Operating System Extensions to Support Host Based Virtual Machines

Samuel T. King and Peter M. Chen

Department of Electrical Engineering and Computer Science
University of Michigan
[kingst,pmchen]@eecs.umich.edu

Abstract: This paper presents two new operating system extensions that facilitate the fast execution of host based virtual machines: KTrace and MMA. KTrace provides a convenient and efficient mechanism for writing kernel modules that control the execution of user-level processes. MMA exposes more detail of the underlying memory management hardware, providing applications with access to high performance intra-address space protection and address space overloading functionality. These extensions are applied to UMLinux, an x86 virtual machine that implements Linux as a user-mode process. As a result, overhead for UMLinux is reduced from 33% for a CPU bound workload and 819% - 1240% for system call and I/O intensive workloads, to 6% and 49% - 24% respectively.

1. Introduction

Originally developed by IBM in the 1960's, virtual machines were used to provide software backward compatibility on time-shared mainframe computers. This amount of portability allowed users to run complete operating systems, meant for old hardware, on new computers without modifications. Although this level of portability is useful, virtual machines also exhibit behavior that makes them popular today. Virtual machines provide software with the illusion that it possesses exclusive access to the computer hardware. Therefore, multiple operating systems can run on the same hardware in complete isolation. In addition to isolation, virtual machines can effectively multiplex hardware between several virtual machine instances in a fair way. Because of these two properties, virtual machines have seen a resurgence of research activity in recent years. Among the current research projects in the area are intrusion analysis [14] and prevention [22], fault injection [8], server farms [31], and secure mobile computing [23, 19].

My research group, CoVirt, is interested in implementing services below the virtual machine [11]. Because they are implemented below a virtual machine, secure services only need to trust the virtual machine monitor (VMM), not an entire operating system. This is beneficial because the VMM code size is significantly smaller than a full-blown operating system, making verification much easier.

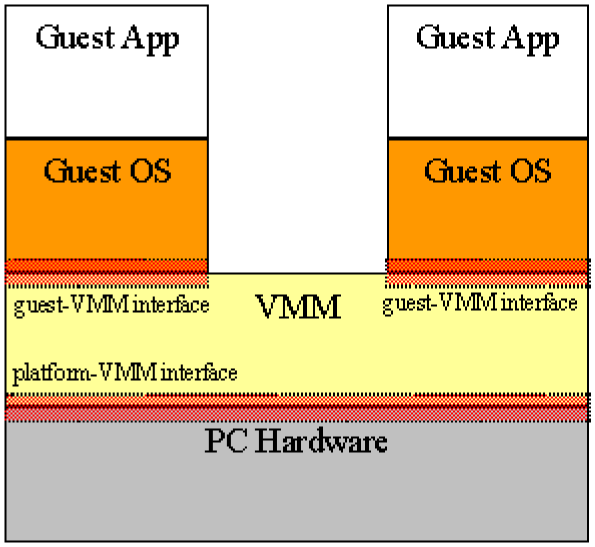

Figure 1: Direct-on-hardware virtual machine.

Although there are many different virtual machines available, running on various platforms, User-Mode Linux (UMLinux) running on x86 is the virtual machine CoVirt uses for security applications. Researchers at the University of Erlangen-Nurnberg developed UMLinux for fault injection experiments, but the extremely small VMM implementation provides a strong isolation barrier, making it an excellent choice for security applications. However, the original UMLinux implementation suffers from poor performance, making it unfeasible in a practical setting. The main reason for inefficiencies is because UMLinux sacrifices performance for VMM simplicity by using operating system abstractions, rather than accessing the hardware directly. Lack of efficient operating system

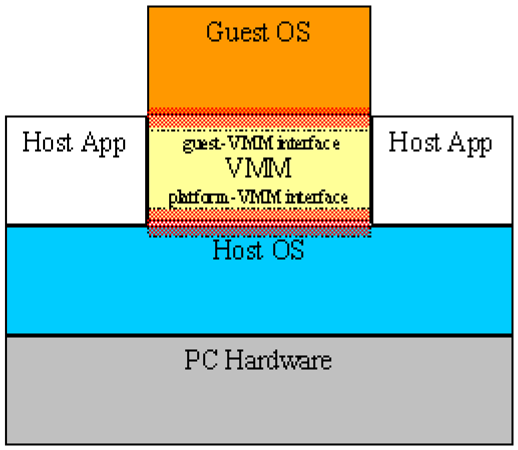

Figure 2: Host based virtual machine.

extensibility, and virtual memory abstractions that hide the power of the underling memory management haware are known to contribute to this performance degradation. In an effort to improve performance, I developed two new abstractions: KTrace and memory management abstractions (MMA). KTrace is a mechanism through which kernel modules can register callback functions that are triggered by exceptions generated from a specified process. MMA unleashes the power of the x86 memory management hardware by providing access to page table manipulation functions, and hardware protection mechanisms. Together, KTrace and MMA can be used to implement simple VMMs, without sacrificing performance.

The remainder of this paper is organized as follows: Section 2 presents background information on virtual machines and discusses the design tradeoffs of three modern virtual machines. Section 3 details the design of KTrace and MMA, and Section 4 present experimental results showing the improvements made by using KTrace and MMA on UMLinux. Section 5 discusses related work and section 6 concludes.

