system. For example, the VAX VMM security kernel [Karger91], VMware Workstation's guest tools [Sugerman01], and Disco [Bugnion97] all add special drivers in the guest operating system to accelerate the virtualization of some devices. VMMs built on host operating systems often require additional modifications to the guest operating system. For example, the original version of SimOS adds special signal handlers to support virtual interrupts and requires relinking the guest operating system into a different range of addresses [Rosenblum95]; similar changes are needed by User-Mode Linux [Dike00] and UMLinux [Buchacker01].

Other virtualization strategies make the higher-level interface further different from the underlying

hardware. The Denali isolation kernel does not support instructions that are sensitive but unprivileged, adds several virtual instructions and registers, and changes the memory management model [Whitaker02]. Microkernels provide higher-level services above the hardware to support abstractions such as threads and inter-process communication [Golub90]. The Java virtual machine defines a virtual architecture that is completely independent from the underlying hardware.

A second way to classify VMMs is according to the platform upon which they are built [Goldberg73]. *Type I* VMMs such as IBM's VM/370, Disco, and VMware's ESX Server are implemented directly on the physical hardware. *Type II* VMMs are built completely on top of a host operating system. SimOS, User-Mode Linux, and UMLinux are all implemented completely on top of a host operating system. Other VMMs are a hybrid between Type I and II: they operate mostly on the physical hardware but use the host OS to perform I/O. For example, VMware Workstation [Sugerman01] and Connectix VirtualPC [Con01] use the host operating system to access some virtual I/O devices.

A host operating system makes a very convenient platform upon which to build a VMM. Host operating system provide a set of abstractions that map closely to each part of a virtual machine [Rosenblum95]. A host process provides a sequential stream of execution similar to a CPU; host signals provide similar functionality to interrupts; host files and devices provide similar functionality to virtual I/O devices; host memory mapping and protection provides similar functionality to a virtual MMU. These features make it possible to implement a VMM as a normal user process with very little code.

Other reasons contribute to the attractiveness of using a Type II VMM. Because a Type II VMM runs as a normal process, the developer or administrator of the VMM can use the full power of the host operating system to monitor and debug the virtual machine's execution. For example, the developer or administrator can examine or copy the contents of the virtual machine's I/O devices or memory or attach a debugger to the virtual-machine process. Finally, the simplicity of Type II VMMs and the availability of several good open-source implementations make them an excellent platform for experimenting with virtual-machine services.

A potential disadvantage of Type II VMMs is performance. Current host operating systems do not provide sufficiently powerful interfaces to the bare hardware to support the intensive usage patterns of VMMs. For example, compiling the Linux 2.4.18 kernel inside the UMLinux virtual machine takes 18 times as long as compiling it directly on a Linux host operating system. VMMs that run directly on the bare hardware achieve much lower performance overhead. For example, VMware Workstation 3.1 compiles the Linux 2.4.18 kernel with only a 30% overhead relative to running directly on the host operating system.

The goal of this paper is to examine and reduce the order-of-magnitude performance overhead associated with running a VMM on a host operating system. We find that a few simple extensions to a host operating system can make it a much faster platform for running a VMM, while preserving the conceptual elegance of the Type II approach.

3. UMLinux

To conduct our study, we use a Type II VMM called UMLinux [Buchacker01]. UMLinux was developed by researchers at the University of Erlangen-Nürnberg for use in fault-injection experiments. UMLinux is a Type II VMM: the guest operating system and all guest applications run as a single process (the *guest-machine process*) on a host Linux operating system. UMLinux provides a higher-level interface to the guest operating system that is similar but not identical to the underlying hardware. As a result, the machine-dependent portion of the guest Linux operating system must be modified to use the interface provided by the VMM. Simple device drivers must be added to interact with the host abstractions used to implement the devices for the virtual machine; a few assembly-language instructions (e.g. `iret` and `in`/`out`) must be replaced with function calls to emulation code; and the guest kernel must be relinked into a different address range [Hoxer02]. About 17,000 lines of code were added to the guest kernel to work on the new platform. Applications compiled for the host operating system work without modification on the guest operating system.

UMLinux uses functionality from the host operating system that maps naturally to virtual hardware. The guest-machine process serves as a virtual CPU; host files and devices serve as virtual I/O devices; a host TUN/TAP device serves as a virtual network; host signals serve as virtual interrupts; and host memory mapping and protection serve as a virtual MMU. The virtual machine's memory is provided by a host file that is mapped into different parts of the guest-machine process's address space. We store this host file in a memory file system (ramfs) to avoid needlessly writing to disk the virtual machine's transient state.

The address space of the guest-machine process differs from a normal host process because it contains

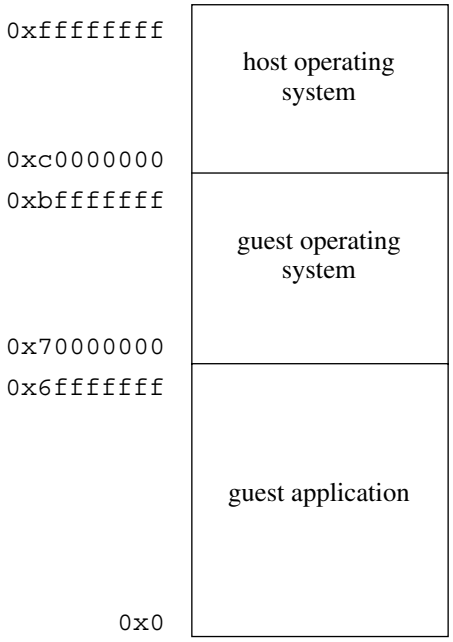


Figure 2: UMLinux address space. As with all Linux processes, the host kernel address space occupies [0xc0000000, 0xffffffff], and the host user address space occupies [0x0, 0xc0000000). The guest kernel occupies the upper portion of the host user space [0x70000000, 0xc0000000), and the current guest application occupies the remainder of the host user space [0x0, 0x70000000).

