Scale and Performance in the Denali Isolation Kernel

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble
University of Washington
{andrew,mar,gribble}@cs.washington.edu

Abstract

This paper describes the Denali isolation kernel, an operating system architecture that safely multiplexes a large number of untrusted Internet services on shared hardware. Denali's goal is to allow new Internet services to be "pushed" into third party infrastructure, relieving Internet service authors from the burden of acquiring and maintaining physical infrastructure. Our isolation kernel exposes a virtual machine abstraction, but unlike conventional virtual machine monitors, Denali does not attempt to emulate the underlying physical architecture precisely, and instead modifies the virtual architecture to gain scale, performance, and simplicity of implementation. In this paper, we first discuss design principles of isolation kernels, and then we describe the design and implementation of Denali. Following this, we present a detailed evaluation of Denali, demonstrating that the overhead of virtualization is small, that our architectural choices are warranted, and that we can successfully scale to more than 10,000 virtual machines on commodity hardware.

1 Introduction

Advances in networking and computing technology have accelerated the proliferation of Internet services, an application model in which service code executes in the Internet infrastructure rather than on client PCs. Many applications fit this model, including web sites, search engines, and wide area platforms such as content distribution networks, caching systems, and network experimentation testbeds [25]. The Denali project seeks to encourage and enhance the Internet service model by making it possible for untrusted software services to be "pushed" safely into third party hosting infrastructure, thereby separating the deployment of services from the management of the physical infrastructure on which they run.

While this has clear benefits, it also faces difficult technical challenges. One challenge is *scale*: for cost-efficiency and convenience, infrastructure providers will need to multiplex many services on each server machine, as it would be prohibitively expensive to dedicate a separate machine to each service. A second challenge is *security*: infrastructure providers cannot trust hosted services, and services will not trust each other. There must be strong isolation between services, both for security and to enforce fair resource provisioning.

In this paper, we present the design, implementation, and evaluation of the Denali *isolation kernel*, an x86-based operating system that isolates untrusted software services in separate protection domains. The architecture of Denali is similar to that of virtual machine monitors such as Disco [6], VMWare [31], and VM/370 [9]. A virtual machine monitor carves a physical machine into multiple virtual machines; by virtualizing all hardware resources, a VMM can prevent one VM from even naming the resources of another VM, let alone modifying them.

To support unmodified legacy "guest" OSs and applications, conventional VMMs have the burden of faithfully emulating the complete architecture of the physical machine. However, modern physical architectures were not designed with virtualization or scale in mind. In Denali, we have reconsidered the exposed virtual architecture, making substantial changes to the underlying physical architecture to enhance scalability, performance, and simplicity, while retaining the strong isolation properties of VMMs.

For example, although Denali exposes virtual disks and NICs, their interfaces have been redesigned for simplicity and performance. Similarly, Denali exposes an instruction set architecture which is similar to x86 (to gain the performance benefits of directly executing instructions on the host processor), but in which non-virtualizable aspects have been hidden for simplicity, and in which the interrupt model has been changed for scalability.

The cost of Denali's virtual architecture modifications is backwards compatibility: Denali is not able to run unmodified legacy guest operating systems. However, the Denali virtual architecture is complete, in the sense that a legacy operating system could be ported to Denali (although this is still work in progress). To evaluate Denali in the absence of a ported legacy OS, we implemented our own lightweight guest OS, called Ilwaco, which contains a port of the BSD TCP/IP networking stack, thread support, and support for a subset of the POSIX API. We have ported several applications to Ilwaco, including a web server, the Quake II game server, tel-

net, and various utilities.

1.1 Contributions

The contributions of this paper are:

1. A case for isolation kernels, an OS structure for isolating untrusted software services;

2. A set of design principles for isolation kernels, arguing for a VMM-like structure, but with strategic modifications to the virtual architecture for scalability, performance, and simplicity;

3. The design, implementation, and evaluation of the Denali isolation kernel, focusing on the challenges of scale, and demonstrating that Denali can scale to over 10,000 VMs on commodity hardware.

