NOVA: A Microhypervisor-Based Secure Virtualization Architecture

Udo Steinberg
Technische Universität Dresden
udo@hypervisor.org

Bernhard Kauer
Technische Universität Dresden
bk@vmmon.org

Abstract

The availability of virtualization features in modern CPUs has reinforced the trend of consolidating multiple guest operating systems on top of a hypervisor in order to improve platform-resource utilization and reduce the total cost of ownership. However, today's virtualization stacks are unduly large and therefore prone to attacks. If an adversary manages to compromise the hypervisor, subverting the security of all hosted operating systems is easy. We show how a thin and simple virtualization layer reduces the attack surface significantly and thereby increases the overall security of the system. We have designed and implemented a virtualization architecture that can host multiple unmodified guest operating systems. Its trusted computing base is at least an order of magnitude smaller than that of existing systems. Furthermore, on recent hardware, our implementation outperforms contemporary full virtualization environments.

Categories and Subject Descriptors
D.4.6 [*Operating Systems*]: Security and Protection;
D.4.7 [*Operating Systems*]: Organization and Design;
D.4.8 [*Operating Systems*]: Performance

General Terms Design, Performance, Security

Keywords Virtualization, Architecture

1. Introduction

Virtualization is used in many research and commercial environments to run multiple legacy operating systems concurrently on a single physical platform. Because of the increasing importance of virtualization, the security aspects of virtual environments have become a hot topic as well.

The most prominent use case for virtualization in enterprise environments is server consolidation. Operating systems are typically idle for some of the time, so that hosting several of them on a single physical machine can save computing resources, power, cooling, and floor space in data centers. The resulting reduction in total cost of ownership makes virtualization an attractive choice for many vendors.

However, virtualization is not without risk. The security of each hosted operating system now additionally depends on the security of the virtualization layer. Because penetration of the virtualization software compromises all hosted operating systems at once, the security of the virtualization layer is of paramount importance. As virtualization becomes more prevalent, attackers will shift their focus from breaking into individual operating systems to compromising entire virtual environments [20, 28, 40].

We propose to counteract emerging threats to virtualization security with an architecture that minimizes the trusted computing base of virtual machines. In this paper, we describe the design and implementation of NOVA — a secure virtualization architecture, which is based on small and simple components that can be independently designed, developed, and verified for correctness. Instead of decomposing an existing virtualization environment [27], we took a from-scratch approach centered around fine-grained decomposition from the beginning. Our paper makes the following research contributions:

- We present the design of a decomposed virtualization architecture that minimizes the amount of code in the privileged hypervisor. By implementing virtualization at user level, we trade improved security and lower interface complexity for a slight decrease in performance.
- Compared to existing full virtualization environments, our work reduces the trusted computing base of virtual machines by at least an order of magnitude.
- We show that the additional communication overhead in a component-based system can be lowered through careful design and implementation. By using hardware support for CPU virtualization [37], nested paging [5], and I/O virtualization, NOVA can host fully virtualized legacy operating systems with less performance overhead than existing monolithic hypervisors.

The paper is organized as follows: In Section 2, we provide the background for our work, followed by a discussion of related research in Section 3. Section 4 presents the design of our secure virtualization architecture. In Sections 5, 6, and 7,

http://doi.acm.org/10.1145/1755913.1755935

we describe the microhypervisor, the root partition manager, and the user-level virtual-machine monitor. We evaluate the performance of our implementation in Section 8 and compare it to other virtualization environments. Section 9 summarizes our results and outlines possible directions for future work and Section 10 presents our conclusions.

2. Background

Virtualization is a technique for hosting different guest operating systems concurrently on the same machine. It dates back to the mid-1960s [11] and IBM's mainframes. Unpopular for a long time, virtualization experienced a renaissance at the end of the 1990s with Disco [6] and the commercial success of VMware. With the introduction of hardware support for full virtualization in modern x86 processors [5, 37], new virtualization environments started to emerge. Typical implementations add a software abstraction layer that interposes between the hardware and the hosted operating systems. By translating between virtual devices and the physical devices of the platform, the virtualization layer facilitates sharing of resources and decouples the guest operating systems from hardware.

The most privileged component of a virtual environment, which runs directly on the hardware of the host machine is called hypervisor. The functionality of the hypervisor is similar to that of an OS kernel: abstracting from the underlying hardware platform and isolating the components running on top of it. In our system, the hypervisor is accompanied by multiple user-level virtual-machine monitors (VMMs) that manage the interactions between virtual machines and the physical resources of the host system. Each VMM exposes an interface that resembles real hardware to its virtual machine, thereby giving the guest OS the illusion of running on a bare-metal platform. It should be noted that our terminology differs from that used in existing literature where the terms hypervisor and VMM both denote the single entity that implements the virtualization functionality. In our system we use the terms hypervisor and VMM to refer to the privileged kernel and the deprivileged user component respectively.

Virtualization can be implemented in different ways [1]. In a fully virtualized environment, a guest operating system can run in its virtual machine without any modifications and is typically unaware of the fact that it is being virtualized. Paravirtualization is a technique for reducing the performance overhead of virtualization by making a guest operating system aware of the virtualization environment. Adding the necessary hooks to a guest OS requires access to its source code. When the source code is unavailable, binary translation can be used. It rewrites the object code of the guest OS at runtime, thereby replacing sensitive instructions with traps to the virtualization layer.

Virtualization can positively or negatively impact security, depending on how it is employed. The introduction of a virtualization layer generally increases the attack surface and makes the entire system more vulnerable because, in addition to the guest operating system, the hypervisor and VMM are susceptible to attacks as well. However, security can be improved if virtualization is then used to segregate functionality that was previously combined in a non-virtualized environment. For example, a system that moves the firewall from a legacy operating system into a trusted virtual appliance retains the firewall functionality even when the legacy OS has been fully compromised.

3. Related Work

3.1 Microkernels

We share our motivation of a small trusted computing base with microkernel-based systems. These systems take an extreme approach to the principle of least privilege by using a small kernel that implements only a minimal set of abstractions. Liedtke [24] identified three key abstractions that a microkernel should provide: address spaces, threads, and inter-process communication. Additional functionality can be implemented at user level. The microkernel approach reduces the size and complexity of the kernel to an extent that formal verification becomes feasible [19]. However, it implies a performance overhead because of additional communication. Therefore, most of the initial work on L4 focused on improving the performance of IPC [25]. Inspired by EROS [32], recent microkernels introduced capabilities for controlling access to kernel objects [3, 19, 21, 33]. In order to support legacy applications in a microkernel environment, these systems have traditionally hosted a paravirtualized legacy operating system, such as L4Linux [13]. Industrial deployments such as OKL4, VMware MVP, and VirtualLogix use paravirtualization in embedded systems where hardware support for full virtualization is still limited. Microkernel techniques have also been employed for improving robustness in Minix [14], to support heterogeneous cores in the Barrelfish multikernel [3], and to provide high-assurance guarantees in the Integrity separation kernel [18]. The NOVA microhypervisor and microkernels share many similarities. The main difference is NOVA's consideration of full virtualization as a primary objective.

3.2 Hypervisors

The microkernel approach has been mostly absent in the design of full virtualization environments. Instead, most of the existing solutions implement hardware support for virtualization in large monolithic hypervisors. By applying microkernel construction principles in the context of full virtualization, NOVA bridges the gap between traditional microkernels and hypervisors. In Figure 1, we compare the size of the trusted computing base for contemporary virtual environments. The total height of each bar indicates how much the attack surface of an operating system increases when it runs inside a virtual machine rather than on bare

hardware. The lowermost box shows the size of the most privileged component that must be fully trusted.

