Operating System Support for Virtual Machines

B Premkiran Raja[1] Dr.Somu Venkateswarlu[2]
1: Research scholar, OPJS University, RawatsarKunjla, Rajgarh, Churu, Rajasthan, India
2: Professor, Sreyas Institute of Engineering and Technology, Bandlaguda, Nagole, Hyderabad, Telangana, India

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 theVMM 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

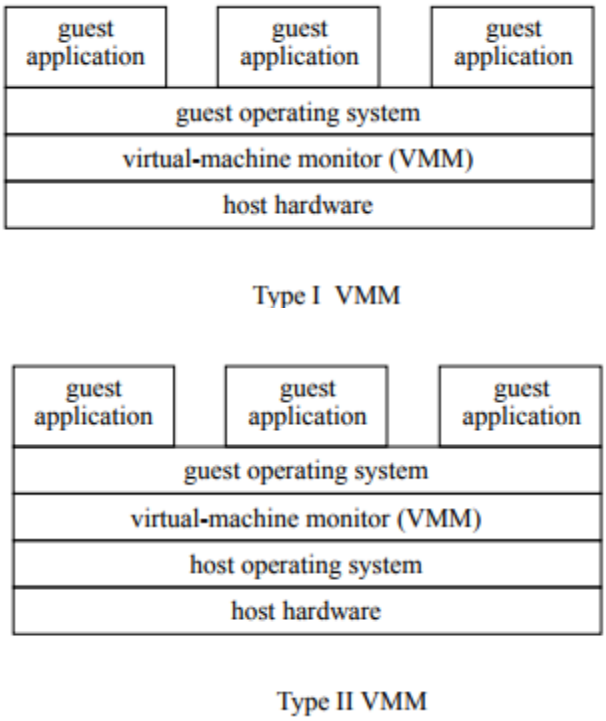


Figure 1: Virtual-machine structures. A virtual-machine monitor is a software layer that runs on a host platform and provides an abstraction of a complete computer

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 lowerlevel 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 VirtutechSimics [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 UserMode Linux [Dike00] and UMLinux [Buchacker01].

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.

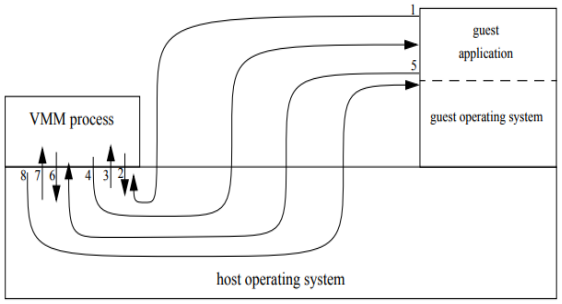
A second way to classify VMMs is according to the platform upon which they are built [Goldberg73]. Type I VMMs such as IBM's VM/370, Disco, and VMware's ESX Server are implemented directly on the physical hardware. Type II VMMs are built completely on top of a host operating system. SimOS, User-Mode 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 ConnectixVirtualPC [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. 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 guestmachine 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 cal

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.

3.1. Extra host context switches

The VMM process in UMLinux uses ptrace to intercept key events (system calls and signals) executed by the

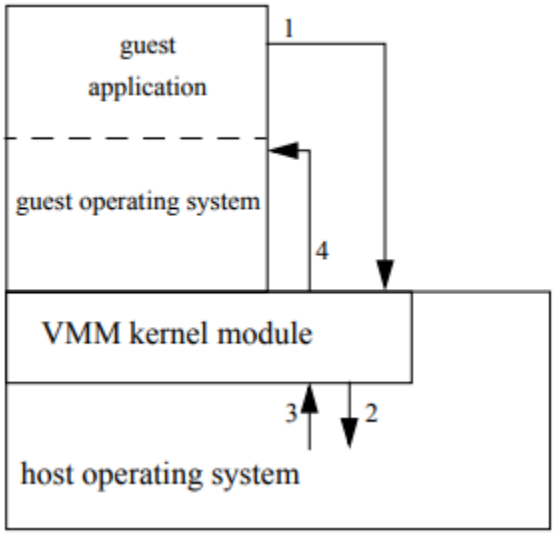
guest-machine process. ptrace is a powerful tool for debugging, but using it to create a virtual machine causes the host OS to context switch frequently between the guest-machine process and the VMM process (Figure 4).

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.

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)

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 supervisoronly 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



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.

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 guestmachine 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.

4. 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 systemWe 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

5.References

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

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

[3] 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.

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

[5] Larry McVoy and Carl Staelin.Lmbench: Portable toolsfor performance analysis. In Proceedings of the Winter 1996 USENIX Conference, January 1996

[6] 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.

[7] George W. Dunlap, Samuel T. King, SukruCinar, MurtazaBasrai, 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

[8] Steven K. Reinhardt, BabakFalsafi, 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