The rest of this paper is organized as follows. In Section 2, we describe the various classes of applications we hope to enable, and derive design principles of isolation kernels. Section 3 discusses the design and implementation of the Denali isolation kernel. In Section 4, we evaluate our implementation, focusing on issues of scale. We compare Denali to related work in Section 5, and we conclude in Section 6.

2 The Case for Isolation Kernels

Many applications and services would benefit from the ability to push untrusted code into the Internet infrastructure. We outline some of these below, and use them to motivate the properties required by an isolation kernel.

Supporting dynamic content in content delivery systems: a progression of content delivery systems has been introduced in recent years, including CDNs, proxy caches [34], and peer-to-peer networks [30]. All suffer from the limitation that only static content is supported, whereas a large and increasing fraction of content is dynamically generated [34]. Dynamic content distribution requires the ability to execute and isolate untrusted content generation code.

Pushing Internet services into virtual hosting infrastructure: a "virtual" hosting center would allow new Internet services to be uploaded into managed data centers. In addition to supporting commercial services, we believe virtual hosting centers would encourage the emergence of a grassroots development community for Internet services, similar to the shareware community that exists for desktop applications.

Internet measurement and experimentation infrastructure: NIMI [25] and CAIRN [2] have sought to deploy wide-area testbeds to support network measurement research. Recent projects such as Chord [30] would benefit from the ability to deploy research prototypes at scale across the Internet. Whether for measurement or prototyping, the infrastructure must be able to multiplex and isolate mutually distrusting experiments.

Mobile code: deploying mobile code in routers and servers has been proposed by both active networks and mobile agent systems [19].

All of these services and applications share several properties. For the sake of cost-efficiency, multiple services will need to be multiplexed on shared infrastructure. As a result, software infrastructure must exist to isolate multiplexed services from each other: a service must not be able to corrupt another service or the underlying protection system. Additionally, performance isolation is required to bound each service's resource consumption. Finally, the degree of information sharing between these multiplexed services will be small, or entirely non-existent. Because of this, it is reasonable to strengthen isolation at the cost of high sharing overhead.

As we will argue in detail, no existing software system has the correct set of properties to support this emerging class of Internet services. Existing software protection systems (including operating systems, language-based protection techniques, and virtual machine monitors) suffer from some combination of security vulnerabilities, complexity, insufficient scalability, poor performance, or resource management difficulties. We believe that a new software architecture called an *isolation kernel* is required to address the challenges of hosting untrusted services.

2.1 Isolation Kernel Design Principles

An isolation kernel is a small-kernel operating system architecture targeted at hosting multiple untrusted applications that require little data sharing. We have formulated four principles that govern the design of isolation kernels.

1. Expose low-level resources rather than high-level abstractions. In theory, one might hope to achieve isolation on a conventional OS by confining each untrusted service to its own process (or process group). However, OSs have proven ineffective at containing insecure code, let alone untrusted or malicious services. An OS exposes high-level abstractions, such as files and sockets, as opposed to low-level resources such as disk blocks and network packets. High-level abstractions entail significant complexity and typically have a wide API, violating the security principle of economy of mechanism [29]. They also invite "layer below" attacks, in which an attacker gains unauthorized access to a resource by requesting it below the layer of enforcement [18].

An isolation kernel exposes hardware-level resources, displacing the burden of implementing operating systems abstractions to user-level code. In this respect, an isolation kernel resembles other "small

kernel" architectures such as microkernels [1], virtual machine monitors [6], and Exokernels [20]. Although small kernel architectures were once viewed as prohibitively inefficient, modern hardware improvements have made performance less of a concern.

2. Prevent direct sharing by exposing only private, virtualized namespaces. Conventional OSs facilitate protected data sharing between users and applications by exposing global namespaces, such as file systems and shared memory regions. The presence of these sharing mechanisms introduces the problem of specifying a complex access control policy to protect these globally exposed resources.