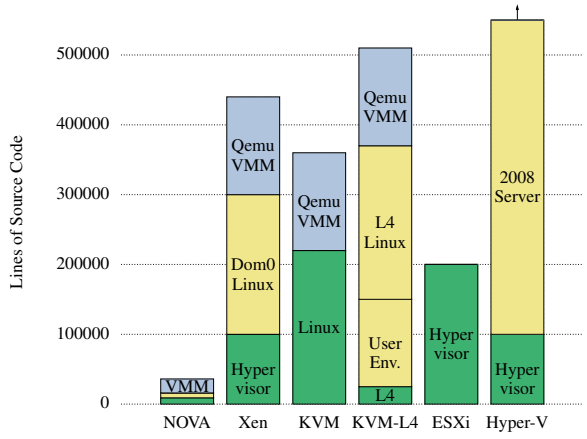

Figure 1: Comparison of the TCB size of virtual environments. NOVA consists of the microhypervisor (9 KLOC), a thin user-level environment (7 KLOC), and the VMM (20 KLOC). For Xen, KVM, and KVM-L4 we assume that all unnecessary functionality has been removed from the Linux kernel, so that it is devoid of unused device drivers, file systems, and network support. We estimate that such a kernel can be shrunk to 200 KLOC. KVM adds approximately 20 KLOC to Linux. By removing support for non-x86 architectures, QEMU can be reduced to 140 KLOC.

The Xen [2] hypervisor has a size of approximately 100 thousand lines of source code (KLOC) and executes in the most privileged processor mode. Xen uses a privileged "domain zero", which hosts Linux as a service OS. Dom0 implements management functions and host device drivers with direct access to the platform hardware. QEMU [4] runs as a user application on top of Linux and provides virtual devices and an instruction emulator. Although Dom0 runs in a separate virtual machine, it contributes to the trusted computing base of all guest VMs that depend on its functionality. In our architecture privileged domains do not exist. KVM [17] adds support for hardware virtualization to Linux and turns the Linux kernel with its device drivers into a hypervisor. KVM also relies on QEMU for implementing virtual devices and instruction emulation. Unlike Xen, KVM can run QEMU and management applications directly on top of the Linux hypervisor in user mode, which obviates the need for a special domain. Because it is integrated with the kernel and its drivers, Linux is part of the trusted computing base of KVM and increases the attack surface accordingly. KVM-L4 [29] is a port of KVM to L4Linux, which runs as a paravirtualized Linux kernel on top of an L4 microkernel. When used as a virtual environment, the trusted computing base of KVM-L4 is even larger than that of KVM. However, KVM-L4 was designed to provide a small TCB for L4 applications running side-by-side with virtual machines while reusing a legacy VMM for virtualization. In NOVA, the trusted computing base is extremely small both for virtual machines and for applications that run directly on top of the microhypervisor.

Commercial virtualization solutions have also aimed for a reduction in TCB size, but are still an order of magnitude larger than our system. VMware ESXi [39] is based on a 200 KLOC hypervisor [38] that supports management processes running in user mode. In contrast to our approach, ESXi implements device drivers and VMM functionality inside the hypervisor. Microsoft Hyper-V [26] uses a Xen-like architecture with a hypervisor of at least 100 KLOC [22] and a privileged parent domain that runs Windows Server 2008. It implements instruction and device emulation and provides drivers for even the most exotic host devices, at the cost of inflating the TCB size. For ESXi and Hyper-V, we cannot conduct a more detailed analysis because the source code is not publicly available.

A different idea for shrinking the trusted computing base is splitting applications [35] to separate security-critical parts from the rest of the program at the source-code level. The critical code is executed in a secure domain while the remaining code runs in an untrusted legacy OS. ProxOS [36] partitions application interfaces by routing security-relevant system calls to trusted VMs.

Virtual machines can provide additional security to an operating system or its applications. SecVisor [31] uses a small hypervisor to defend against kernel code injection. Bitvisor [34] is a hypervisor that intercepts device I/O to implement OS-transparent data encryption and intrusion detection. Overshadow [7] protects the confidentiality and integrity of guest applications in the presence of a compromised guest kernel by presenting the kernel with an encrypted view on application data. In contrast to these systems, the goal of our work is not to retrofit guest operating systems with additional protection mechanisms, but to improve the security of the virtualization layer itself.

Virtualization can also be used to replay [10], debug [16], and live-migrate [9] operating systems. These concepts are orthogonal to our architecture and can be implemented on top of the NOVA microhypervisor in the user-level VMM. However, they are outside the scope of this paper.

4. NOVA OS Virtualization Architecture

In this section, we present the design of our architecture, which adheres to the following two construction principles:

1. Fine-grained functional decomposition of the virtualization layer into a microhypervisor, root partition manager, multiple virtual-machine monitors, device drivers, and other system services.
2. Enforcement of the principle of least privilege among all of these components.

We show that the systematic application of these principles results in a minimized trusted computing base for user applications and VMs running on top of the microhypervisor.

We do not use paravirtualization in our system, because we neither want to depend on the availability of source code for the guest operating systems nor make the extra effort of porting operating systems to a paravirtualization interface. Depending on the OS, the porting effort might be necessary for each new release. That said, the NOVA design does not inherently prevent the application of paravirtualization techniques. If desired, explicit hypercalls from an enlightened guest OS to the VMM are possible. We also chose not to use binary translation, because it is a rather complex technique. Instead our architecture relies on hardware support for full virtualization, such as Intel VT or AMD-V, both of which have been available in processors for several years now. Figure 2 depicts the key components of our architecture.

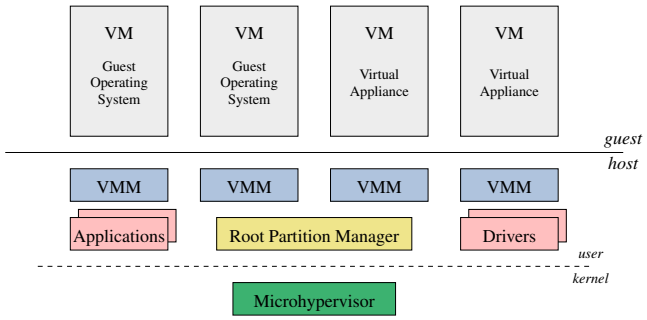

Figure 2: NOVA consists of the microhypervisor and a user-level environment that contains the root partition manager, virtual-machine monitors, device drivers, and special-purpose applications that have been written for or ported to the hypercall interface.

The hypervisor is the only component that runs in the most privileged processor mode (host mode, ring 0). According to our first design principle, the hypervisor should be as small as possible, because it belongs to the trusted computing base of every other component in the system. Therefore, we implemented policies and all functionality that is neither security- nor performance-critical outside the hypervisor. The resulting microhypervisor comprises approximately 9000 lines of source code and provides only simple mechanisms for communication, resource delegation, interrupt control, and exception handling. Section 5 describes these mechanisms in detail. The microhypervisor drives the interrupt controllers of the platform and a scheduling timer. It also controls the memory-management unit (MMU) and the IOMMU — if the platform provides one. User applications run in host mode, ring 3, virtual machines in guest mode. A virtual machine can host a legacy operating system with its applications or a virtual appliance. A virtual appliance is a prepackaged software image that consists of a small kernel and few special-purpose applications. Secure virtual appliances, such as a microkernel with an online banking application, benefit from a small trusted computing base for virtual machines. Each user application or virtual machine has its own address space. Applications such as the VMM manage these address spaces using the hypercall interface. The VMM exposes an interface to its guest operating system that resembles real hardware. Because of the need to emulate virtual devices and sensitive instructions, this interface is as wide and complex as the x86 architecture, with its legacy devices and many obscure corner cases.