both the host and guest operating system address ranges (Figure 2). In a standard Linux process, the operating system occupies addresses [0xc0000000, 0xffffffff] while the application is given [0x0, 0xc0000000). Because the UMLinux guest-machine process must hold both the host and guest operating systems, the address space for the guest operating system must be moved to occupy [0x70000000, 0xc0000000), which leaves [0x00000000, 0x70000000) for guest applications. The guest kernel memory is protected using host `mmap` and `munmap` system calls. To facilitate this protection, UMLinux maintains a virtual current privilege level, which is analogous to the x86 current privilege level. This is used to differentiate between guest user and guest kernel modes, and the guest kernel memory will be accessible or protected according to the virtual privilege level.

Figure 3 shows the basic system structure of UMLinux. In addition to the guest-machine process, UMLinux uses a VMM process to implement the VMM.

The VMM process serves two purposes: it redirects to the guest operating system signals and system

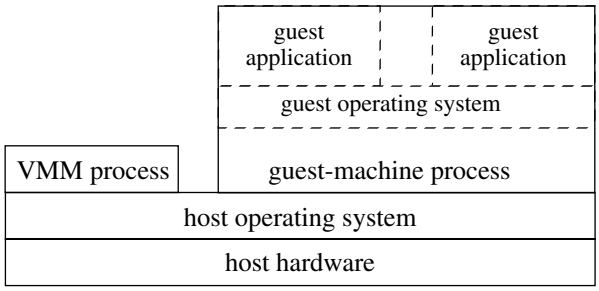


Figure 3: UMLinux system structure. UMLinux uses two host processes. The guest-machine process executes the guest operating system and all guest applications. The VMM process uses `ptrace` to mediate access between the guest-machine process and the host operating system.

calls that would otherwise go to the host operating system, and it restricts the set of system calls allowed by the guest operating system. The VMM process uses `ptrace` to mediate access between the guest-machine process and the host operating system. Figure 4 shows the sequence of steps taken by UMLinux when a guest application issues a system call.

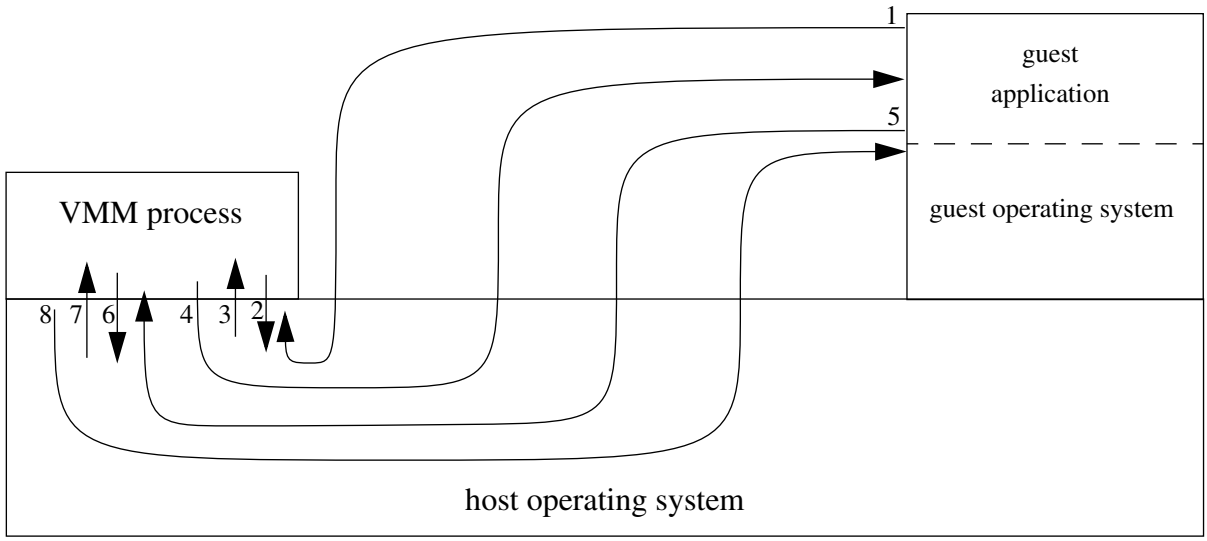
The VMM process is also invoked when the guest kernel returns from its SIGUSR1 handler and when the guest kernel protects its address space from the guest application process. A similar sequence of context switches occurs on each memory, I/O, and timer exception received by the guest-machine process.

4. Host OS support for Type II VMMs

A host operating system makes an elegant and convenient base upon which to build and run a VMM such as UMLinux. Each virtual hardware component maps naturally to an abstraction in the host OS, and the administrator can interact conveniently with the guest-machine process just as it does with other host processes. However, while a host OS provides sufficient functionality to support a VMM, it does not provide the primitives needed to support a VMM *efficiently.*

In this section, we investigate three bottlenecks that occur when running a Type II VMM, and we eliminate these bottlenecks through simple changes to the host OS.