Little direct sharing is needed across Internet services, and therefore an isolation kernel should prevent direct sharing by confining each application to a private namespace. Memory pages, disk blocks, and all other resources should be virtualized, eliminating the need for a complex access control policy: the only sharing allowed is through the virtual network.

Both principles 1 and 2 are required to achieve strong isolation. For example, the UNIX `chroot` command discourages direct sharing by confining applications to a private file system name space. However, because `chroot` is built on top of the file system abstraction, it has been compromised by a layer-below attack in which the attacker uses a cached file descriptor to subvert file system access control.

Although our discussion has focused on security isolation, high-level abstractions and direct sharing also reduce performance isolation. High-level abstractions create contention points where applications compete for resources and synchronization primitives. This leads to the effect of "crosstalk" [23], where application resource management decisions interfere with each other. The presence of data sharing leads to hidden shared resources like the file system buffer cache, which complicate precise resource accounting.

3. Zipf's Law implies the need for scale. An isolation kernel must be designed to scale up to a large number of services. For example, to support dynamic content in web caches and CDNs, each cache or CDN node will need to store content from hundreds (if not thousands) of dynamic web sites. Similarly, a wide-area research testbed to simulate systems such as peer-to-peer content sharing applications must scale to millions of simulated nodes. A testbed with thousands of contributing sites would need to support thousands of virtual nodes per site.

Studies of web documents, DNS names, and other network services show that popularity tends to be driven by Zipf distributions [5]. Accordingly, we anticipate that isolation kernels must be able to handle Zipf workloads. Zipf distributions have two defining traits: most requests go to a small set of popular services, but a significant fraction of requests go to a large set of unpopular services. Unpopular services are accessed infrequently, reinforcing the need to multiplex many services on a single machine.

To scale, an isolation kernel must employ techniques to minimize the memory footprint of each service, including metadata maintained by the kernel. Since the set of all unpopular services won't fit in memory, the kernel must treat memory as a cache of popular services, swapping inactive services to disk. Zipf distributions have a poor cache hit rate [5], implying that we need rapid swapping to reduce the cache miss penalty of touching disk.

4. Modify the virtualized architecture for simplicity, scale, and performance. Virtual machine monitors (VMMs), such as Disco [6] and VM/370 [9], adhere to our first two principles. These systems also strive to support legacy OSs by precisely emulating the underlying hardware architecture. In our view, the two goals of isolation and hardware emulation are orthogonal. Isolation kernels decouple these goals by allowing the virtual architecture to deviate from the underlying physical architecture. By so doing, we can enhance properties such as performance, simplicity, and scalability, while achieving the strong isolation that VMMs provide.

The drawback of this approach is that it gives up support for unmodified legacy operating systems. We have chosen to focus on the systems issues of scalability and performance rather than backwards compatibility for legacy OSs. However, we are currently implementing a port of the Linux operating system to the Denali virtual architecture; this port is still work in progress.

3 The Denali Isolation Kernel

The Denali isolation kernel embodies all of the principles described in the previous section of this paper. Architecturally, Denali is a thin software layer that runs directly on x86 hardware. Denali exposes a virtual machine (VM) architecture that is based on the x86 hardware, and supports the secure multiplexing of many VMs on an underlying physical machine. Each VM can run its own "guest" operating system and applications (Figure 1).

This section of the paper presents the design of the Denali virtual architecture, and the implementation of an isolation kernel to support it. We also describe the *Ilwaco* guest OS, which is tailored for building Internet services that execute on the Denali virtual architecture.

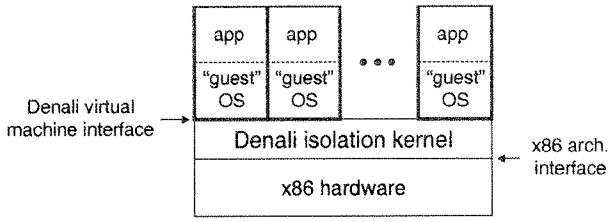


Figure 1: **The Denali architecture:** the Denali isolation kernel is a thin software layer that exposes a virtual machine abstraction that is based on the underlying x86 architecture.