Apart from the VMM, the user environment on top of the microhypervisor contains applications that provide additional OS functionality such as device drivers, file systems, and network stacks to the rest of system. We have written NOVA-specific drivers for most legacy devices (keyboard, interrupt controllers, timers, serial port, VGA) and standardized driver interfaces (PCI, AHCI). For vendor-specific hardware devices, we hope to benefit from external work that generates driver code [8, 30] or wraps existing drivers with a software adaptation layer. On platforms with an IOMMU, NOVA facilitates secure reuse of existing device drivers by directly assigning hardware devices to driver VMs [23].

4.1 Attacker Model

Before we outline possible attacks on a virtual environment and illustrate how our architecture mitigates the impact of these attacks, we describe the attacker model. We assume that an attacker cannot tamper with the hardware, and that firmware such as CPU microcode, platform BIOS, and code running in system management mode is trustworthy. Furthermore, we assume that an attacker is unable to violate the integrity of the booting process. This means, that code and data of the virtualization layer cannot be altered during booting, or that trusted-computing techniques such as authenticated booting and remote attestation can be used to detect all modifications [12]. However, an attacker can locally or remotely modify the software that runs on top of the microhypervisor. He can run malicious guest operating systems in virtual machines, install hostile device drivers that perform DMA to arbitrary memory locations, or use a flawed user-level VMM implementation.

4.2 Attacks on Virtual Environments

For attacks that originate from inside a virtual machine, we make no distinction between a malicious guest kernel and user applications that may first have to compromise their kernel through an exploit. In both cases, the goal of the attacker is to escape the virtual machine and to take over the host. The complexity of the interface between a virtual machine and the host defines the attack surface that can be leveraged by a malicious guest operating system to attack the virtualization layer. Because we implemented the virtualization functionality outside the microhypervisor in the VMM, our architecture splits the interface in two parts: 1) The microhypervisor provides a simple message-passing interface that transfers guest state from the virtual machine to the VMM and back. 2) The VMM emulates the complex x86 interface and uses the message-passing interface to control execution of its associated virtual machine.

Guest Attacks

By exploiting a bug in the x86 interface, a VM can take control of or crash its associated virtual-machine monitor. We achieve an additional level of isolation by using a dedicated VMM for each virtual machine. Because a compromised virtual-machine monitor only impairs its associated VM, the integrity, confidentiality, and availability of the hypervisor and other VMs remains unaffected. A vulnerability in the x86 interface will be common across all instances of a particular VMM. However, each guest operating system that exploits the bug can only compromise its own VMM and virtual machines that do not trigger the bug or use a different VMM implementation will remain unaffected. In the event of a compromised virtual-machine monitor, the hypervisor continues to preserve the isolation between virtual machines.

To attack the hypervisor, a virtual machine would have to exploit a flaw in the message-passing interface that transfers state to the VMM and back. Given the low complexity of that interface and the fact that VMs cannot perform hypercalls, a successful attack is unlikely. In other architectures where all or parts of the virtualization functionality are implemented in the hypervisor, a successful attack on the x86 interface would compromise the whole virtual environment. In our architecture the impact is limited to the affected virtual-machine monitor. A virtual machine cannot attack any other part of the system directly, because it can only communicate with its associated virtual-machine monitor.

VMM Attacks

If a virtual machine has taken over its VMM, the attack surface increases, because the VMM can use the hypercall interface and exploit any flaw in it. However, from the perspective of the hypervisor, the VMM is an ordinary untrusted user application with no special privileges.

Virtual-machine monitors cannot attack each other directly, because each instance runs in its own address space and direct communication channels between VMMs do not exist. Instead an attacker would need to attack the hypervisor or another service that is shared by multiple VMMs, e.g., a device driver. Device drivers use a dedicated communication channel for each VMM. When a malicious VMM performs a denial-of-service attack by submitting too many requests, the driver can throttle the communication or shut the channel to the virtual-machine monitor down.

Device-Driver Attacks

The primary security concern with regard to device drivers is their use of DMA. If the platform does not include an IOMMU, then any driver that performs DMA must be trusted, because it has access to the entire memory of the system. On newer platforms that provide an IOMMU, the hypervisor restricts the usage of DMA for drivers to regions of memory that have been explicitly delegated to the driver. Each device driver that handles requests from multiple VMs is responsible for separating their concerns. A compromised or malicious driver can only affect the availability of its device and the integrity and confidentiality of all memory regions delegated to it. Therefore, if a VMM delegates the entire guest-physical memory of its virtual machine to a driver, then the driver can manipulate the entire guest. However, if the VMM delegates only the guest's DMA buffers, then the driver can only corrupt the data or transfer it to the buffers of another guest. Using the IOMMU, the hypervisor blocks DMA transfers to its own memory region and restricts the interrupt vectors available to drivers. In other architectures where drivers are part of the hypervisor, an insecure device driver can undermine the security of the entire system.

Remote Attacks

Remote attackers can access the virtual environment through a device such as a network card. By sending malformed packets, both local and remote attackers can potentially compromise or crash a device driver and then proceed with a device-driver attack. In our architecture the impact of the exploit is limited to the driver, whereas in architectures with in-kernel drivers the whole system is at risk. Because we treat the inside of a virtual machine as a black box, we make no attempt to prevent remote exploits that target a guest operating system inside a VM. If such protection is desired, then a VMM can take action to harden the kernel inside its VM, e.g., by making regions of guest-physical memory corresponding to kernel code read-only. However, such implementations are beyond the scope of this paper.

5. Microhypervisor

The microhypervisor implements a capability-based hypercall interface organized around five different types of kernel objects: protection domains, execution contexts, scheduling contexts, portals, and semaphores.

For each newly created kernel object, the hypervisor installs a capability that refers to that object in the capability space of the creator protection domain. Capabilities are opaque and immutable to the user; they cannot be inspected, modified, or addressed directly. Instead, applications access a capability via a capability selector. The capability selector is an integral number (similar to a Unix file descriptor) that serves as an index into the domain's capability space. The use of capabilities leads to fine-grained access control and supports our design principle of least privilege among all components. Initially, the creator protection domain holds the only capability to the object. Depending on its own policy, it can then delegate copies of the capability with the same or reduced permissions to other domains that require access to the object. Because the hypercall interface uses capabilities for all operations, each protection domain can only access kernel objects for which it holds the corresponding capabilities. In the following paragraphs we briefly describe each kernel object. The remainder of this section details the mechanisms provided by the microhypervisor.

Protection Domain: The kernel object that implements spatial isolation is the protection domain. A protection domain acts as resource container and abstracts from the differences between a user application and a virtual machine. Each protection domain consists of three spaces: The memory space manages the page table, the I/O space manages the I/O permission bitmap, and the capability space controls access to kernel objects.

Execution Context: Activities in a protection domain are called execution contexts. Execution contexts abstract from the differences between threads and virtual CPUs. They can execute program code, manipulate data, and use portals to send messages to other execution contexts. Each execution context has its own CPU/FPU register state.

Scheduling Context: In addition to the spatial isolation implemented by protection domains, the hypervisor enforces temporal separation. Scheduling contexts couple a time quantum with a priority and ensure that no execution context can monopolize the CPU. The priority of a scheduling context reflects its importance. The time quantum facilitates round-robin scheduling among scheduling contexts with equal importance.

Portal: Communication between protection domains is governed by portals. Each portal represents a dedicated entry point into the protection domain in which the portal was created. The creator can subsequently grant other protection domains access to the portal in order to establish a cross-domain communication channel.