We find that three bottlenecks are responsible for the bulk of the virtualization overhead. First, UMLinux's system structure with two separate host processes causes an inordinate number of context switches on the host. Second, switching between the guest kernel and the guest user space generates a large number of



1. guest application issues system call; intercepted by VMM process via `ptrace`
2. VMM process changes system call to no-op (getpid)
3. getpid returns; intercepted by VMM process
4. VMM process sends SIGUSR1 signal to guest SIGUSR1 handler
5. guest SIGUSR1 handler calls `mmap` to allow access to guest kernel data; intercepted by VMM process
6. VMM process allows `mmap` to pass through
7. `mmap` returns to VMM process
8. VMM process returns to guest SIGUSR1 handler, which handles the guest application's system call

Figure 4: Guest application system call. This picture shows the steps UMLinux takes to transfer control to the guest operating system when a guest application process issues a system call. The `mmap` call in the SIGUSR1 handler must reside in guest user space. For security, the rest of the SIGUSR1 handler should reside in guest kernel space. The current UMLinux implementation includes an extra section of trampoline code to issue the `mmap`; this trampoline code is started by manipulating the guest machine process's context and finishes by causing a breakpoint to the VMM process; the VMM process then transfers control back to the guest-machine process by sending a SIGUSR1.

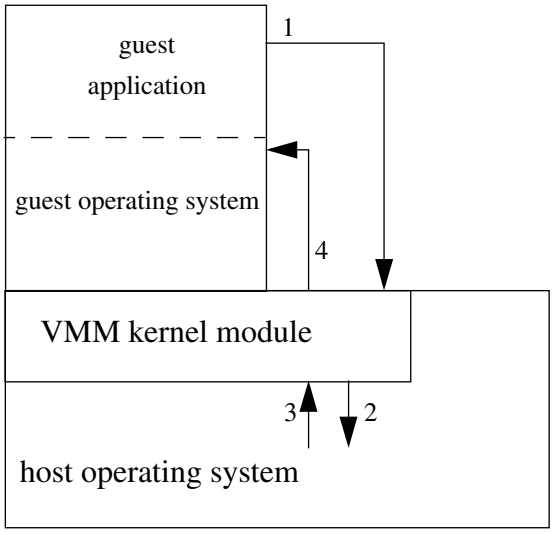
memory protection operations. Third, switching between two guest application processes generates a large number of memory mapping operations.

4.1. Extra host context switches

The VMM process in UMLinux uses `ptrace` to intercept key events (system calls and signals) executed by the guest-machine process. `ptrace` is a powerful tool for debugging, but using it to create a virtual machine causes the host OS to context switch frequently between the guest-machine process and the VMM process (Figure 4).

We can eliminate most of these context switches by moving the VMM process's functionality into the host kernel. We encapsulate the bulk of the VMM process functionality in a VMM loadable kernel module. We also modified a few lines in the host kernel's system call and signal handling to transfer control to the VMM kernel module when the guest-machine process executes a system call or receives a signal. The VMM kernel module and other hooks in the host kernel were implemented in 150 lines of code (not including comments).

Moving the VMM process's functionality into the host kernel drastically reduces the number of context switches in UMLinux. For example, transferring control to the guest kernel on a guest system call can be done in just two context switches (Figure 5). It also simplifies the system conceptually, because the VMM kernel module has more control over the guest-machine process than is provided by `ptrace`. For example, the VMM kernel module can change directly the protections of the guest-machine process's address space, whereas the ptracing VMM process must cause the guest-machine process to make multiple system calls to change protections.



1. guest application issues system call; intercepted by VMM kernel module
2. VMM kernel module calls `mmap` to allow access to guest kernel data
3. `mmap` returns to VMM kernel module
4. VMM kernel module sends SIGUSR1 to guest SIGUSR1 handler

Figure 5: Guest application system call with VMM kernel module. This picture shows the steps taken by UMLinux with a VMM kernel module to transfer control to the guest operating system when a guest application issues a system call.

4.2. Protecting guest kernel space from guest application processes

The guest-machine process switches frequently between guest user mode and guest kernel mode. The guest kernel is invoked to service guest system calls and other exceptions issued by a guest application process and to service signals initiated by virtual I/O devices. Each time the guest-machine process switches from guest kernel mode to guest user mode, it must first prevent access to the guest kernel's portion of the address space [`0x70000000`, `0xc0000000`). Similarly, each time the guest-machine process switches from guest user mode to guest kernel mode, it must first enable access to the guest kernel's portion of the address space. The guest-machine process performs these address space manipulations by making the host system calls `mmap`, `munmap`, and `mprotect`.

Unfortunately, calling `mmap`, `munmap`, or `mprotect` on large address ranges incurs significant overhead, especially if the guest kernel accesses many pages in its address space. In contrast, a standalone host machine incurs very little overhead when switching between user mode and kernel mode. The page table on x86 processors need not change when switching between kernel mode and user mode, because the page table entry for a page can be set to simultaneously allow kernel-mode access and prevent user-mode access.

We developed two solutions that use the x86 paged segments and privilege modes to eliminate the overhead incurred when switching between guest kernel mode and guest user mode. Linux normally uses paging as its primary mechanism for translation and protection, using segments only to switch between privilege levels. Linux uses four segments: kernel code segment, kernel data segment, user code segment, and user data segment. Normally, all four segments span the entire address range. Linux normally runs all host user code in CPU privilege ring 3 and runs host kernel code in CPU privilege ring 0. Linux uses the supervisor-only bit in the page table to prevent code running in CPU privilege ring 3 from accessing the host operating system's data (Figure 6).

Our first solution protects the guest kernel space from guest user code by changing the bound on the user code and data segments (Figure 7). When the guest-machine process is running in guest user mode, the VMM kernel module shrinks the user code and data segments to span only [`0x0`, `0x70000000`). When the guest-machine process is running in guest kernel mode, the VMM kernel module grows the user code and data segments to its normal range of [`0x0`, `0xffffffff`]. This solution added only 20 lines of code to the VMM kernel module and is the solution we currently use.