3.1 The Denali Virtual Architecture

The Denali virtual architecture consists of an instruction set, a memory architecture, and an I/O architecture (including an interrupt model). We describe each of these components in turn.

3.1.1 ISA

The Denali virtual instruction set was designed for both performance and simplicity. The ISA primarily consists of a subset of the x86 instruction set, so that most virtual instructions execute directly on the physical processor. The x86 ISA is not strictly virtualizable, as it contains instructions that behave differently in user mode and kernel mode [17, 27]; x86 virtual machine monitors must use a combination of complex binary rewriting and memory protection techniques to virtualize these instructions. Since Denali is not designed to support legacy OSs, our virtual architecture simply defines these instructions to have ambiguous semantics. If a VM executes one of these instructions, at worst the VM could harm itself. In practice, they are rarely used; most deal with legacy architecture features like segmentation, and none are emitted by C compilers such as gcc (unless they appear in inlined assembly fragments).

Denali defines two purely virtual instructions. The first is an "idle-with-timeout" instruction that helps VMs avoid wasting their share of the physical CPU by executing OS idle loops. The idle-with-timeout instruction lets a VM halt its virtual CPU for either a bounded amount of physical time, or until an interrupt arrives for the VM.[1] The second purely virtual instruction simply allows a virtual machine to terminate its own execution.

Denali adds several virtual registers to the x86 register file, to expose system information such as CPU speed, the size of memory, and the current system time. Virtual registers also provide a lightweight communication mechanism between virtual machines and the kernel. For example, we implemented Denali's interrupt-enabled flag as a virtual register.

[1] Denali's idle instruction is similar to the x86 halt instruction, which is executed to put the system into a low-power state during idle periods. Denali's timeout feature allows for fine-grained CPU sharing.

3.1.2 Memory Architecture

Each Denali VM is given its own (virtualized) physical 32-bit address space. A VM may only access a subset of this 32-bit address space, the size and range of which is chosen by the isolation kernel when the VM is instantiated. The kernel itself is mapped into a portion of the address space that the VM cannot access; because of this, we can avoid physical TLB flushes on VM/VMM crossings.

By default, a VM cannot virtualize its own (virtualized) physical address space: in other words, by default, there is no virtual MMU. In this configuration, a VM's OS shares its address space with applications, similar to a libOS in Exokernel [20]. Exposing a single address space to each VM improves performance, by avoiding TLB flushes during context switches between applications in the same VM [31].

We have recently added support for an optional, virtual MMU to Denali; this virtual MMU looks nothing like the underlying x86-based physical MMU, but instead is modeled after a simple software-loaded TLB, similar to those of modern RISC architectures. A software-loaded TLB has the advantage that the VM itself gets to define its own page-table structure, and the software TLB interface between the VMM and the VM is substantially simpler than the more complex page table interface mandated by the x86 hardware-loaded TLB architecture.

3.1.3 I/O Devices and Interrupt Model

Denali exposes several virtual I/O devices, including an Ethernet NIC, a disk, a keyboard, a console, and a timer. Denali's virtual devices have similar functionality to their physical counterparts, but they expose a simpler interface. Physical devices often have "chatty" interfaces, requiring many programmed I/O instructions per operation. VMMs that emulate real hardware devices suffer high overhead, since each PIO must be emulated [31]. Another benefit of simplification is portability: our virtual device interfaces are independent of the underlying physical devices.

Denali has chosen to omit many x86 architectural features. Virtual devices have been simplified to begin in a well-known, pre-initialized state when a VM boots. This simplifies both the Denali isolation kernel and guest OSs by eliminating the need to probe virtual devices on boot. Denali also does not expose the BIOS[2] or segmentation hardware, because these features are rarely used outside of system boot.