Semaphore: Semaphores facilitate synchronization between execution context on the same or on different processors. The hypervisor uses semaphores to signal the occurrence of hardware interrupts to user applications.

5.1 Scheduling

The microhypervisor implements a preemptive priority-driven round-robin scheduler with one runqueue per CPU. The scheduling contexts influence how the microhypervisor makes dispatch decisions. When invoked, the scheduler selects the highest-priority scheduling context from the runqueue and dispatches the execution context attached to it. The scheduler is oblivious as to whether an execution context is a thread or a virtual CPU. Once dispatched, the execution context can run until the time quantum of its scheduling context is depleted or until it is preempted by the release of a higher-priority scheduling context.

5.2 Communication

We now describe the simple hypervisor message-passing interface that the components of our system use to communicate with each other. Depending on the purpose of the communication, the message payload differs. Figure 3 illustrates the communication between a virtual machine and its user-level virtual-machine monitor as an example. During the creation of a virtual machine, the VMM installs for each type of VM exit a capability to one of its portals in the capability space of the new VM. In case of a multiprocessor VM, every virtual CPU has its own set of VM-exit portals and a dedicated handler execution context in the VMM.

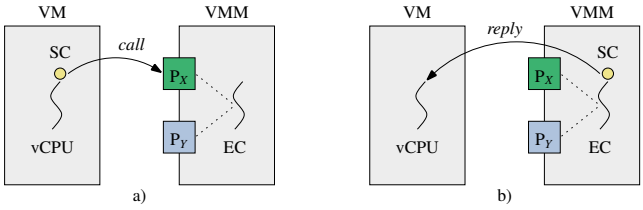

Figure 3: Communication between a virtual CPU (vCPU) in a VM and the corresponding handler execution context (EC) in the VMM. a) Upon a VM exit, the hypervisor delivers a message on behalf of the virtual CPU through an event-specific portal P_X that leads to the VMM. The communication is similar to a remote function call because the virtual CPU donates its scheduling context (SC) to the handler EC bound to the portal. b) After handling the event, the handler EC invokes the reply capability and thereby directly responds to the virtual CPU. The donated SC is automatically returned back to the virtual CPU.

When a virtual CPU executes a sensitive instruction, it generates a VM exit and the hypervisor gains control. After looking up the portal that corresponds to the VM-exit type in the capability space of the virtual machine, the hypervisor delivers a message through that portal to the corresponding handler execution context in the VMM. VM exits and exceptions evaluate the message transfer descriptor stored in the portal to determine what guest state the hypervisor should transmit in the message to the VMM. This performance optimization minimizes the amount of state that must be read from the virtual-machine control structure (VMCS). Access to the VMCS is an expensive operation on older Intel processors. The virtual CPU donates its scheduling context to the execution context of the VMM for the duration of the communication. Because of the donation, the handler in the VMM inherits the priority of the virtual CPU and the hypervisor can directly switch to the VMM without having to invoke the scheduler. Furthermore, the entire handling of the VM exit, which we describe in Section 7, is accounted to the time quantum of the virtual CPU.

During the portal traversal, the hypervisor creates a reply capability that refers to the virtual CPU. Communication ends when the VMM invokes the reply capability to respond with a message that contains new state for the virtual CPU. The hypervisor destroys the reply capability, returns the donated scheduling context, installs the new state, and finally resumes the virtual CPU.

Other client-server communication in the user environment uses the same communication mechanism. The only difference is that clients can explicitly specify the capability for the destination portal during the hypercall. Furthermore, the contents of the message is not guest state, but a protocol-specific number of parameters and return values.

5.3 Memory Management

The microhypervisor maintains a host address space for each protection domain. The host page table of an application in the user-level environment translates host-virtual to host-physical addresses (HVA→HPA); for a virtual machine the host page table translates guest-physical to host-physical addresses (GPA→HPA). A guest operating system that uses paging inside its virtual machine additionally creates for each guest address space a guest page table that translates guest-virtual to guest-physical addresses (GVA→GPA).

On platforms that implement nested paging in hardware, the MMU consults both the guest and host page tables to carry out the two-dimensional GVA→GPA→HPA translation required for performing a memory access [5]. For this lookup no interaction with virtualization software is necessary. If the hardware does not support nested paging, the microhypervisor must perform the same translation in software and store the result in a shadow page table, which is used by the memory-management unit. In that case, the microhypervisor maintains one shadow page table for each virtual CPU in the system.

Our implementation uses hardware-supported nested paging where possible and otherwise relies on the virtual TLB algorithm [15] for filling and flushing the shadow page tables. The vTLB algorithm requires the microhypervisor to intercept page faults and instructions that cause TLB flushes in the guest operating system (CR writes, INVLPG). To perform a vTLB fill in response to a guest page fault, the microhypervisor must parse the multi-level guest page table. Because the guest page table contains guest-physical addresses, the microhypervisor must, for each level of the guest page table, perform a GPA→HPA translation via the host page table and then read the entry using a suitable HVA→HPA mapping[1].

To accelerate the lookup we use the following trick: The microhypervisor runs on the host page table of the currently active virtual machine, which causes the MMU to reinterpret the GPA→HPA translation as a HVA→HPA translation. Because guest-physical addresses turn into host-virtual addresses, the microhypervisor can directly dereference guest page table entries. The two-dimensional page walk becomes one-dimensional (GVA→GPA) for software and the MMU handles the other dimension (GPA/HVA→HPA) transparently. However, the microhypervisor must take action to recover from a page fault that could occur when dereferencing a guest page table entry pointing outside the mapped guest-physical address space. Because the microhypervisor occupies the top part of each host page table, the size of guest-physical memory is restricted to 3 GB on 32-bit platforms. We compare the performance of memory virtualization using a virtual TLB to hardware-based nested paging in Section 8.

[1] Physical memory cannot be read directly when paging is active.

6. Root Partition Manager

The microhypervisor does not contain a policy for resource allocations. At boot time it claims its own memory area and the memory and I/O resources of security-critical devices such as interrupt controllers and IOMMUs. Afterwards the microhypervisor creates the first protection domain, the root partition manager, which receives capabilities for all remaining memory regions, I/O ports, and interrupts. The root partition manager performs the initial resource allocation decisions. Like any other protection domain, it can create and destroy new protection domains and, by delegating the corresponding capabilities, assign resources to them. The creator of a protection domain obtains a capability that can be used to destroy the domain. This scheme resembles the recursive address-space model of L4 [24] and facilitates flexible and dynamic delegation and revocation of resources, with the ability to make policy decisions at each level [21].

The microhypervisor implements the mechanism for transferring capabilities during communication. The sender specifies in the message transfer descriptor one or more regions of its memory space, I/O space, or capability space and can optionally reduce the access permissions during the transfer. The receiver declares a region where it is willing to accept resource delegations. The microhypervisor only delegates resources that fit into both the sender- and receiver-specified ranges. By revoking the respective capabilities, an execution context can recursively withdraw any resources that have previously been delegated from its domain to other protection domains.

7. Virtual-Machine Monitor

The virtual-machine monitor (VMM) runs as a user-level application in an address space on top of the microhypervisor and supports the execution of an unmodified guest operating system in a virtual machine. It emulates sensitive instructions and provides virtual devices. The VMM manages the guest-physical memory of its associated virtual machine by mapping a subset of its own address space into the host address space of the VM. The VMM can also map any of its I/O ports and MMIO regions into the virtual machine to grant direct access to a hardware device. The VMM is the handler for all VM-exit events that occur in its associated virtual machine. For this purpose, it creates a dedicated portal for each event type and sets the transfer descriptor in the portals such that the microhypervisor transmits only the architectural state required for handling the particular event. For example, the virtual-machine monitor configures the portal corresponding to the CPUID instruction with a transfer descriptor that includes only the general-purpose registers, instruction pointer, and instruction length.