One limitation of the first solution is that it assumes the guest kernel space occupies a contiguous region directly below the host kernel space. Our second solution allows the guest kernel space to occupy arbitrary ranges of the address space within [`0x0`, `0xc0000000`) by using the page table's supervisor-only bit to distinguish between guest kernel mode and guest user mode (Figure 8). In this solution, the VMM kernel module marks the guest kernel's pages as accessible only by supervisor code (ring 0-2), then runs the guest-machine process in ring 1 while in guest kernel mode. When running in ring 1, the CPU can access pages marked as supervisor in the page table, but it cannot execute privileged instructions (such as changing the segment descriptor). To prevent the guest-machine process from accessing host kernel space, the VMM kernel module shrinks the user code and data segment to span

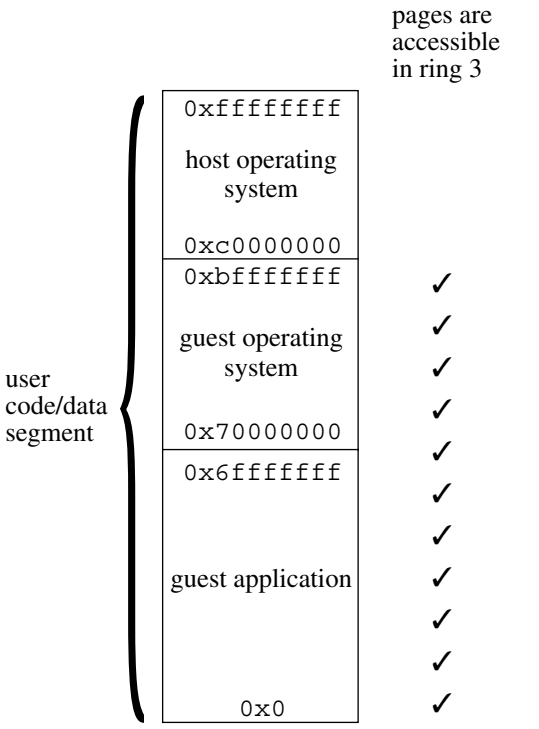


Figure 6: Segment and page protections when running a normal Linux host processes. A normal Linux host process runs in CPU privilege ring 3 and uses the user code and data segment. The segment bounds allow access to all addresses, but the supervisor-only bit in the page table prevents the host process from accessing the host operating system's data. In order to protect the guest kernel's data with this setup, the guest-machine process must `munmap` or `mprotect` [`0x70000000`, `0xc0000000`) before switching to guest user mode.

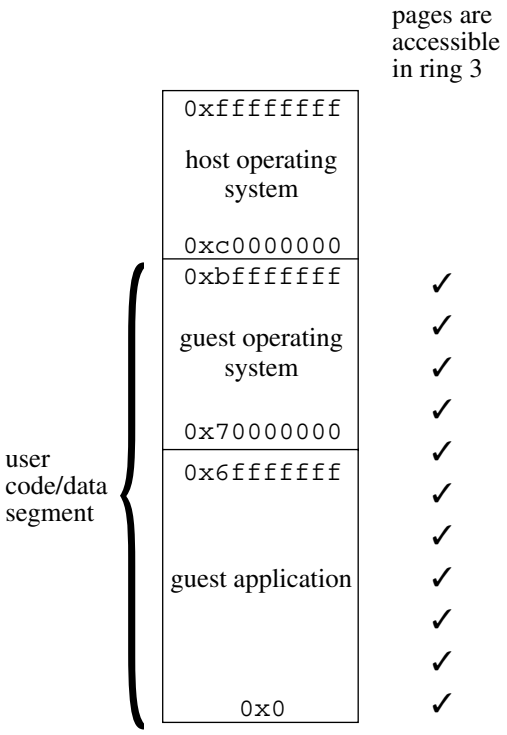


Figure 7: Segment and page protections when running the guest-machine process in guest user mode (solution 1). This solution protects the guest kernel space from guest application processes by changing the bound on the user code and data segments to [`0x0`, `0x70000000`) when running guest user code. When the guest-machine process switches to guest kernel mode, the VMM kernel module grows the user code and data segments to its normal range of [`0x0`, `0xffffffff`].

only [`0x0`, `0xc0000000`). The guest-machine process runs in ring 3 while in guest user mode, which prevents guest user code from accessing the guest kernel's data. This allows the VMM kernel module to protect arbitrary pages in [`0x0`, `0xc0000000`) from guest user mode by setting the supervisor-only bit on those pages. It does still require the host kernel and user address ranges to each be contiguous.

4.3. Switching between guest application processes

A third bottleneck in a Type II VMM occurs when switching address spaces between guest application processes. Changing guest address spaces means changing the current mapping between guest virtual pages and the page in the virtual machine's "physical" memory file. Changing this mapping is done by calling `munmap` for the outgoing guest application process's virtual address space, then calling `mmap` for each resident virtual page in the incoming guest application process. UMLinux minimizes the calls to `mmap` by doing it on demand, i.e. as the incoming guest application process faults in its address space. Even with this optimization, however, UMLinux generates a large number of calls to `mmap`, especially when the working sets of the guest application processes are large.