[2] The BIOS is also involved in power management; Denali does not expose this to VMs.

Denali exposes virtual interrupts to VMs, much in the same way that the physical x86 architecture exposes real interrupts to the host processor. Virtual interrupts are typically triggered by physical interrupts, such as when an Ethernet packet arrives that is destined for a particular VM. However, not all physical interrupts cause virtual interrupts; for example, a packet may arrive that is not destined for any of the running VMs, in which case the isolation kernel simply drops the packet without raising a virtual interrupt.

Denali's interrupt dispatch model differs significantly from the underlying x86 hardware to better support the multiplexing of many virtual machines. As the number of simultaneously running VMs grows, it becomes increasingly unlikely that the VM which is the ultimate recipient of a physical interrupt is executing when the interrupt is raised. In some cases, the target VM could even be swapped out to disk. Rather than preserving the immediate interrupt semantics of x86, Denali delays and batches interrupts destined for non-running VMs. A VM receives pending interrupts once it begins its normal scheduler quantum, and if multiple interrupts are pending for a VM, all interrupts are delivered in a single VMM/VM crossing. This is similar to the Mach 3.0 user-level I/O interface [15].

Denali's asynchronous, batched interrupt model changes the semantics of timing-related interrupts. For example, a conventional timer interrupt implies that a fixed-size time interval has just passed. In Denali, a virtual timer interrupt implies that some amount of physical time has passed, but the duration may depend on how many other VMs are contending for the CPU. As a result, the interpretation of timer interrupts in the implementation of guest OS software timers must be altered.

3.2 Isolation Kernel Implementation

The Denali isolation kernel runs directly on x86 hardware. The core of the kernel, including multiprogramming, paging, and virtual device emulation, was implemented from scratch; we used the Flux OSKit [14] for device drivers and other hardware support routines, and some support libraries such as libc.

The isolation kernel serves two roles: it implements the Denali virtual architecture, and it multiplexes physical resources across competing VMs. We have maintained a strict separation between resource management policy and mechanism, so that we could implement different policies without affecting other aspects of the isolation kernel.

3.2.1 CPU Virtualization

Denali uses standard multiprogramming techniques to multiplex the CPU across VMs. The isolation kernel maintains a per-VM thread structure, which contains a kernel stack, space for the register file, and the thread status. The policy for multiplexing the CPU is split into two components: a *gatekeeper policy* enforces admission control by choosing a subset of active machines to admit into the system; the rest are swapped to disk, as we will describe later. The *scheduler policy* controls context switching among the set of admitted machines.

The gatekeeper admits machines in FIFO order as long as there are a minimum number of physical backing pages available. The scheduler uses round-robin scheduling among the set of admitted machines. These policies were chosen because they are simple and starvation-free. When a VM issues an idle-with-timeout instruction, it is removed from scheduler consideration until its timer fires, or a virtual interrupt arrives. As compensation for idling, a VM receives higher scheduler priority for its next quantum.

Virtual registers are stored in a page at the beginning of a VM's (virtual) physical address space. This page is shared between the VM and the isolation kernel, avoiding the overhead of kernel traps for register modifications. In other respects, the virtual registers behave like normal memory (for example, they can be paged out to disk).

Because Denali's ISA is based on x86, we can use existing x86 compilers and linkers when authoring OS or application code to run in a Denali VM. In particular, we have been primarily using the `gcc` C compiler and the `ld` linker on Linux, although we did need to change the link map used by `ld` to take Denali's memory architecture into account.

3.2.2 Memory Management

The (virtual) physical address space exposed to a VM has two components: a portion that is accessible to the VM, and a protected portion accessible only to the isolation kernel. Each VM also has a *swap region* allocated on behalf of it by the isolation kernel; this swap region is striped across local disks. The swap region is used by the isolation kernel to swap or page out portions of the VM's address space. Swap regions are statically allocated at VM creation time, and are large enough to hold the entire VM-visible address space. Static allocation drastically reduces the amount of bookkeeping metadata in the isolation kernel: each swap region is completely described by 20 bytes of kernel memory. Static allocation wastes disk capacity in return for performance and scalabil-

ity, but the decreasing cost of storage capacity makes this trade-off worthwhile.