2. Virtual machines

As with any abstraction, the VMM is responsible for managing the hardware and other resources below while providing an interface for virtual machine instances above. The interface below the VMM, called the platform-VMM interface, could include bare hardware, an operating system, or any combination of the two. Virtual machines that run on bare hardware are called direct-on-hardware virtual machines (Figure 1),

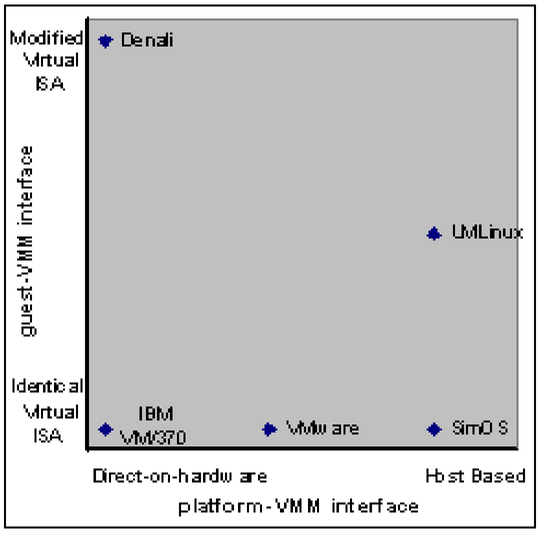

Figure 3: The IBM VM/370 [3] is a classic virtual machine in that it implements a direct-on-hardware virtual machine and exports a virtual instruction set that is identical to the underlying hardware. Like the IBM VM/370, Denali [31] implements a direct-on-hardware virtual machine. However, Denali adds to and removes instructions from the virtual instruction set, making the existing code base written for the underlying architecture unusable. VMware [1] employs a hybrid approach by using a host OS for some services and manipulating hardware directly for others. SimOS [26] implements a complete guest-VMM interface, without accessing hardware directly, running above the host OS. UMLinux [8] runs on top of the host OS, but deprecates certain instructions. These instructions are usually found in operating systems, so operating systems require a UMLinux port, but user-mode applications remain binary compatible.

whereas virtual machines that rely on an operating system for services are called host-based virtual machines (Figure 2). Although direct-on-hardware virtual machines have more direct control of the execution environment, host based virtual machines can be simpler to implement by using well-known operating system abstractions. In the host based approach to virtualization, the underlying operating system is referred to as the host OS, whereas any operating system running on virtual hardware is called a guest OS.

Like the platform-VMM interface, the interface above the VMM can vary widely. This interface, called the guest-VMM interface, is what virtual machines run on. Some virtual machines implement a guest-VMM interface that is identical to the underlying hardware thus, allowing any code written for that particular hardware to be run in the virtual machine. Other virtual machines add and subtract aspects of the underlying hardware. This modification of the instruction set architecture (ISA) is done for various reasons ranging from performance, to simplicity of VMM implementation. To show how different virtual machines define these interfaces, several examples are given in Figure 3.

Virtual machine design is a tradeoff between generality, complexity, and performance. To illustrate this tradeoff, three modern virtual machines are discussed in detail: VMware, Denali, and UMLinux.

2.1. VMware

VMware is the most prevalent x86 virtual machine today. By providing a complete implementation of the x86 ISA at the guest-VMM interface, VMware is a useful tool with many different applications. Although this complete guest-VMM interface is extremely general, it leads to a more complicated VMM. Because there are multiple operating systems running on the same hardware, the VMM must emulate certain instructions to correctly represent a virtual processor to each virtual machine; instructions that must be emulated are called sensitive instructions. For performance reasons, virtual machines typically rely on the processor trap mechanism to run the VMM when a sensitive instruction is executed. However, x86 processors do not trap on all sensitive instructions [25] so additional work must be done. To handle sensitive instructions that do not cause a trap, VMware uses a technique called binary rewriting. In binary rewriting, all instructions are examined before being executed, and the VMM inserts a breakpoint in place of sensitive instructions. The breakpoint causes a trap to the VMM when executed, where the instruction is emulated. Binary rewriting adds to VMM complexity in VMware, but provides a complete x86 virtual instruction set at the guest-VMM interface.

As with the guest-VMM interface, VMware balances many different factors in the design of the platform-VMM interface. For performance reasons, the VMware VMM uses a hybrid approach to implement the platform-VMM interface. Exception handling and memory management are accomplished by manipulating hardware directly, but in an effort to simplify the VMM, I/O is supported through a host OS. By using well-known abstractions to support I/O, the VMM avoids maintaining device drivers, something existing operating systems already do well. This simplicity caused poor performance in old versions of VMware, but optimizations are used to mitigate the effects and speed up I/O [28].

In addition to VMM optimizations to improve I/O performance, several guest device drivers specific to VMware are used. By using VMware device drivers that know about running on virtual hardware, I/O performance approaches that of an operating system running on bare hardware. VMware specific guest device drivers can be viewed as a modification to the guest OS, and although VMware will work without it, the performance is greatly enhanced using these modifications.

Memory management in VMware is carried out by the guest OS directly on the hardware. To insure that there are no conflicts between host and guest memory, VMware allocates and pins host RAM for use in the guest context. The guest then uses the allocated RAM. One aspect that complicates this is the VMM must reside in all guest contexts so it can gain control when needed, causing a small section of the guest linear address space to be emulated on each guest read and write.

To gain control of the guest, VMware implements interrupt service routines for all interrupts. Whenever guest code causes an exception, it is first examined by the VMM. Interrupts that correspond to I/O devices are forwarded to the host so that they can be handled properly, while exceptions generated by guest applications (like system calls) are forwarded back to the guest.

Overall, VMware has attained an excellent balance between all virtual machine design factors. The guest-VMM interface is true to x86 hardware, and using the host OS for I/O simplifies the VMM design. Optimizations to alleviate overhead have been implemented, but the VMM does still carry significant complexity because binary rewriting is used, and many aspects of x86 hardware must be emulated. For VMware, trading VMM complexity for guest-VMM generality and performance is a necessary compromise.

2.2. Denali

The main design goals for Denali are to reduce virtualization overhead (overhead due to running in a virtual machine), provide strong isolation between different virtual machine instances, and simplify the VMM. These goals are accomplished using a technique called para-virtualization. Para-virtualization

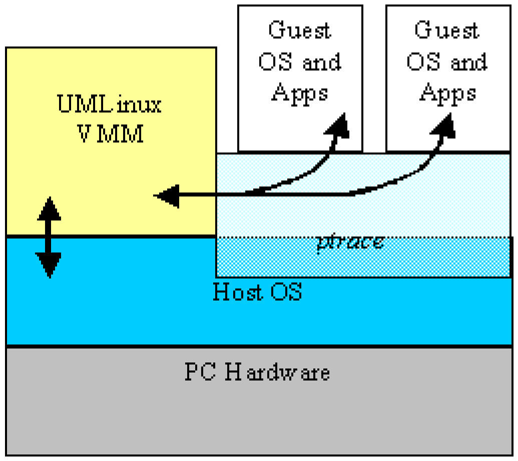

Figure 4: UMLinux architecture.