To improve the speed of guest context switches, we enhance the host OS to allow a single process to maintain several address space definitions. Each address space is defined by a separate set of page tables, and the guest-machine processes switches between address space definitions via a new host system call `switch-guest`. To switch address space definitions, `switch-guest` needs only to change the pointer to the current first-level page table. This task is much faster than mmap'ing each virtual page of the incoming guest application process. We modify the guest kernel to use `switchguest` when context switching from one guest application process to another. We reuse initialized

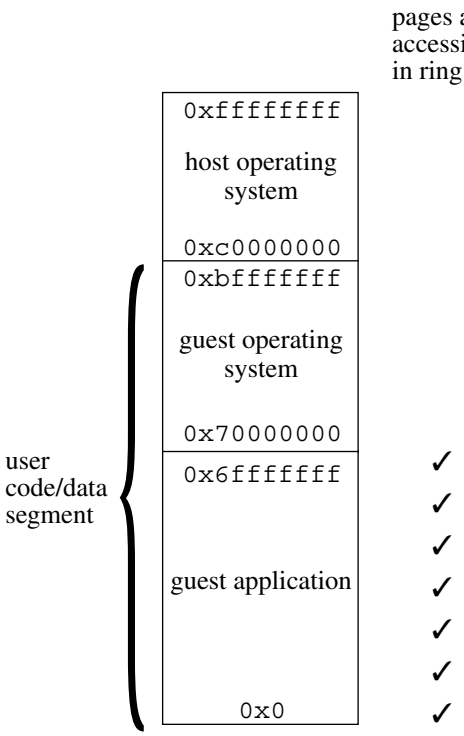


Figure 8: Segment and page protections when running the guest-machine process (solution 2). This solution protects the guest kernel space from guest application processes by marking the guest kernel's pages as accessible only by code running in CPU privilege ring 0-2 and running the guest-machine process in ring 1 when executing guest kernel code. To prevent the guest-machine process from accessing host kernel space, the VMM kernel module shrinks the user code and data segment to span only [`0x0`, `0xc0000000`).

address space definitions to minimize the overhead of creating guest application processes. We take care to prevent the guest-machine process from abusing `switchguest` by limiting it to 1024 different address spaces and checking all parameters carefully. This optimization added 340 lines of code to the host kernel.

5. Performance results

This section evaluates the performance benefits achieved by each of the optimizations described in Section 4.

We first measure the performance of three important primitives: a null system call, switching between two guest application processes (each with a 64 KB working set), and transferring 10 MB of data using TCP across a 100 Mb/s Ethernet switch. The first two of these microbenchmarks come from the lmbench suite [McVoy96].

We also measure performance on three macrobenchmarks. POV-Ray is a CPU-intensive ray-tracing program. We render the benchmark image from the POV-Ray distribution at quality 8. kernel-build compiles the complete Linux 2.4.18 kernel (make bzImage). SPECweb99 measures web server performance, using the 2.0.36 Apache web server. We configure SPECweb99 with 15 simultaneous connections spread over two clients connected to a 100 Mb/s Ethernet switch. kernel-build and SPECweb99 exercise the virtual machine intensively by making many system calls. They are similar to the I/O-intensive and kernel-intensive workloads used to evaluate Cellular Disco [Govil00]. All workloads start with a warm guest file cache. Each results represents the average of 5 runs. Variance across runs is less than 3%.

All experiments are run on an computer with an AMD Athlon 1800+ CPU, 256 MB of memory, and a Samsung SV4084 IDE disk. The guest kernel is Linux 2.4.18 ported to UMLinux, and the host kernels for UMLinux are all Linux 2.4.18 with different degrees of support for VMMs. All virtual machines are configured with 192 MB of "physical" memory. The virtual hard disk for UMLinux is stored on a raw disk partition on the host to avoid double buffering the virtual disk data in the guest and host file caches and to prevent the virtual machine from benefitting unfairly from the host's file cache. The host and guest file systems have the same versions of all software (based on RedHat 6.2).

We measure baseline performance by running directly on the host operating system (standalone). The host uses the same hardware and software installation as the virtual-machine systems and has access to the full 256 MB of host memory.

We use VMware Workstation 3.1 to illustrate the performance of VMMs that are built directly on the host hardware. We chose VMware Workstation because it executes mostly on host hardware and because it is regarded widely as providing excellent performance. However, note that VMware Workstation may be slower than a Type I VMM that is ideal for the purposes of comparing with UMLinux. First, VMware Workstation issues I/O through the host OS rather than controlling the host I/O devices directly. Second, unlike UMLinux, VMware Workstation can support unmodified guest operating systems, and this capability forces VMware Workstation to do extra work to provide the same interface to the guest OS as the host hardware does. The configuration for VMware Workstation matches that of the other virtual-machine systems, except that VMware

Workstation uses the host disk partition's cacheable block device for its virtual disk.

Figures 9 and 10 summarize results from all performance experiments.

The original UMLinux is hundreds of times slower for null system calls and context switches and is not able to saturate the network. UMLinux is 8x as slow as the standalone host on SPECweb99, 18x as slow as the standalone host on kernel-build and 10% slower than the standalone host on POV-Ray. Because POV-Ray is compute-bound, it does not interact much with the guest kernel and thus incurs little virtualization overhead. The overheads for SPECweb99 and kernel-build are higher because they issue more guest kernel calls, each of which must be trapped by the VMM kernel module and reflected back to the guest kernel by sending a signal.

VMMs that are built directly on the hardware execute much faster than a Type II VMM without host OS support. VMware Workstation 3.1 executes a null system call nearly as fast as the standalone host, can saturate the network, and is within a factor of 5 of the context switch time for a standalone host. VMware Workstation 3.1 incurs an overhead of 6-30% on the intensive macrobenchmarks (SPECweb99 and kernel-build).

Our first optimization (Section 4.1) moves the VMM functionality into the kernel. This improves performance by a factor of about 2-3 on the microbenchmarks, and by a factor of about 2 on the intensive macrobenchmarks.