The isolation kernel is pinned in physical memory, but VMs are paged in on demand. Upon taking a page fault, the kernel verifies that the faulting VM hasn't accessed an illegal virtual address, allocates necessary page tables, and initiates a read from the VM's swap region.

The system periodically redistributes physical memory from inactive VMs to active VMs. We use the WSClock [7] page replacement algorithm, which attempts to maintain each VM's working set in memory by maintaining a virtual time stamp along with a clock reference bit. This helps reduce thrashing, and is more fair to machines that experience heavy paging (such as reactivated machines that are entirely swapped out). To encourage good disk locality, all memory buffers for a given VM are clustered together in the clock circular list.

For the remainder of this paper, we configured the system to expose only 16MB of accessible (virtual) physical address space to each VM. This models the challenging scenario of having many small services multiplexed on the same hardware. Because virtual MMUs are such a recent addition and are still being performance optimized, we did not turn on virtual MMU support for the experiments presented in Section 4. Although we hope that enabling virtual MMU support will not affect our overall performance results, we have not yet demonstrated this.

3.2.3 I/O Devices and Interrupt Model

Denali emulates a switched Ethernet LAN connecting all VMs. Each VM is assigned a virtual Ethernet NIC; from the perspective of external physical hosts, it appears as though each VM has its own physical Ethernet card. VMs interact with the virtual NIC using standard programmed I/O instructions, although the interface to the virtual NIC is drastically simpler than physical NICs, consisting only of a packet send and a packet receive operation. On the reception path, the isolation kernel emulates an Ethernet switch by demultiplexing incoming packets into a receive queue for the destination virtual machine. VMs can only process packets during their scheduler quantum, effectively implementing a form of lazy-receiver processing [11]. On the transmit path, the kernel maintains a per-machine queue of outbound packets which a VM can fill during its scheduler quantum. These transmit queues are drained according to a packet scheduler policy; Denali currently processes packets in round-robin order from the set of actively sending VMs.

Denali provides virtual disk support to VMs. The isolation kernel contains a simple file system in which it manages persistent, fixed-sized virtual disks. When a VM is instantiated, the isolation kernel exposes a set of virtual disks to it; the VM can initiate asynchronous reads and writes of 4KB blocks from the disks to which it has been given access. Because virtual disks exist independently of virtual machines, Denali trivially supports optimizations such as the read-only sharing of virtual disks across VMs. For example, if multiple VMs all use the same kernel boot image, that boot image can be stored on a single read-only virtual disk and shared by the VMs.

Denali also emulates a keyboard, console, and timer devices. These virtual devices do not differ significantly from physical hardware devices, and we do not describe them further.

Denali's batched interrupt model is implemented by maintaining a bitmask of pending interrupts for each VM. When a virtual interrupt arrives, the kernel posts the interrupt to the bitmask, activates the VM if it is idle, and clears any pending timeouts. When a VM begins its next quantum, the kernel uploads the bitmask to a virtual register, and transfers control to an interrupt handler. A VM can disable virtual interrupts by setting a virtual register value; VMs can never directly disable physical interrupts.

3.2.4 Supervisor Virtual Machine

Denali gives special privileges to a *supervisor VM*, including the ability to create and destroy other VMs. Because complexity is a source of security vulnerabilities, wherever possible we have displaced complexity from the isolation kernel to the supervisor VM. For example, the isolation kernel does not have a network stack: if a remote VM image needs to be downloaded for execution, this is done by the supervisor VM. Similarly, the supervisor VM keeps track of the association between virtual disks and VMs, and is responsible for initializing or loading initial disk images into virtual disks. The supervisor VM can be accessed via the console, or through a simple telnet interface. In a production system, the security of the supervisor VM should be enhanced by using a secure login protocol such as ssh.