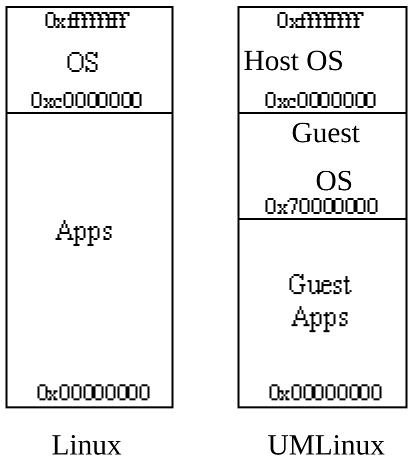

Figure 5: Address space of Linux and UMLinux.

involves modifying the guest-VMM interface; instructions are added to and subtracted from the virtual architecture. By deprecating certain x86 instructions, binary rewriting is no longer needed. Because x86 hardware is complicated and requires a great deal of emulation for backward compatibility, removing instructions significantly simplifies the VMM. As part of the Denali virtual architecture, virtual I/O is supported at the ISA level. For example, an entire Ethernet frame can be sent to the VMM using a single instruction. This significantly reduces the number of guest to VMM traps, and simplifies the design of guest OSs.

Denali runs directly on top of the hardware, there is no host operating system. As a consequence, the Denali VMM must provide device drivers for all hardware running on the specific platform. By implementing drivers in the VMM, Denali can enforce policies for fair hardware multiplexing. This insures isolation between different virtual machine instances, but complicates the VMM.

There is no concept of virtual memory hardware in Denali virtual machines; all guest code runs in a single, private, address space. Although this simplifies the guest OS, the lack of intra-virtual machine protection limits the capability of the guest OS. It requires a complete change of design in the way applications are built. Essentially, any standard "process" would have to be run in a new virtual machine if protection is needed.

Interrupt handling is also different in Denali. Rather than handling interrupts when they occur, they are queued until the specific virtual machine is run. Because this reduces the number of virtual machine to VMM switches needed to operate, the virtualization overhead is less.

To support a high performance and simplified guest OS and VMM design, Denali sacrifices guest-VMM generality to the point of loosing x86 compatibility, even with user-mode applications. The virtualization overhead is significantly low such that many different virtual machines can run concurrently, which makes Denali interesting and a possible glimpse into the future of computing. However, the total loss of generality makes Denali limited in a practical setting because of the extensive amount of software already in use for the x86 platform.

2.3. UMLinux

UMLinux makes extensive use of host OS abstractions to implement guest services. The UMLinux guest OS is implemented by modifying the architecture specific sections of the Linux operating system. Because the guest OS is modified, the need for binary rewriting is removed. This simplifies the VMM significantly, but results in a loss of generality. Fortunately, user-mode applications are still binary compatible, insuring that the diverse, extensive set of applications already written for Linux can still be used.

The VMM is a single host process that controls all virtual machines. There is one host process for each virtual machine instance, and a single host process for the UMLinux VMM. Using ptrace, the VMM gains control of virtual machines at the entrance and exit of any guest system call, and before any signals are delivered (Figure 4). Ptrace is also used to manipulate the guest context.

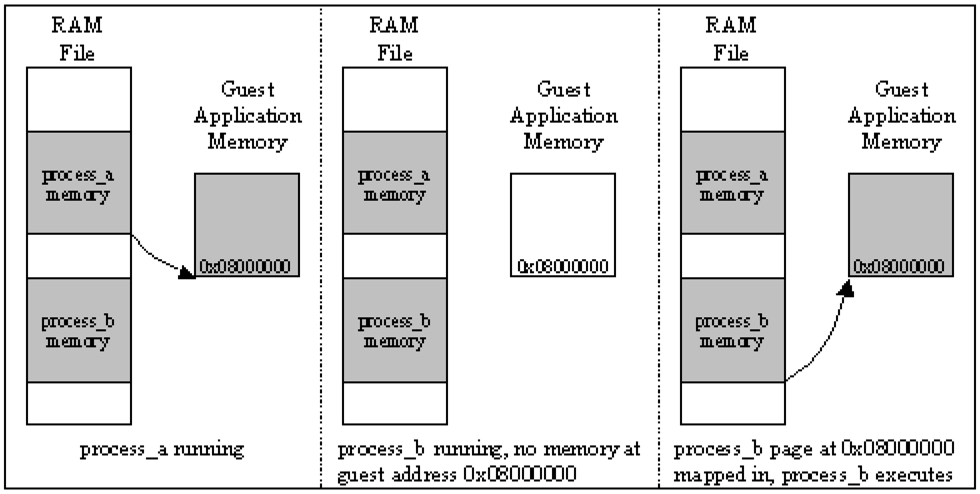
RAM File
process_a memory
process_b memory
Guest Application Memory
0x08000000
process_a running
process_b running, no memory at guest address 0x08000000
process_b page at 0x08000000 mapped in, process_b executes

Figure 6: UMLinux guest context switching.

Like VMware, UMLinux uses the host OS for I/O. Because a UMLinux virtual machine is a normal host process, switching between different virtual machine instances is as fast as switching between normal host processes. Also, modifications to guest device drivers reduce the need for context switches, an optimization also used by VMware.

Because each virtual machine is a host process, the address space must be modified slightly (Figure 5). In a standard Linux application, the host OS occupies addresses [0xc0000000, 0xffffffff] while the application is given [0x00000000, 0xc0000000). Because a UMLinux virtual machine must hold the host OS, a guest OS, and guest applications, the address space for the host OS remains in the same place, but the guest OS occupies [0x70000000, 0xc0000000) leaving [0x00000000, 0x70000000) for guest applications. The guest kernel memory is protected by the VMM using host `mmap` and `munmap` system calls. To facilitate this protection, the VMM maintains a virtual current privilege level (VCPL), which is analogous to the x86 current privilege level (CPL). This is used to differentiate between user and kernel modes, and the VMM will map or un-map kernel memory depending on the VCPL.