When a VM exit occurs, the microhypervisor sends a message to the portal corresponding to the VM-exit event and transfers the requested architectural state of the virtual CPU to the handler execution context in the VMM. The

virtual-machine monitor can determine the type of virtualization event from the portal that was called and then execute the correct handler function. The emulation of simple instructions like CPUID is straightforward: The VMM loads the general-purpose registers with new values and advances the instruction pointer to point behind the instruction that caused the VM exit. By invoking the reply capability, the VMM transmits the updated state to the microhypervisor and the virtual CPU can resume execution. When the guest operating system tries to access a device using memory-mapped I/O, the virtual CPU generates a host page fault because the corresponding region of guest-physical memory is not mapped in the VM. In such cases, the hardware does not provide more information than the fault address and instruction pointer. Therefore, the VMM needs to obtain additional information by decoding the faulting instruction.

7.1 Instruction Emulator

The VMM performs the emulation of faulting instructions using the following steps: It fetches the opcode bytes of the instruction from the guest's instruction pointer and then uses an instruction decoder to determine the length and operands of the instruction. If the operands are memory operands, the instruction emulator fetches them as well. To execute simple instructions, the emulator calls an instruction-specific assembly snippet and for more complex instructions a C function. The execution of an instruction can cause an exception such as division by zero. The VMM provides fixup code to handle such cases. After the emulation is complete, the VMM writes the results of the execution back to registers or memory and advances the instruction pointer. The current implementation of the emulator is mature enough to run even obscure programs such as bootloaders.

7.2 Device Emulation

The virtual-machine monitor provides virtual devices for its guest operating system. Each virtual device is modeled as a software state machine that mimics the behavior of the corresponding hardware device. Whenever an instruction reads from or writes to an I/O port or memory-mapped I/O register, the VMM updates the state machine of the corresponding device model similar to how the hardware device would update its internal state. Not all reads and writes to a virtual device require interception by the VMM. For example, the frame buffer of a graphics device can be mapped directly into the virtual machine. Likewise, device registers without read side effects can be mapped read-only.

A careful selection of virtual devices in the VMM is beneficial for lowering the I/O virtualization overhead. The emulation of a device that requires fewer VM exits to be programmed is preferable, because it reduces the number of round-trips to the VMM. Devices with DMA support and interrupt coalescing can additionally limit the overhead for copying data and handling interrupts.

7.3 Interaction with Host Device Drivers

Whenever the guest operating system has programmed its virtual device to perform an operation such as a disk read, the VMM needs to contact the device driver for the host device to deliver the data. Figure 4 shows how the virtual-machine monitor handles a read request on the virtual disk controller.

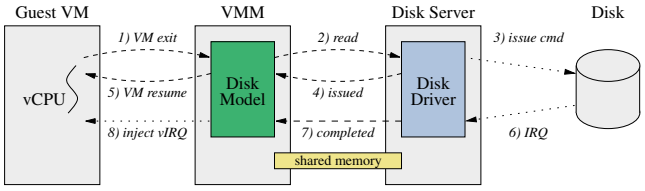

Figure 4: Interaction of a device model with the driver of the corresponding host device. Dashed lines indicate inter-process communication and dotted lines denote hardware operations.

When the virtual CPU performs a memory-mapped I/O access, it causes a VM exit (1). The microhypervisor sends a fault message to the virtual-machine monitor because the region of guest-physical memory corresponding to the disk controller is not mapped in the host address space of the virtual machine. The VMM decodes the instruction and determines that the instruction accesses the virtual disk controller. By executing the instruction, the virtual-machine monitor updates the state machine of the disk model.

After the guest operating system has programmed the command register of the virtual disk controller to read a block, the VMM sends a message to the disk server to request the data (2). The device driver in the disk server programs the physical disk controller with a command to read the block into memory (3). The disk driver requests a DMA transfer of the data directly into the memory of the virtual machine. It then returns control back to the VMM (4), which resumes the virtual machine (5).

Once the block has been read from disk, the disk controller generates an interrupt to signal completion (6). The disk server writes completion records for all finished requests into a region of memory shared with the VMM. Once the VMM has received a notification message that disk operations have completed (7), it updates the state machine of the device model to reflect the completion and signals an interrupt at the virtual interrupt controller. During the next VM exit, the VMM injects the pending interrupt into the virtual machine (8).

7.4 BIOS Virtualization

Most operating systems require the support of the BIOS at boot time. The BIOS provides functions for screen output and disk access until the corresponding drivers of the operating system have been loaded. Other virtualization implementations inject a virtual BIOS into the guest operating system and execute that BIOS inside the VM. This approach has the disadvantage that every I/O operation performed by the virtual BIOS causes a fault. Furthermore, the execution

of BIOS code is a slow process because most of the code runs in real-mode and needs to be emulated.

A more efficient solution is to move the BIOS into the virtual-machine monitor, which facilitates direct access to the device models without expensive transitions between the virtual machine and the VMM. Furthermore, the code of the virtual BIOS can be hidden from the guest OS. In our implementation, the BIOS is integrated with the VMM. It provides the standard BIOS services to guest operating systems and implements booting via the multiboot standard.

7.5 Multiprocessor Virtualization

The NOVA interface supports the virtualization of multiprocessor guest operating systems. The microhypervisor provides semaphores for cross-processor synchronization and implements a hypercall that allows the VMM to recall the virtual CPUs of its associated virtual machine.

For a virtual machine with multiple processors, the VMM creates the desired number of virtual CPUs and then maps them to physical processors by assigning appropriate scheduling contexts. For each virtual CPU, there exists a dedicated handler in the form of an execution context in the VMM, which resides on the same physical processor as the virtual CPU. Because each handler maintains its own instance of the instruction emulator, the VMM can handle most VM exits by different virtual CPUs in parallel, without the need for any locking or cross-processor signaling.

However, when multiple virtual CPUs access a device model concurrently, the VMM must serialize the operations in such a way that the atomicity of transactions is maintained and state changes follow the expected behavior of the device. For simple cases such as toggling a status bit in a device register, the use of atomic instructions is sufficient. For updates to more complex data structures, the VMM employs the semaphore interface of the microhypervisor to guarantee mutual exclusion.

When an interrupt becomes pending for a virtual CPU that is currently running, the VMM issues a recall operation. Upon a recall, the microhypervisor forces the virtual CPU to take a VM exit, which sends it back to the virtual-machine monitor, so that the VMM can inject the interrupt in a timely manner. For example, when a guest OS broadcasts an interprocessor interrupt to perform a global TLB shootdown, the VMM recalls all virtual CPUs of its associated VM to inject the pending interrupt. The interrupt handler of the guest operating system then runs on each virtual CPU and executes the TLB flush operation locally.

8. Evaluation

We evaluated the performance of our implementation on an Intel machine based on a DX58SO motherboard with 3GB DDR3 RAM, a Core i7 CPU with 2.67 GHz, and a 250GB Hitachi SATA disk. To improve the accuracy of our results, we disabled Hyper-Threading and Turbo Boost on the CPU.

8.1 Linux Kernel Compilation

Our first benchmark measures the wall-clock time required for compiling an x86 Linux 2.6.32 kernel from source code using the default configuration. The compilation starts with a cold buffer cache and uses 4 parallel jobs to minimize the effect of disk delays. The results are the median of several dozen trials. The bars labeled *Native* in Figure 5 are our baseline and show the time for compiling Linux on a bare-metal machine running Linux 2.6.32 with one physical CPU and 512 MB RAM. The other bars show the same Linux system running in different virtual environments, each with one virtual CPU and 512 MB of guest-physical memory.