3.3 Ilwaco Guest OS

Although the Denali virtual machine interface is functionally complete, it is not a convenient interface for developing applications. Accordingly, we have developed the Ilwaco guest operating system which presents customary high-level abstractions. Ilwaco is implemented as a library, in much the same fashion as a Exokernel libOS. Applications directly link against the OS; there is no hardware-enforced protection boundary.

Ilwaco contains the Alpine user-level TCP stack [12], a port of the FreeBSD 3.3 stack. We modified Alpine to utilize Denali's virtual interrupt and timer mechanisms, and linked the stack against a device driver for the Denali virtual Ethernet NIC.

Ilwaco contains a thread package that supports typical thread primitives, locks, and condition variables. If there are no runnable threads, the thread scheduler invokes the idle-with-timeout virtual instruction to yield the CPU. Ilwaco also contains a subset of libc, including basic console I/O, string routines, pseudo-random number generation, and memory management. Most of these routines were ported from OSKit libraries; some functions needed to be modified to interact with Denali's virtual hardware. For example, `malloc` reads the size of (virtual) physical memory from a virtual register.

4 Evaluation

This section presents a quantitative evaluation of the Denali isolation kernel. We ran microbenchmarks to (1) quantify the performance of Denali's primitive operations, (2) validate our claim that our virtual architecture modifications result in enhanced scale, performance, and simplicity, and (3) characterize how our system performs at scale, and why. As previously mentioned, none of these experiments were run with the virtual MMU enabled. Additionally, in all experiments, our VMs ran with their data in virtual core, and as such, they did not exercise the virtual disks.[3]

In our experiments, Denali ran on a 1700MHz Pentium 4 with 256KB of L2 cache, 1GB of RAM, an Intel PRO/1000 PCI gigabit Ethernet card connected to an Intel 470T Ethernet switch, and three 80 GB 7200 RPM Maxtor DiamondMax Plus IDE drives with 2 MB of buffering each. For any experiment involving the network, we used a 1500 byte MTU. To generate workloads for network benchmarks, we used a mixture of 1700MHz Pentium 4 and 930MHz Pentium III machines.

4.1 Basic System Performance

To characterize Denali's performance, we measured the context switching overhead between VMs, and the swap disk subsystem performance. We also characterized virtualization overhead by analyzing packet dispatch latency, and by comparing the application-level TCP and HTTP throughput of Denali with that of BSD.

[3]Of course, our scaling experiments do stress the swapping functionality in the isolation kernel itself.

4.1.1 VM Context Switching Overhead

To measure context-switching overhead, we considered two workloads: a "worst-case" that cycles through a large memory buffer between switches, and a "best-case" that does not touch memory between switches. For the worst-case workload, context switch time starts at 3.9 μs for a single virtual machine, and increases to 9 μs for two or more VMs. For the best-case workload, the context switch time starts at 1.4 μs for a single virtual machine, and it increases slightly as the number of VMs increases; the slight increases coincide with the points at which the capacity of the L1 and L2 caches become exhausted. These results are commensurate with process context switching overheads in modern OSs.

4.1.2 Swap Disk Microbenchmarks

Denali stripes VM swap regions across physical disks. To better understand factors that influence swap performance at scale, we benchmarked Denali's disk latency and throughput for up to three attached physical disks. The results are presented in Table 1; all measured throughputs and latencies were observed to be limited by the performance of the physical disks, but not the Denali isolation kernel. For three disks, a shared PCI bus became the bottleneck, limiting sequential throughput.

4.1.3 Packet Dispatch Latency

Figure 2 shows packet processing costs for application-level UDP packets, for both 100 and 1400 byte packets. A transmitted packet first traverses the Alpine TCP/IP stack and then is processed by the guest OS's Ethernet device driver. This driver signals the virtual NIC using a PIO, resulting in a trap to the isolation kernel. Inside the kernel, the virtual NIC implementation copies the packet out of the guest OS into a transmit FIFO. Once the network scheduler has decided to transmit the packet, the physical device driver is invoked. Packet reception essentially follows the same path in reverse.