In addition to using `mmap` and `munmap` for kernel address space protection, UMLinux also uses them for overloading the guest application address space and performing guest context switches. To allocate memory for a process, the guest kernel uses host `mmap` system calls on a RAM file. This RAM file is created by writing a file large enough to hold the guest RAM into a temporary directory, mapping it, and pinning it. When a context switch is called for, the entire guest application address space is un-mapped and the new process is demand paged back in. For example, if process_a is running and occupying virtual address 0x08000000, that address would correspond to a specific offset into the RAM file (Figure 6). To switch to process_b, also running at virtual address 0x08000000, the guest kernel first makes a host `munmap` system call, removing all data in the address space between virtual addresses [0x00000000, 0x70000000). When process_b runs, it generates a host segmentation fault because there is no data in the guest application address range. The resulting host segmentation fault is treated as a page fault by the guest kernel. Using the host `mmap` system call, the guest kernel associates virtual address 0x08000000 with a different offset into the RAM file, thus changing the address space of the guest application and allowing process_b to run.

Guest exception handling is implemented using UNIX signals. The guest OS registers handlers for all signals, and treats them similarly to their hardware equivalents. Because signals are used, the VMM does not have to implement hardware interrupt service routines, simplifying the VMM.

Although UMLinux does sacrifice some degree of generality by requiring guest OS modifications, guest applications are still binary compatible. Changing the

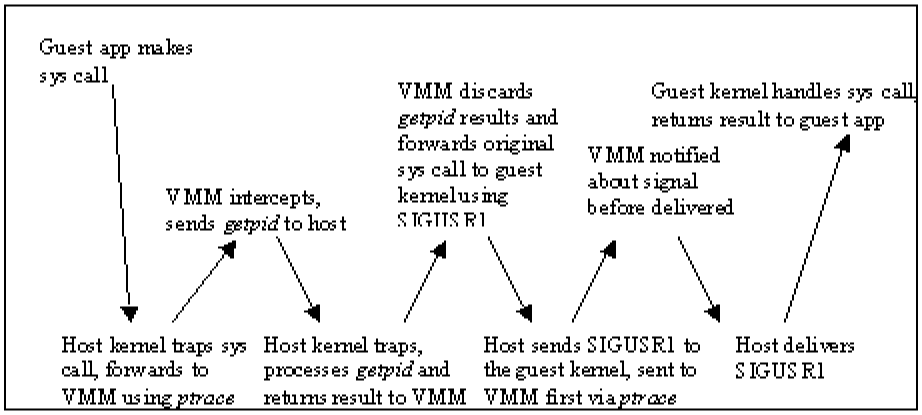
Guest app makes sys call
VMM intercepts, sends getpid to host
VMM discards getpid results and forwards original sys call to guest kernel using SIGUSR1
VMM notified about signal before delivered
Guest kernel handles sys call, returns result to guest app
Host kernel traps sys call, forwards to VMM using ptrace
Host kernel traps, processes getpid and returns result to VMM
Host sends SIGUSR1 to the guest kernel, sent to VMM first via ptrace
Host delivers SIGUSR1

Figure 7: Call trace of a guest application system call in UMLinux.

guest OS is a technique also used by VMware, and results in better performance. Implementing the memory management and interrupt handling using host OS abstractions results in a simple VMM, but suffers performance degradation when compared with hardware equivalents.

3. Design

There are advantages and disadvantages to each of the different virtual machines discussed above. Implementation of a virtual machine on top of the host, without direct access to the underlying hardware, leads to a simple VMM implementation, but suffers performance degradation. Specifically, there is not an efficient way to trace processes without paying the high performance penalty of using `ptrace`. Also, current memory management abstractions hide the true power of the underlying x86 hardware, creating a significant amount of extra work for protection and address space overloading. To mitigate these effects, and still offer the simplicity of implementing a virtual machine completely above the host OS, KTrace and MMA are used.

3.1. KTrace

`Ptrace` is a versatile tool with applications ranging from binary instrumentation to debugging. Although it does provide a convenient mechanism for implementing operating system extensibility, applications suffer a significant performance penalty when using `ptrace`. One major cause for overhead is the number of cross process context switches that are needed to implement services. In Linux running on x86 processors, traps are different than context switches. Because the Linux kernel is mapped into each process, traps are used only to change protection domains. As a result, traps to the Linux kernel are inexpensive. On the other hand, context switches require flushing the translation lookaside buffer (TLB), making them less efficient than a trap. Because context switches are more expensive than traps, applications using `ptrace` to implement services will suffer from poor performance. For example, UMLinux uses `ptrace` to redirect system calls from guest applications to the guest kernel.

When a guest application makes a system call, the UMLinux VMM intercepts this system call, using `ptrace`, before it gets executed by the host OS (Figure 7). Once the system call has been intercepted, it is redirected back up to the guest so that the guest kernel can handle it. Because the host OS thinks that a user mode process made a system call, the VMM must still send a system call down. Rather than sending the system call that was meant for the guest kernel (it would not make any sense to the host since it does not even know there is a guest OS), it sends a `getpid` system call, which has no effect on either kernel. After the host kernel finishes the getpid call, the VMM gains control again and restores the context to reflect the guest system call and invokes the guest kernel system call handler. This invocation is implemented via the host kernel signal mechanism; the UMLinux VMM sends a signal (SIGUSER1) to the guest kernel, using `ptrace`, signifying the arrival of a guest system call.