To measure the overhead of nested paging, we created a virtual machine that used large host pages and was configured not to generate any VM exits during the benchmark. For this purpose we disabled all intercepts and assigned all host devices and interrupts directly to the guest. The bar labeled *Direct* shows that using nested paging results in a 0.6% performance decrease, which we attribute to the higher page-walk cost during TLB fills. It should be noted that this bar represents a limit for this particular benchmark and hardware configuration, which no virtual environment using nested paging can exceed. The third bar illustrates that our implementation has an overhead of less than 1% when using nested paging with 2 MB host pages and TLB entries tagged with a virtual processor identifier (VPID). The next four bars present the benchmark numbers for KVM 2.6.32, Xen 3.4.2, VMWare ESXi 4.0 U1[2], and Microsoft Hyper-V Build 7100.

The next set of bars shows the benefit of tagged TLB entries. Without VPIDs the CPU needs to flush the hardware TLB during VM transitions, which adds several seconds to a Linux kernel compilation run. If the VMM uses 4 KB host pages to facilitate fine-grained paging of virtual machines, the overhead increases by approximately 2%, as shown in the third set of bars. The use of small host pages does not result in more VM exits. However, it causes higher pressure on the hardware TLB and leads to more capacity evictions and TLB fills. The fourth set of bars indicates that the overhead increases significantly when nested paging is not available and software must use shadow page tables instead.

We also measured the performance of paravirtualization by running the benchmark in Xen's Dom0 and under L4Linux 2.6.32. In contrast to systems with full virtualization, the paravirtualized environments had direct access to the disk. For Xen we used a modified C library that avoids costly segment register reloads for thread-local storage. Without this modification the performance of Xen is substantially degraded.

For L4Linux, we measured a higher performance overhead than what has been reported in 1997 [13]. Since then, L4Linux has undergone several architectural changes with the goal of reducing the porting effort. In particular, the

[2] Our test system is not in the Hardware Compatibility List for ESXi.

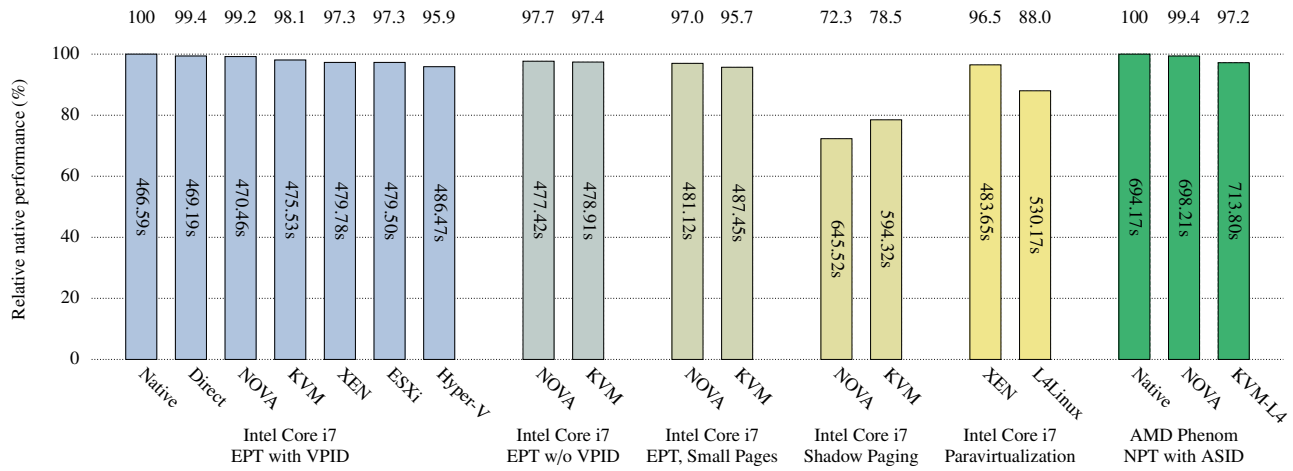

Figure 5: Linux kernel compilation in fully virtualized and paravirtualized environments on Intel Core i7 and AMD Phenom CPUs. More than 99% of native performance can be achieved when nested paging (EPT/NPT), tagged TLBs (VPID/ASID), and large host pages are used.

performance optimization of small address spaces, which uses segmentation to mimic the behavior of a tagged TLB, is no longer present. As a result, each transition between kernel and user mode in L4Linux requires an address space switch and a complete TLB flush. The overhead caused by the TLB flushes and subsequent refills is approximately 9%.

The last set of bars presents the results for the kernel compilation benchmark on an AMD Phenom X3 8450 CPU with 2.1 GHz and 4GB DDR2 RAM. We measured the performance of KVM-L4 on the AMD machine, because KVM-L4 currently does not support Intel CPUs. The results reveal that the performance overhead for NOVA is even lower than on the Intel machine. The primary reason is that AMD CPUs support 4MB host pages with two-level page tables, whereas Intel CPUs use 2MB host pages with four-level page tables.

8.2 Disk Performance

To provide I/O access to a virtual machine, the VMM can either fully virtualize an I/O device or use the IOMMU to grant its VM direct access to a hardware device. We analyze the overhead for both approaches in the following benchmarks. The overhead for full virtualization of an I/O device can be split into three parts:

Device Virtualization: The guest operating system in a virtual machine programs a virtual device using I/O and MMIO instructions. Each such instruction causes a VM exit and requires a round trip to the VMM to emulate the behavior of the device.

Interrupt Virtualization: Platform devices signal the completion of a request by generating an interrupt. Each hardware interrupt causes a VM exit. If the virtual CPU runs with interrupts disabled and the virtual-machine monitor needs to inject a virtual interrupt, another VM exit occurs when the virtual CPU reenables interrupts. Masking, acknowledging, and unmasking the interrupt at the virtual interrupt controller causes up to four more VM exits.

Data Transfer: Copying data from a physical to a virtual device does not cause VM exits, but adds an overhead that depends on the size of the data.

In Figure 6 we compare the performance of our virtual AHCI SATA controller to the same SATA controller in the host machine. The benchmark issues sequential disk reads with different block sizes. To eliminate the influence of the buffer cache, we configured the Linux VM to use direct I/O.

The graph labeled *Native* shows the CPU utilization of the AHCI driver for the hardware device. In our first experiment we assigned the physical AHCI controller directly to the virtual machine. The IOMMU performed the translation of guest-physical to host-physical addresses. The graph labeled *Direct* illustrates the overhead for interrupt virtualization and DMA remapping. In a second experiment, we virtualized the AHCI controller. The VMM intercepts all MMIO operations and copies the DMA descriptors from the virtual device to the host driver. The host driver performs DMA directly to and from the buffers of the virtual machine, which eliminates the need for copying the data. The graph labeled *Virtualized* includes the additional overhead for full device virtualization. With block sizes of less than 8 Kbytes the throughput scales with the block size and request rate and the CPU utilization remains nearly constant. For larger block sizes, the disk bandwidth becomes the limiting factor and the request rate and CPU utilization drop linearly.