Our second optimization (Section 4.2) uses segment bounds to eliminate the need to call `mmap`, `munmap`, and `mprotect` when switching between guest kernel mode and guest user mode. Adding this optimization improves performance on null system calls and context switches by another factor of 5 (beyond the performance with just the first optimization) and enables UMLinux to saturate the network. Performance on the two intensive macrobenchmarks improves by a factor of 3-4.

Our final optimization (Section 4.3) maintains multiple address space definitions to speed up context switches between guest application processes. This optimization has little effect on benchmarks with only one main application process, but it has a dramatic affect on benchmarks with more than one main application process. Adding this optimization improves the context switch microbenchmark by a factor of 13 and improves kernel-build by a factor of 2.

With all three host OS optimizations to support VMMs, UMLinux runs all macrobenchmarks well within our performance target of a factor of 2 relative to standalone. POV-Ray incurs 1% overhead; kernel-build incurs 35% overhead; and SPECweb99 incurs 14% overhead. These overheads are comparable to those attained by VMware Workstation 3.1.

The largest remaining source of virtualization overhead for kernel-build is the cost and frequency of handling memory faults. kernel-build creates a large number of guest application processes, each of which maps its executable pages on demand. Each demand-mapped page causes a signal to be delivered to the guest kernel, which must then ask the host kernel to map the new page. In addition, UMLinux currently does not support the ability to issue multiple outstanding I/Os on the host. We plan to update the guest disk driver to take advantage of non-blocking I/O when it becomes available on Linux.

6. Related work

User-Mode Linux is a Type II VMM that is very similar to UMLinux [Dike00]. Our discussion of User-Mode Linux assumes a configuration that protects guest kernel memory from guest application processes (jail mode). The major technical difference between the User-Mode Linux and UMLinux is that User-Mode Linux uses a separate host process for each guest application process, while UMLinux runs all guest code in a single host process. Assigning each guest application process to a separate host process technique speeds up context switches between guest application processes, but it leads to complications such as keeping the shared portion of the guest address spaces consistent and difficult synchronization issues when switching guest application processes [Dike02a].

User-Mode Linux in jail mode is faster than UMLinux (without host OS support) on context switches (157 vs. 2029 microseconds) but slower on system calls (296 vs. 96 microseconds) and network transfers (54 vs. 39 seconds). User-Mode Linux in jail mode is faster on kernel-build (1309 vs. 2294 seconds) and slower on SPECweb99 (200 vs. 172 seconds) than UMLinux without host OS support.

Concurrently with our work on host OS support for VMMs, the author of User-Mode Linux proposed modifying the host OS to support multiple address space definitions for a single host process [Dike02a]. Like the optimization in Section 4.3, this would speed up switches between guest application processes and allow User-Mode Linux to run all guest code in a single host