Implementing a guest system call requires many traps and context switches, whereas a normal host system call only requires a trap to the host kernel. Moving the VMM logic to the host kernel reduces the number of context switches needed to carry out a guest system call significantly. Porting code to run in the kernel is a

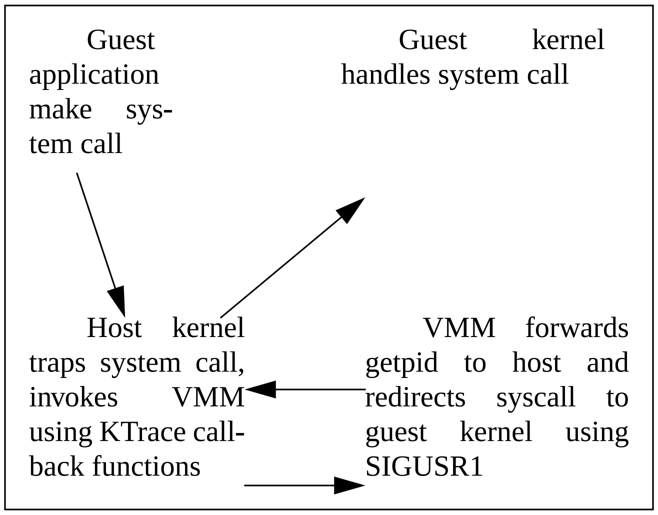
Guest application make system call
Guest kernel handles system call
Host kernel traps system call, invokes VMM using KTrace callback functions
VMM forwards getpid to host and redirects syscall to guest kernel using SIGUSR1

Figure 8: UMLinux guest system call implemented using KTrace

well-known optimization, but using KTrace makes the processes easy for applications that require `ptrace` functionality. KTrace allows loadable kernel modules (LKMs) and other kernel code the convenience of `ptrace`, without the overhead, by facilitating event notification through callback functions.

Event notification is implemented using callback functions. The callback functions are passed a pointer to a `pt_regs struct` that represents the context of calling processes. Any changes made to this structure are reflected in the process when execution resumes. System calls, signals, and breakpoints each generate events that kernel code can register for notification on. System call events are fired at the entrance and exit of all system calls, signal events are triggered before a signal is delivered to the process, and breakpoint callback functions are invoked after the breakpoint has occurred.

In addition to event notification, using KTrace allows for efficient access to the guest address space. Using `ptrace`, a system call is required for each four byte read or write of the processes address space. Because KTrace code resides in the host kernel, the processes address space can be accessed directly.

As an example of how KTrace can improve efficiency, a UMLinux system call can be implemented with KTrace, instead of ptrace (Figure 8). When a guest application system call traps to the host kernel, the UMLinux VMM kernel module is invoked through the event notification mechanism. Essentially, this notification is a procedure call rather than a full context switch between two processes. Like in the original VMM, the registers are manipulated so that the host kernel believes a `getpid` system call was made. Once the `getpid` is executed, the VMM is sent a system call return event notification. Then, the VMM manipulates the guest context to reflect the original system call, and a signal is sent to the guest kernel. As a result of the context switch reduction, implementing a UMLinux guest system call using KTrace significantly reduces overhead.

Using KTrace is much simpler than implementing equivalent functionality in a kernel module. To accomplish the system call and breakpoint notification, the appropriate entries in the interrupt vector would be overwritten, and the new service routines would call the originals after doing some processing. Although this is straight forward, it involves detailed low-level knowledge of x86 processors. However, implementing notification before signals are delivered is more involved. To implement the signal notification, kernel functions would have to be overwritten in memory. Overwriting code in a running kernel would work, but is not a good solution.

3.2. MMA

Memory management hardware for x86 processors is diverse and flexible. Support for both segmentation and paging is provided and used in different degrees by various operating systems. For example, Linux uses only the necessary segmentation hardware, relying mostly on paging hardware for virtual memory implementation and protection. Although this makes sense because Linux supports several different architectures, some of which do not have segmentation, applications may have use for the segmentation hardware. Furthermore, Linux only utilizes two of the possible four protection rings in x86 processors. Protection rings are used as a way to protect the kernel section of the address space, and privileged instructions, from applications. Most modern operating systems only use rings 0 (kernel mode) and 3 (user mode), but the intermediate rings are useful, especially to virtual machines.

The cause of poor address space performance in UMLinux is due to the available abstractions. For example to protect the guest kernel, the UMLinux VMM must `munmap` the guest kernel memory region when the virtual machine is running in user-mode. `Mmap` and `munmap` are expensive operations when used on the guest kernel memory, and often result in writes to disk. Unfortunately, using mmap and munmap is necessary; otherwise applications could access guest kernel memory and easily break into the system. In addition to guest kernel protection, guest context switching is slow because the guest application address space is overloaded using `mmap` and `munmap`. In contrast, a host kernel only needs to change the page table directory register to switch the entire address space. To combat these inefficiencies, several new memory management functions are added to the kernel, called MMA.

3.2.1. Segmentation Limits and Intermediate Privilege Rings

Linux makes minimal use of x86 segmentation hardware, utilizing it as a necessary mechanism for switching between privilege levels. There are four different segments used in Linux: a kernel code segment, a kernel data segment, a user code segment, and a user data segment. For all segments, the base is equal to 0 and the bound spans the entire address space. All host processes share the two user segments and the kernel has exclusive access to the two kernel segments. Because kernel-mode and user-mode code use different segments, changes to the user-mode segment base and bounds do not affect the kernel. Therefore changing the user-mode segment bounds can be used as a mechanism for implementing hardware protection in user-mode applications. To facilitate this, MMA implements a dynamic, per process, segmentation bound parameter. The `mmaSetBound` function sets the bound for the given process. Adding the segmentation bound to the process context allows each process to maintain different segmentation bounds. Because the primary use for this is memory protection, only host kernel code can set the bound for a process. For example, MMA segmentation bounds are used to protect the guest kernel in UMLinux. By setting the bound to 0x70000000 when the virtual machine is in virtual user mode, any attempted access of the guest kernel memory by a guest application correctly results in a fault. Before the guest kernel runs, the segmentation bound is set to span the entire address space once again. However for this protection mechanism to be secure, the code that performs the bounds modification cannot be available to guest code. The guest kernel cannot make the change because the segmentation hardware prevents it from running, so the change is made by the VMM. Using KTrace, the VMM running in the host kernel tracks all transitions between virtual user and kernel-modes. At these transitions, the segmentation limit is changed for a UMLinux virtual machine by the VMM.