The graphs illustrate that the virtualization overhead depends on the number of disk requests, but is independent of the block size. A directly assigned disk controller nearly doubles the CPU utilization. For example, with a block size of 16 Kbytes and a rate of 4100 requests per second, the CPU utilization increases from 3.7% to 7%. At 2.67 GHz this corresponds to 21500 cycles for handling 6 VM exits

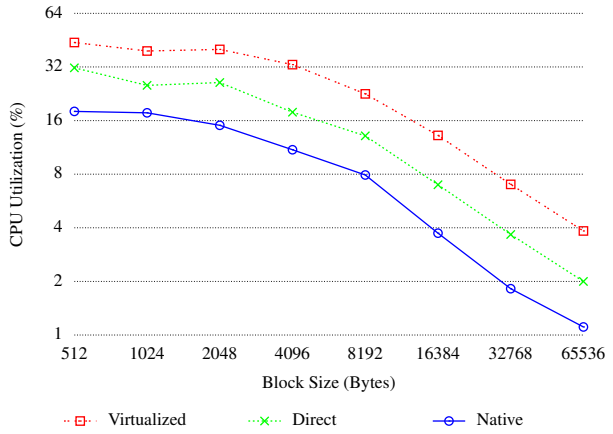

Figure 6: CPU overhead for sequential disk reads with different block sizes. The benchmark compares a fully virtualized and a directly assigned AHCI disk controller with the native counterpart.

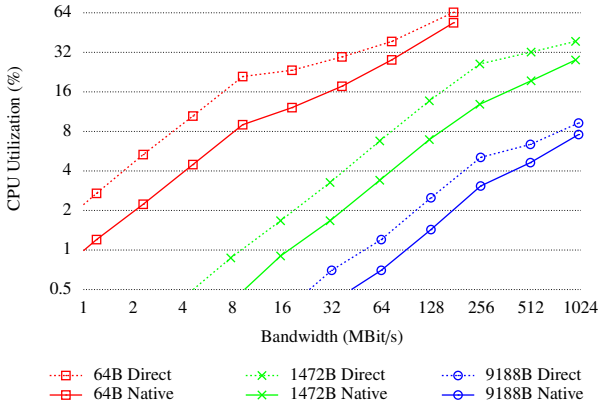

Figure 7: CPU overhead for receiving UDP streams with different bandwidths and packet sizes of 64, 1472 and 9188 bytes. The benchmark compares a directly assigned NIC against a native NIC.

per request. The fully virtualized AHCI controller doubles the CPU utilization again. It requires 6 additional VM exits to intercept the MMIO operations that program the device. MMIO exits are among the most expensive VM exits, because they must be decoded and executed by the instruction emulator.

8.3 Network Performance

We measured the network performance of our implementation with the Netperf benchmark. We configured the sender machine to generate a UDP packet stream with constant bandwidth using a token bucket traffic shaper. The receiver was the Intel Core i7 machine with an Intel 82567 Gigabit Ethernet controller, running Linux 2.6.32. In Figure 7 we compare the CPU utilization for different bandwidths on native hardware and in a virtual machine that had the network controller directly assigned to it.

The interrupt rate on the receiver side depends on the bandwidth of the incoming network stream and the packet size. Interrupt coalescing in the network controller limits the interrupt rate by delaying the generation of the next interrupt until multiple packets have been received. The overhead for network virtualization scales linearly with the network interrupt rate. A network stream that generates twice as many interrupts doubles the virtualization overhead. We compute the actual overhead per interrupt by multiplying the CPU utilization difference with the clock speed and dividing by the interrupt rate. For example, a network stream with a bandwidth of 124 MBit/s and a packet size of 1472 bytes generates approximately 11000 interrupts per second and causes 6 VM exits per interrupt. We measured a 6.7% higher CPU utilization for the virtual machine, which corresponds to an overhead of approximately 16300 cycles per interrupt or 4.3ms per MB. For larger or smaller packets, the overhead decreases or increases respectively. With interrupt coalescing, the maximum rate is approximately 20000 interrupts per second. A further increase in stream bandwidth only generates additional packet processing overhead. Because this overhead is the same for native and virtual environments, the graphs start to converge at that point.

8.4 Transition Times

To determine the overhead introduced by our virtualization environment in more detail, we conducted a series of microbenchmarks to measure the transition times between user and kernel mode and between guest and host mode. Table 1 lists the processors used in the following benchmarks.

CPU Model	Core	Frequency
AMD Opteron 2212	Santa Rosa (K8)	2.00 GHz
AMD Phenom 9550	Agena (K10)	2.20 GHz
Intel Core Duo T2500	Yonah (YNH)	2.00 GHz
Intel Core2 Duo E6600	Conroe (CNR)	2.40 GHz
Intel Core2 Duo E8400	Wolfdale (WFD)	3.00 GHz
Intel Core i7 920	Bloomfield (BLM)	2.67 GHz

Table 1: Processors Used for Microbenchmarks

Figure 8 correlates the transition cost between user and kernel mode (lowermost box) with the basic cost for a message transfer between two threads. The performance of inter-domain communication is crucial for the overall performance of our system because all virtualization events, except for those related to the virtual TLB, require a message to be sent from the microhypervisor to the VMM and back. Furthermore, the VMM uses inter-domain communication to exchange data with host device drivers.

If both threads are located in the same address space, the basic cost for a message transfer comprises the number of clock cycles for entering and leaving the hypervisor (sysenter, sti, sysexit) and the cycles for the hypervisor IPC path (capability lookup, portal traversal, context switch). The actual cost depends on the size of the message that

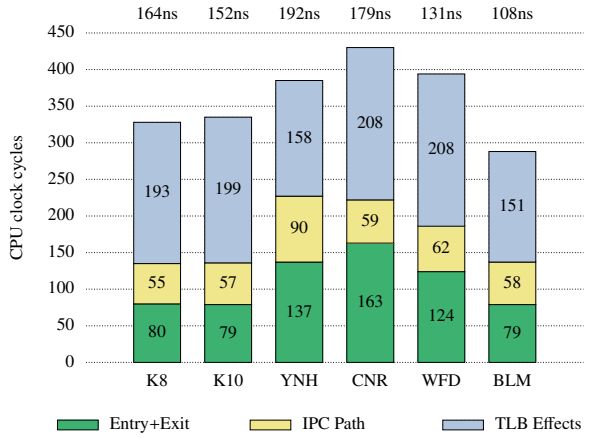

Figure 8: IPC Microbenchmark

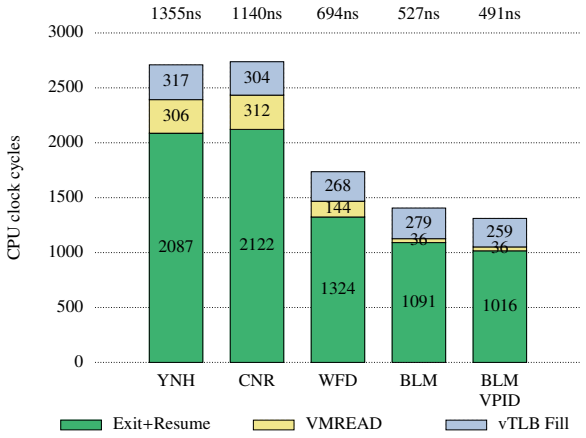

Figure 9: vTLB Miss Microbenchmark

is transferred (additional 2–3 cycles per word). For an IPC between threads in different address spaces there is an extra cost for flushing and repopulating the translation lookaside buffer (TLB effects).

Figure 9 correlates the transition cost between guest and host mode (lowermost box) with the cost for handling a vTLB miss on platforms without hardware-based nested paging. The last two bars show the performance difference when using VPID-tagged TLB entries. The cost for handling a vTLB miss includes the number of clock cycles for the VM exit and subsequent VM resume, the number of clock cycles for performing six VMREAD instructions to determine the cause of the vTLB miss, and the cost for parsing the guest and host page table to update the shadow page table (vTLB fill). The hardware transition cost accounts for almost 80% of the total vTLB miss overhead.