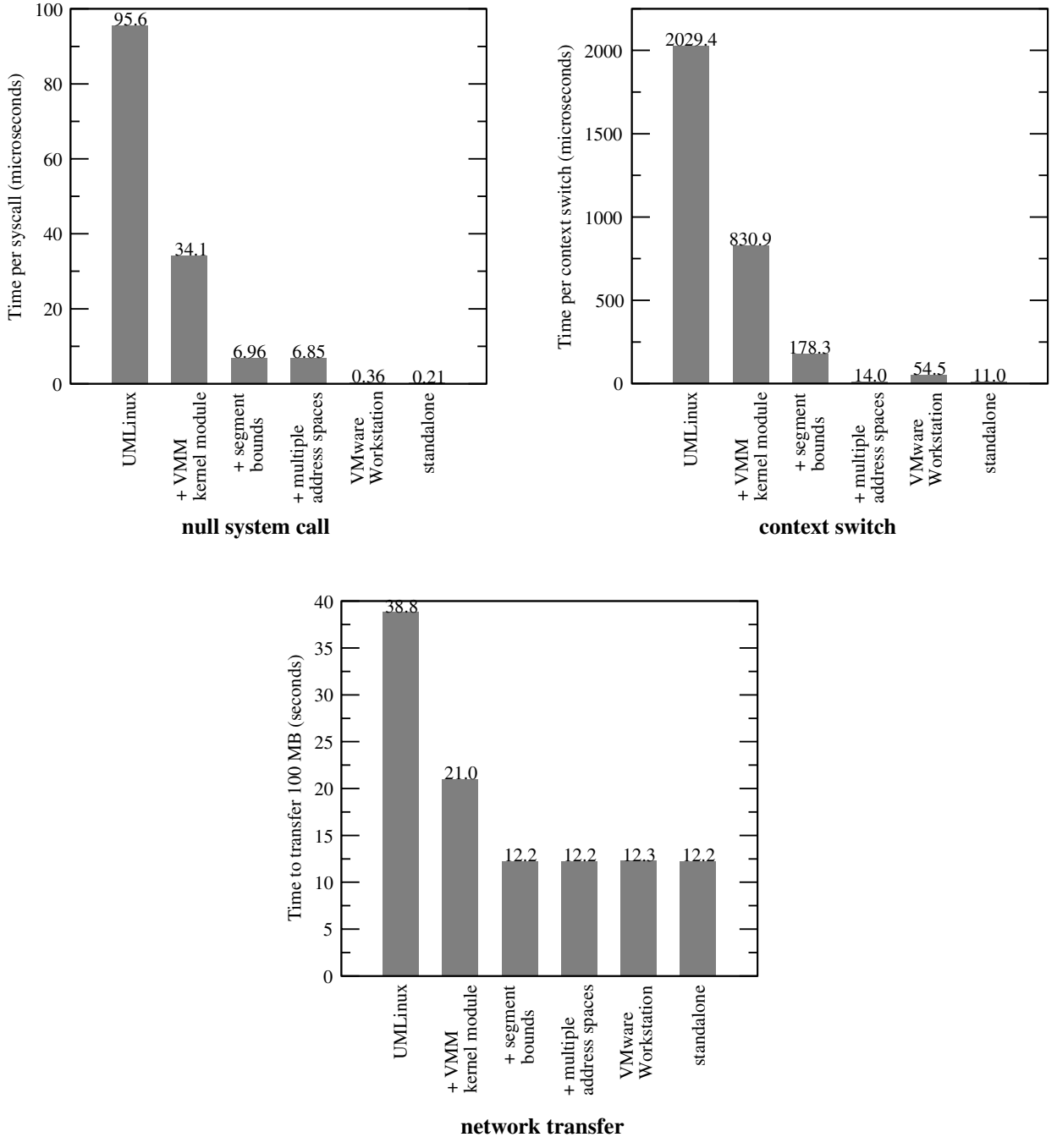


Figure 9: Microbenchmark results. This figure compares the performance of different virtual-machine monitors on three microbenchmarks: a null guest system call, context switching between two 64 KB guest application processes, and receiving 10 MB of data over the network. The first four bars represent the performance of UMLinux with increasing support from the host OS. Each optimization level is **cumulative**, i.e. it includes all optimizations of the bars to the left. The performance of a standalone host (no VMM) is shown for reference. Without support from the host OS, UMLinux is much slower than a standalone host. Adding three extensions to the host OS improves the performance of UMLinux dramatically.

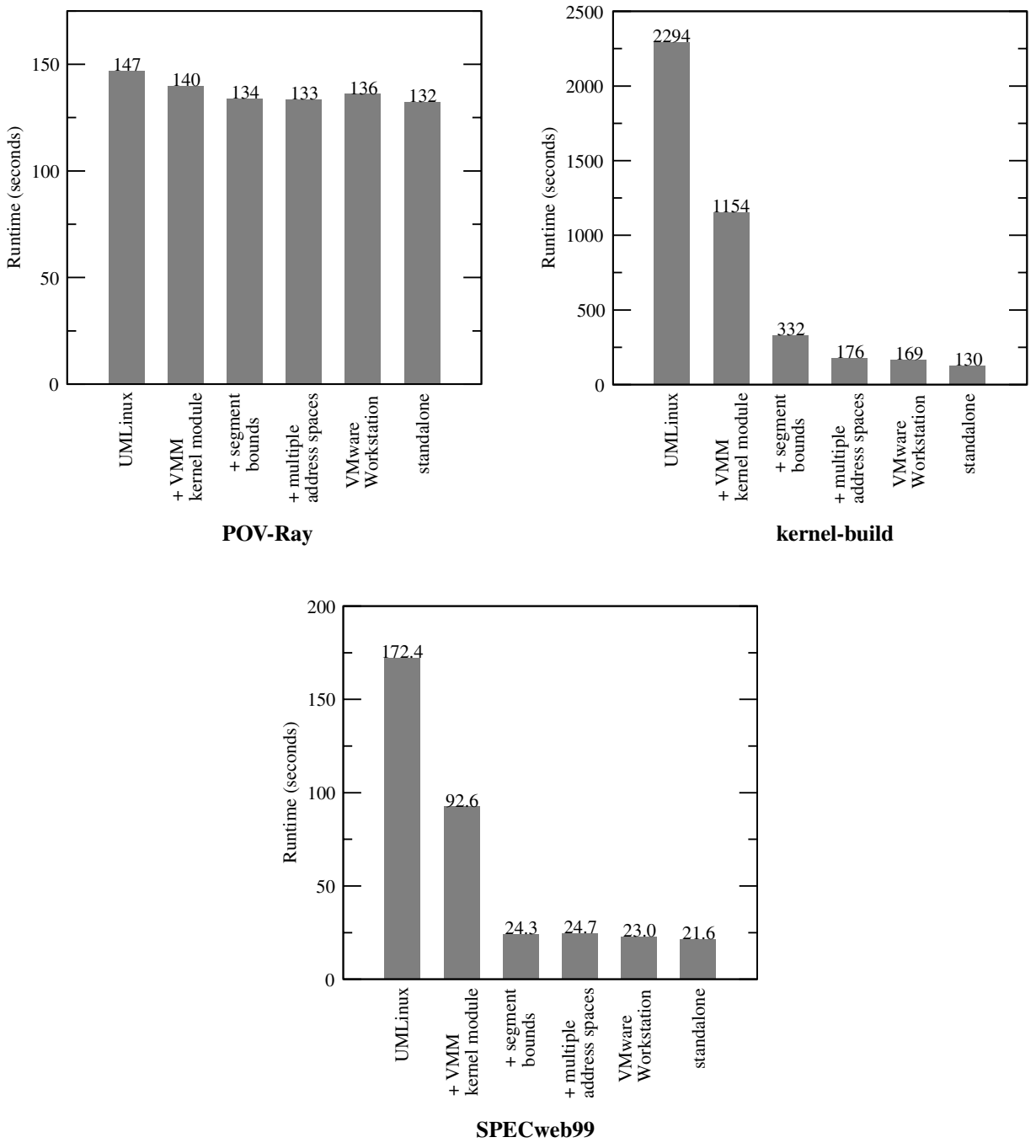


Figure 10: Macrobenchmark results. This figure compares the performance of different virtual-machine monitors on three macrobenchmarks: the POV-Ray ray tracer, compiling a kernel, and SPECweb99. The first four bars represent the performance of UMLinux with increasing support from the host OS. Each optimization level is **cumulative**, i.e. it includes all optimizations of the bars to the left. The performance of a standalone host (no VMM) is shown for reference. Without support from the host OS, UMLinux is much slower than a standalone host. Adding three extensions to the host OS allows UMLinux to approach the speed of a Type I VMM.

process. Implementation of this optimization is currently underway [Dike02b], though User-Mode Linux still uses two separate host processes, one for the guest kernel and one for all guest application processes. We currently use UMLinux for our CoVirt research project on virtual machines [Chen01] because running all guest code in a single host process is simpler, uses fewer host resources, and simplifies the implementation of our VMM-based replay service (ReVirt) [Dunlap02].