In addition to modifying segmentation bounds, MMA implements the ability for user-mode applications to run in ring 1. Running in ring 1 is an additional way to use hardware for user-mode address space protection. Code running in ring 1 can access protected memory, but cannot execute privileged instructions. Because of this freedom, the segmentation hardware must also be used to protect the host kernel from code running in ring 1. And like `mmaSetBound`, the ring 1 transition code must be executed outside of the application. Changing execution rings is done using the `mmaChangeRing` function. Since `mmaChangeRing` is called from the host kernel, the process runs in ring 1 once execution is resumed. To facilitate protection, the `mmaSetPageProtections` function is used. A page that is protected cannot be accessed by code running in ring 3, and the segmentation hardware protects the host kernel from the ring 1 code.

Like setting segmentation limits, parts of the user-mode address space can be protected by running certain code in ring 1. Setting segmentation bounds can only be used to protect a contiguous range of the upper part of the user-mode address space, while setting page protections can be used for any page in the user-mode address range. Both are efficient, but the segmentation bound solution is simpler than allowing code to run in ring 1, while changing rings provides greater flexibility.

3.2.2. Address Space Overloading

MMA provides address space overloading functions, allowing a process to maintain several different copies of an address space range. Rather than using `mmap` and `munmap` to maintain a range of address space, MMA modifies the first level hardware page table and associated kernel data structures. Applications can then maintain several different copies of the same range of address space, and switch them using a system call. Because the page table is manipulated directly, and there is no backing file to deal with, changing an address space range is fast.

Unlike MMA protection mechanisms, the address space overloading functionality is available to user-mode applications; care must be taken to guard against attacks. First, because the actual data is stored in the host kernel, each process is limited to 1024 different address spaces, thus preventing malicious code from allocating all free kernel memory. Also, each process

Workload	Original UMLinux	UMLinux + KTrace	UMLinux + KTrace + MMA segment bounds protection	UMLinux + KTrace + MMA segment bounds protection + fast context switching	VMware
POV-Ray	33.9%	16.7%	5.5%	5.6%	3.7%
kernel build	1240%	627.3%	102.5%	48.9%	20.3%
SPECweb 99	819.2%	392.8%	26.8%	23.8%	3.0%

Table 1: Virtualization overhead for the macro benchmarks. Results shown are relative to stand-alone Linux and calculated by dividing the different between the virtual machine time divided by the time for stand alone Linux.

has separate storage (i.e. one process cannot swap in the address space of another process). Finally, the address range is referenced by the application using an index, so out of range or invalid entries can easily be handled.

One example of using address space overloading is for UMLinux context switching. The guest kernel maintains different copies of the address range [0x00000000, 0x70000000) for each guest application. When a new process is scheduled, rather than calling `munmap` and demand paging in the new address space, an invokation of the `switchguest` system call updates the host process address space and the new guest process is run.

4. Evaluation

To illustrate the capability of KTrace and MMA, they are implemented in a 2.4.18 Linux kernel, and UMLinux is modified to utilize them. The UMLinux VMM is implemented as a loadable kernel module on the host, which uses KTrace to trace each virtual machine instance. Also, MMA is used by the VMM to implement guest kernel protection. Finally, the guest kernel is updated to use the MMA system calls for context switching. Applying these optimizations results in a significant performance enhancement for UMLinux. To quantify the performance boost, several macro and micro benchmarks are used.

All test are run on several different configurations to illustrate the affect of various performance enhancements. The systems are an unmodified version of UMLinux, UMLinux using KTrace, UMLinux using KTrace and MMA for guest kernel memory protection, UMLinux using KTrace and MMA for guest kernel memory protection and fast guest context switching, VMware, and stand-alone Linux.

The experiments are run on a PC compatible computer with a 1.5 GHz AMD Athlon processor, 512 MB of memory, and a Samsung SV4084 IDE disk. All tests are run three times and the resulting variance is less than 3% in all cases.

For UMLinux and VMware, the guest OS is Linux 2.2.20 with a Linux 2.4.18 host; the stand-alone kernel is Linux 2.2.20. Virtual machines are allocated only 164 MB of RAM because of limitations in UMLinux, but this should have minimal affect on performance testing. Also, raw disk partitions are used for the two virtual machines to avoid both file cache double buffering and excessive speedup due to host file caching.

4.1. Macro Benchmarks

Several macro benchmarks are used to test how well the UMLinux system performs compared to stand-alone Linux and VMware (Table 1). Stand-alone Linux is considered the upper limit for how well a virtual machine could perform, and VMware is an example of a virtual machine that attains excellent performance. Three different macro benchmarks are used to test the system: a kernel build, SPECweb99, and POV-Ray. The kernel build is a compilation of the 2.4.18 Linux kernel, using default configuration values. SPECweb99 is run with 15 simultaneous connections made from two different client computers. The server is given 300 seconds of warm up time before the test commenced; each test runs for 300 seconds. The virtual machine is using the 2.0.36 apache web server and all computers are connected using a 100 Mb Ethernet

Workload	Original UMLinux	UMLinux + KTrace	UMLinux + KTrace + MMA segment bounds protection	UMLinux + KTrace + MMA segment bounds protection + fast context switching
Null system call	71.0 µs	37.3 µs	3.3 µs	3.3 µs
Context Switching	23137 µs	12412 µs	3764 µs	1745 µs

Table 2: Micro-benchmark result. The times listed are the times to complete a single system call, or a single context switch.

switch. POV-Ray is a ray tracing application and the benchmark image is rendered using a quality factor of eight. Each benchmark tests many different aspects of the system, and are similar to those used to evaluate Cellular Disco [20].

After applying all optimizations, UMLinux attains excellent performance compared to stand-alone Linux and VMware. This is due to the significant speedups gained from implementing UMLinux using KTrace and MMA. Before these were used, UMLinux had 1240% overhead for a kernel compile compared to stand-alone Linux and anly 48.9% overhead when all optimizations are applied. Clearly, these optimizations make UMLinux feasible as an everyday system. Furthermore, there is no noticeable performance degradation on less demanding tasks like web surfing or editing a text file.