Figure 9 shows that the transition times between guest and host mode decrease with each new processor generation. Because the performance of our system largely depends on these hardware-induced costs, we believe that current hardware trends to lower transition times will further reduce the virtualization overhead in the future.

8.5 Frequency of VM Events

The virtualization overhead depends not only on the cost for transitions between guest and host mode, but also on the frequency of VM exits that cause such transitions. Table 2 shows the distribution of VM exits and related events for the kernel compilation and disk benchmarks.

The most prominent exits with nested paging are port I/O for acknowledging and masking virtual interrupts, followed by external interrupts caused by hardware timers and MMIO accesses that program the virtual disk controller. Without nested paging, the vTLB-related exits (vTLB fills and guest page faults) dominate. The table illustrates that nested paging reduces the number of VM exits by two orders of magnitude. The VM event distribution also shows that Linux issues six MMIO operations for each disk read or write request. An additional six VM exits are caused by interrupt virtualization.

Event	EPT	vTLB	Disk 4k
vTLB Fill		181966391	
Guest Page Fault		13987802	
CR Read/Write		3000321	
vTLB Flush		2328044	
Port I/O	610589	723274	
INVLPG		537270	
Hardware Interrupts	174558	239142	
Memory-Mapped I/O	76285	75151	600102
HLT	3738	4027	101185
Interrupt Window	2171	3371	961
$\sum$	867341	202864793	
Injected vIRQ	*131982*	*177693*	*102507*
Disk Operations	*12715*	*12526*	*100017*
Runtime (seconds)	*470*	*645*	*10*

Table 2: Distribution of Virtualization Events

Dividing the time difference of 1.27 seconds between the second and third bar in Figure 5 by the 867341 VM exits from Table 2 reveals that the average cost for handling a VM exit in NOVA is approximately 3900 cycles. This cost can be broken down as follows: 1016 cycles (26%) are caused by the transition between guest mode and host mode. The transfer of virtual CPU state from the microhypervisor to the VMM and back requires an IPC in each direction and costs approximately 600 cycles (15%). The remaining 59% can be attributed to instruction and device emulation in the VMM. It should be noted that only the overhead of communication between the microhypervisor and the VMM (15% of the total cost for handling a VM exit) is a direct consequence of our decomposed virtualization architecture.

9. Discussion and Future Work

Our evaluation shows that the performance overhead of full virtualization on current hardware can be as low as 1% for memory-bound workloads. We compared memory virtualization using hardware-based nested paging to a software implementation that uses shadow page tables and observed that nested paging can eliminate more than 99% of the VM exits and thereby reduce the virtualization overhead from more than 20% to 1–3%.

We also quantified the overhead of using small host pages, which is a requirement for efficient VM page-sharing algorithms, with 2%. The introduction of tagged TLBs in recent processors eliminates the need for TLB flushes during VM transitions. Figure 8 illustrates that extending hardware support for TLB tags to user address spaces would reduce the inter-domain communication cost in NOVA by 50%. Expensive TLB flushes and refills on address-space switches would become unnecessary.

We are currently extending the virtual BIOS of the VMM to support Windows as a guest operating system. We also research how fair resource scheduling between VMs can be implemented and how virtual machines can guarantee certain real-time properties. Furthermore, we plan to improve the network subsystem with multi-queue SR-IOV network cards, which facilitate direct assignment of network queues to VMs and enable a drastic boost in network performance.

10. Conclusions

In this paper we argue that a small and simple virtualization layer is crucial for the security of hosted guest operating systems. To minimize the attack surface, NOVA takes an extreme microkernel-like approach to virtualization by moving most functionality to user level. Because our entire system adheres to the principle of least privilege, we achieve a trusted computing base that is at least an order of magnitude smaller than that of other full virtualization environments. One interesting result of this work is that such an architecture can be built with negligible performance overhead, thanks to recent hardware virtualization features and careful system design and implementation.

Acknowledgments

We would like to thank our advisor Hermann Härtig for supporting our work on NOVA and our colleagues at TU Dresden for many interesting discussions. We would also like to thank Leonid Ryzhyk, Norman Feske, our shepherd Gernot Heiser, and the anonymous reviewers for comments on earlier versions of this paper. We are grateful to Intel and Microsoft for valuable research grants and extend thanks to Intel Labs and AMD OSRC for providing us with development hardware. This work has been supported by the European Union through PASR grant 104600 (ROBIN).

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13. ACM, 2006.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177. ACM, 2003.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44. ACM, 2009.

[4] F. Bellard. QEMU, A Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 41–46. USENIX Association, 2005.

[5] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two–Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–35. ACM, 2008.

[6] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–156. ACM, 1997.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization–Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13. ACM, 2008.

[8] V. Chipounov and G. Candea. Reverse Engineering of Binary Device Drivers with RevNIC. In *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2010.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286. USENIX Association, 2005.

[10] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual–Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224. ACM, 2002.

[11] R. P. Goldberg. Survey of Virtual Machine Research. *Computer*, 7(6):34–45, 1974.

[12] D. Grawrock. *The Intel Safer Computing Initiative*. Intel Press, 2006.

[13] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The Performance of μ-kernel-based Systems.

In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 66–77. ACM, 1997.

[14] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proceedings of the 6th European Dependable Computing Conference (EDCC)*, pages 3–12. IEEE Computer Society, 2006.

[15] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide*. Intel Corporation, 2008. SKU #253669.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time–Traveling Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–15. USENIX Association, 2005.

[17] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[18] D. N. Kleidermacher. Towards a High Assurance Multi-level Secure PC for Intelligence Communities. In *Proceedings of the IEEE Conference on Technologies for Homeland Security*, pages 609–614. IEEE Computer Society, 2008.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[20] K. Kortchinsky. Cloudburst - Hacking 3D and Breaking out of VMware. In *Black Hat USA*, 2009.

[21] A. Lackorzynski and A. Warg. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *Proceedings of the 2nd Workshop on Isolation and Integration in Embedded Systems (IIES)*, pages 25–30. ACM, 2009.

[22] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Proceedings of the 16th International Symposium on Formal Methods*, pages 806–809. Springer, 2009.

[23] J. LeVasseur, V. Uhlig, J. Stöß, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30. USENIX Association, 2004.

[24] J. Liedtke. On Micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–250. ACM, 1995.

[25] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188. ACM, 1993.

[26] Microsoft Hyper-V. http://www.microsoft.com/hyperv/.

[27] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 151–160. ACM, 2008.

[28] T. Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. In *CanSecWest*, 2007.

[29] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases. In *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems (VTDS)*, pages 18–23. ACM, 2009.

[30] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic Device Driver Synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–86. ACM, 2009.

[31] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350. ACM, 2007.

[32] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185. ACM, 1999.

[33] J. S. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a Verified, General-Purpose Operating System Kernel. In *Proceedings of the 1st NICTA Workshop on Operating System Verification*, pages 1–18. National ICT Australia, 2004.

[34] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Ohyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 5th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. ACM, 2009.

[35] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB Complexity for Security–Sensitive Applications: Three Case Studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 161–174. ACM, 2006.

[36] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 279–292. USENIX Association, 2006.

[37] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38 (5):48–56, 2005.

[38] VMware ESX Server Virtual Infrastructure Node Evaluator's Guide. http://www.vmware.com/pdf/esx_vin_eval.pdf.

[39] VMware ESXi. http://www.vmware.com/esx/.

[40] R. Wojtczuk. Subverting the Xen Hypervisor. In *Black Hat USA*, 2008.