The SUNY Palladium project used a combination of page and segment protections on x86 processors to divide a single address space into separate protection domains [Chiueh99]. Our second solution for protecting the guest kernel space from guest application processes (Section 4.2) uses a similar combination of x86 features. However, the SUNY Palladium project is more complex because it needs to support a more general set of protection domains than UMLinux.

Reinhardt, et al. implemented extensions to the CM-5's operating system that enabled a single process to create and switch between multiple address spaces [Reinhardt93]. This capability was added to support the Wisconsin Wind Tunnel's parallel simulation of parallel computers.

7. Conclusions and future work

Virtual-machine monitors that are built on a host operating system are simple and elegant, but they are currently an order of magnitude slower than running outside a virtual machine, and much slower than VMMs that are built directly on the hardware. We examined the sources of overhead for a VMM that run on a host operating system.

We found that three bottlenecks are responsible for the bulk of the performance overhead. First, the host OS required a separate host user process to control the main guest-machine process, and this generated a large number of host context switches. We eliminated this bottleneck by moving the small amount of code that controlled the guest-machine process into the host kernel. Second, switching between guest kernel and guest user space generated a large number of memory protection operations on the host. We eliminated this bottleneck in two ways. One solution modified the host user segment bounds; the other solution modified the segment bounds and ran the guest-machine process in CPU privilege ring 1. Third, switching between two guest application processes generated a large number of memory mapping operations on the host. We eliminated this bottleneck by allowing a single host process to maintain several address space definitions. In total, 510 lines of code were added to the host kernel to support these three optimizations.

With all three optimizations, performance of a Type II VMM on macrobenchmarks improved to within 14-35% overhead relative to running on a standalone host (no VMM), even on benchmarks that exercised the VMM intensively. The main remaining source of overhead was the large number of guest application processes created in one benchmark (kernel-build) and accompanying page faults from demand mapping in the executable.

In the future, we plan to reduce the size of the host operating system used to support a VMM. Much of the code in the host OS can be eliminated, because the VMM uses only a small number of system calls and abstractions in the host OS. Reducing the code size of the host OS will help make Type II VMMs a fast and trusted base for future virtual-machine services.

8. Acknowledgments

We are grateful to the researchers at the University of Erlangen-Nürnberg for writing UMLinux and sharing it with us. In particular, Kerstin Buchacker and Volkmar Sieh helped us understand and use UMLinux. Our shepherd Ed Bugnion and the anonymous reviewers helped improve the quality of this paper. This research was supported in part by National Science Foundation grants CCR-0098229 and CCR-0219085 and by Intel Corporation. Samuel King was supported by a National Defense Science and Engineering Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. References

[Barnett02] Ryan C. Barnett. Monitoring VMware Honeypots, September 2002. http://honeypots.sourceforge.net/monitoring_vmware_honeypots.html.

[Boc] http://bochs.sourceforge.net/.

[Bressoud96] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[Buchacker01] Kerstin Buchacker and Volkmar Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the 2001 IEEE Symposium on High Assurance System*

Engineering (HASE), pages 95–105, October 2001.

[Bugnion97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[Chen01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[Chiueh99] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. In *Proceedings of the 1999 Symposium on Operating Systems Principles*, December 1999.

[Con01] The Technology of Virtual Machines. Technical report, Connectix Corp., September 2001.

[Dike00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.

[Dike02a] Jeff Dike. Making Linux Safe for Virtual Machines. In *Proceedings of the 2002 Ottawa Linux Symposium (OLS)*, June 2002.

[Dike02b] Jeff Dike. User-Mode Linux Diary, November 2002. http://user-mode-linux.sourceforge.net/diary.html.

[Dunlap02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.

[Goldberg73] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, February 1973.

[Goldberg74] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.

[Golub90] David Golub, Randall Dean, Allessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference*, 1990.

[Govil00] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):226–262, August 2000.

[Hoxer02] H. J. Hoxer, K. Buchacker, and V. Sieh. Implementing a User-Mode Linux with Minimal Changes from Original Kernel. In *Proceedings of the 2002 International Linux System Technology Conference*, pages 72–82, September 2002.

[Karger91] Paul A. Karger, Mary Ellen Zurko, Douglis W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11), November 1991.

[Magnusson95] Peter Magnusson and B. Werner. Efficient Memory Simulation in SimICS. In *Proceedings of the 1995 Annual Simulation Symposium*, pages 62–73, April 1995.

[McVoy96] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the Winter 1996 USENIX Conference*, January 1996.

[Meushaw00] Robert Meushaw and Donald Simard. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*, 9(4), September 2000.

[Nieh00] Jason Nieh and Ozgur Can Leonard. Examining VMware. *Dr. Dobb's Journal*, August 2000.

[Reinhardt93] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the 1993 Usenix Symposium on Microkernels and Other Kernel Architectures*, pages 73–89, September 1993.

[Robin00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 2000 USENIX Security Symposium*, August 2000.

[Rosenblum95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel &*

Distributed Technology: Systems & Applications, 3(4):34–43, January 1995.

[Sugerman01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June 2001.

[Waldspurger02] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[Whitaker02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.