4.2. Micro Benchmarks

To understand the effects of the different optimizations, and to help recognize areas in need of improvement, two micro benchmarks are used: a null system call latency test, and a context switching latency test (Table 2). The null system call test measures the amount of time a null (`getpid`) system call takes to execute, and the context switching latency test measures the overhead of a context switch. Both tests are from the LMbench toolkit [7].

Modifying the UMLinux VMM to use KTrace, instead of running as a separate process, results in a 1155 second reduction in the amount of time needed to compile a 2.4.18 Linux kernel. The majority of this speedup is due to faster guest context switching. Because the Linux kernel compilation configuration uses the pipe option by default, and pipe buffers are only 4096 bytes, there are a number of context switches needed to compile a kernel. Although there are many context switches, it is difficult to quantify the overhead of a UMLinux guest context switch. UMLinux context switch overhead is dependant on the working set of the new process. When a new process is run, each page used is demand paged in because of the `mmap` and `munmap` mechanisms used to overload the guest application address space. Therefore, processes with large working sets will take longer to context switch in. Because of this variability, one cannot simply measure the overhead reduction of a context switch and multiply that by the number of context switches needed in a kernel compile. Rather, the UMLinux system must be examined in more detail before the micro benchmark results can be used.

For a kernel compile, there are 22 million host segmentation fault signals sent to the guest kernel. These signals are used as notification of a guest page fault and the mechanism for demand paging in the address space of guest applications. To estimate the overhead reduction per segmentation fault, the results from the context switching tests are used. The context switching tests use two processes, both with 1 MB of data, which are connected by pipes. These pipes are used to force context switches. Before one process relinquishes control to the other, each of the 256 pages of data are read. Each read results in a segmentation fault, therefore most of the context switching speedup between the original UMLinux and UMLinux using KTrace can be attributed to optimizing the demand paging process.

A single guest context switch is 10725 µs faster when the UMLinux VMM is implemented using KTrace compared to the original UMLinux. For each segmentation fault, there is a 10725 µs/ 256, or 42 µs, speedup. As a result, a kernel compile takes approximately 42 µs * 22,000,000, or 924 s, less to complete. Although this is not account for the entire 1155 s speedup, it certainly makes up the majority. In addition, there are 1.4 million guest system calls. Each executes 34 µs faster, accounting for an additional 48 seconds.

Using MMA hardware protection mechanisms to protect the guest kernel further reduced kernel compilation time. This optimization results in a 1160 second speedup and can be accounted for by faster demand paging and guest system calls. Using the same calculations as above, about 790 seconds can be attributed to faster demand paging and guest system calls.

The final optimization reduces the kernel compilation time by another 110 seconds. However, the speedup is a result of removing several signals rather than speeding them up. By using MMA address space overloading functions, the number of segmentation faults generated during a kernel compile is reduced from 22 million to about 4.9 million. It is reasonable to assume that the speedup can be attributed to reducing the number of segmentation faults generated.

Based on the micro benchmarks, improving guest system call and context switch performance further will have little affect on the overall performance of UMLinux. A fully optimized guest system call incurs only 3 µs of additional overhead, and the guest context switching time is within 23 % of stand-alone Linux. One possible area for additional performance improvements may be the disk I/O subsystem. Currently, all reads and writes to the virtual disk block the entire virtual machine. Implementing asynchronous I/O could help reduce the virtualization overhead even further. However, the overall performance of UMLinux is reasonable, making it a viable option.

5. Related Work

In this section, I compare KTrace and MMA with similar projects: virtual machines, virtualization techniques, tracing mechanisms, intra-address space protection, and fast context switching.

5.1. Virtual Machines

In addition to the virtual machines already discussed in this paper, there is another project that is similar to UMLinux: User Mode Linux (unfortunately the same name). Like UMLinux, User Mode Linux [13] uses Linux user mode processes to implement a virtual machine. Modifications to the guest kernel are required and a host OS is used to implement guest OS services. However, User Mode Linux uses one host process for each guest process, while UMLinux encapsulates the entire virtual machine within a single host process. By using a host process for each guest process, User mode linux context switches between guest processes are as fast as non-virtual Linux. Fortunately, KTrace and MMA allow UMLinux context switching times to approach non-virtual Linux while still maintaining the simplicity of using only a single host process per virtual machine instance.

5.2. Additional Virtualization Techniques

There are several different approaches to virtualization that could be used in addition to virtual machines. Exokernel [15] uses a small kernel that is only responsible for maintaining protection. The interfaces exported expose details of the underlying hardware, and library operating systems are responsible for managing allocated hardware resources. Although there are library operating systems implemented, Exokernel applications in need of low-level hardware access can have it without modifying the kernel.

Mach [18, 17] is a microkernel operating system that typically implements a user mode BSD library OS. Mach separates different operating system components into servers and clients, each with a clearly defined API. This level of modularity allows modification of operating system components without unknowingly affecting other aspects of the kernel, so long as the defined interface is still followed. Although this makes for easier software engineering, the modular design may result in poor performance.

Flux OSKit [24] is a way to construct different operating system components without having to re-implement them from scratch. Among the providied components are basic kernel support, memory management, PThreads, networking, file system support, and a generic device driver framework that supports Linux and BSD drivers. In additions to OS layers, there are also X Display, a C run-time library, and a math library included. These components are pieced together so users can pick and choose the services needed for the specific application.

5.3. Tracing / Extensibility

There are a number of projects related to controlling the execution of applications. `Ptrace` is available on many different operating systems and is used in a wide range of applications. It can intercept system calls and signals, as well as modify the address space of the process that is being traced. Although ptrace does offer a great deal of flexibility, applications may not want to intercept every system call. To accommodate this type of application, the `/proc` file system on many different operating systems (Solaris, OSF/1) is used in a similar manner. Tracing applications can register for notification only on specific system calls and manipulate the traced processes address space directly.

In addition to using operating system tracing mechanisms, applications can be extended by modifying the binary code. These modifications allow for existing programs to be enhanced without access to the original source code. Three different binary instrumentation schemes are ATOM [16], Dyninst [9], and Dynamo [4]. There are many more such systems, but these represent three classes of binary rewriting toolkits. ATOM statically rewrites the binary by modifying executable files before they are run. Dyninst uses a slightly different approach by modifying the C run-time libraries to facilitate code insertion when C run-time functions are invoked. Dyanmo controls the application execution path as it is running and rewrites the binary code in memory, rather than modifying the executable file. Each different scheme has advantages, and all are used to extend existing applications.

5.4. Intra-address space protection

The palladium project [12] uses techniques similar to MMA for implementing intra-address space protection. By using the x86 memory management hardware, palladium offers distinct protection domains within a single address space for both kernel extensions and applications. However, palladium protection for user-level applications is different than MMA because in palladium, there is a central user mode application that loads different extension modules into its address space. These extension modules then switch domains by using palladium specific system calls. In MMA, rather than having the extension module switch protection domains, a kernel module makes the change on behalf of the application. The advantage of this approach is that the user-mode applications benefit from protection without requiring modification. Implementing UMLinux using palladium is possible but requires modifying run-time libraries and certain applications, whereas MMA allows these programs to run unmodified.

Like palladium, Lava [21] uses hardware to implement intra-address space protection. Support for switching protection domains is implemented in the operating system and communication between protection domains is facilitated through fast IPC. However, using Lava requires modifications to existing user-mode applications.

In addition to hardware protection, there are a number of software tools that offer intra-address space protection. Software based fault isolation [30] breaks address spaces into distinct protection segments. Rewriting binaries before they are run enforces protection. Type-safe [6, 10, 32] and interpreted [2] languages also protect memory by disallowing arbitrary pointers. The main drawback of this approach is that all software that requires protection must be implemented using the specific language. This restriction is not feasible for virtual machines that run existing applications.

5.5. Fast Context Switching

In [27], fast context switching is achieved through compiler and kernel support. Speedups are due to encouraging preemptive context switches at so-called fast context switching points. These fast context switching points are instructions that have no live scratch (caller-saved) registers. Therefore when a context switch occurs, there are less registers that must me written to memory. However, larger tagged L1 and L2 cache found in x86 processors mitigate these effects, eliminating the need for this type of optimization.

The L4 microkernel [29] uses the segmentation hardware found in x86 and Power PC processors to break up linear address spaces into different segments. Switching between processes is just a matter of changing segments, the linear address remains the same. This scheme eliminates the need for expensive TLB flushes when switching between processes. However for this scheme to be effective, the working set of the processes must be small.

Lightweight RPC (LRPC) [5] optimizes the common case for RPC of calling functions on a server that resides on the same machine. In this approach, optimizations are made to improve the networking and parameter passing speed. Essentially, the server and client share a stack, which eliminates the need for marshaling arguments and return values between the two. Although the performance for micro benchmarks is improved, it is not clear that the optimization significantly improves the performance of any current applications that would benefit from fast RPC.

Each of these projects are complementary to the address space overloading capabilities of MMA. MMA is used to enhance the performance of context switching within a guest OS, but does so by providing a fast way to switch in and out sections of address space, something none of the existing projects do.

6. Conclusions

Using KTrace and MMA enables UMLinux to achive excellent performance. By reducing the number of context switches needed to carry out common services and leveraging the flexibility of x86 memory management hardware, UMLinux performs almost as well as VMware in several Macro benchmark tests. As a result, UMLinux is a viable option for implementing secure services below the VMM while still maintaining reasonable performance.

7. Acknowledgments

I am gratefull to Pete Chen for all of the useful discussions we had regarding the material presented in this paper. I would also like to thank Landon Cox, Howard Tsai, and Samantha King for taking the time to review this paper.

8. References

[1] Technical White Paper. Technical report, VMware, Inc., February 1999.

[2] The java programming language, 2002. http://java.sun.com.

[3] C. P. Attanasio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: a study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *Proceedings of the ACM SIGPLAN*, 35(5):1–12, 2000.

[5] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[6] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 267–283, December 1995.

[7] Bitmover. LMbench - Tools for Performance Analysis, 2002. http://www.bitmover.com/lmbench/.

[8] 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.

[9] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[10] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[11] 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)*, May 2001.

[12] Tzi cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Symposium on Operating Systems Principles (SOSP)*, pages 140–153, 1999.

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

[14] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. 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)*, December 2002.

[15] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[16] Alan Eustace and Amitabh Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. Technical Report DEC WRL Technical Note TN-44, Digital Equipment Corporation, July 1994.

[17] Trent Fisher. HURD: Towards a New Strategy of OS Design, 1996. http://www.gnu.org/software/hurd/hurd-paper.html.

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

[19] Rop Gonggrijp. NAH6 Secure Notebook, 2002. http://www.nah6.com/.

[20] 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.

[21] Trent Jaeger, Jochen Liedtke, and Nayeem Islam. Operating System Protection for Fine-Grained Programs. In *Proceedings of USENIX Security Symposium*, pages 143–157, January 1998.

[22] Samuel King, Adam Everspagh, Tun-Han Lee, Peter Chen, and Atul Prakash. VM-Guard: A Virtual Machine-Based Intrusion Detection and Prevention System. In *rejected from ACM Conference on Computer Security (ACMCCS)*, December 2002.

[23] 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.

[24] Flux OSKit. The OSKit Project, 2002. http://www.cs.utah.edu/flux/oskit.

[25] 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 Conference*, August 2000.

[26] 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.

[27] Jeffrey S. Snyder, David B. Whalley, and Theodore P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, pages 35–42, 1995.

[28] 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.

[29] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of Address-Space Multiplexing on the Pentium. In *Interner Bericht 2002-1, Fakultat fur Informatik*, May 2002.

[30] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[31] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *submitted to Proceedings of the 2002 USENIX Technical Conference*, 2002.

[32] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Symposium on Operating Systems Principles*, pages 1–14, 2001.