On the transmission path, our measurement ends when the physical device driver signals to the NIC that a new packet is ready for transmission; packet transmission costs therefore do not include the time it takes the packet to be DMA'ed into the NIC, the time it takes the NIC to transmit the packet on the wire, or the interrupt that the NIC generates to indicate that the packet has been transmitted successfully. On the reception path, our measurement starts when a physical interrupt arrives from the NIC; packet reception costs therefore include interrupt processing and interacting with the PIC.

The physical device driver and VM's TCP/IP stack incur significantly more cost than the isolation

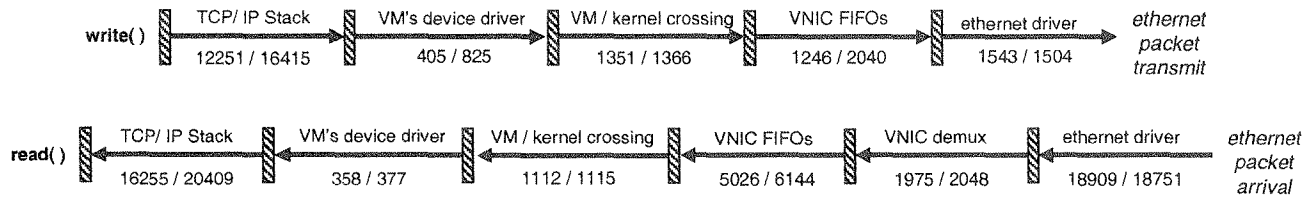


Figure 2: **Packet processing overhead:** these two timelines illustrate the cost (in cycles) of processing a packet, broken down across various functional stages, for both packet reception and packet transmission. Each pair of numbers represents the number of cycles executed in that stage for 100 byte and 1400 byte packets, respectively.

	latency	random throughput	sequential throughput
1 disk	7.1 / 5.9	2.20 / 2.66	38.2 / 31.5
2 disks	7.0 / 5.8	4.45 / 5.41	75.6 / 63.5
3 disks	7.0 / 5.8	6.71 / 8.10	91.3 / 67.1

Table 1: **Swap disk microbenchmarks:** latency (ms), random throughput (MB/s), and sequential throughput (MB/s) versus the number of disks. Numbers separated by a slash are for reads and writes, respectively.

kernel, confirming that the cost of network virtualization is low. The physical driver consumes 43.3% and 38.4% of the total packet reception costs for small and large packets, respectively. Much of this cost is due to the Flux OSKit's interaction with the 8259A PIC; we plan on modifying the OSKit to use the more efficient APIC in the future. The TCP stack consumes 37.3% and 41.8% of a small and large packet processing time, respectively.

The transmit path incurs two packet copies and one VM/kernel boundary crossing; it may be possible to eliminate these copies using copy-on-write techniques. The receive path incurs the cost of a packet copy, a buffer deallocation in the kernel, and a VM/kernel crossing. The buffer deallocation procedure attempts to coalesce memory back into a global pool and is therefore fairly costly; with additional optimization, we believe we could eliminate this.

4.1.4 TCP and HTTP Throughput

As a second measurement of networking performance on Denali, we compared the TCP-level throughput of BSD and a Denali VM running Ilwaco. To do this, we compiled a benchmark application on both Denali and BSD, and had each application run a TCP throughput test to a remote machine. We configured the TCP stacks in all machines to use large socket buffers. The BSD-Linux connection was able to attain a maximum throughput of 607 Mb/s, while Denali-Linux achieved 569 Mb/s, a difference of 5%.

As further evaluation, we measured the performance of a single web server VM running on Denali. Our home-grown web server serves static content out of (virtual) physical memory. For comparison, we

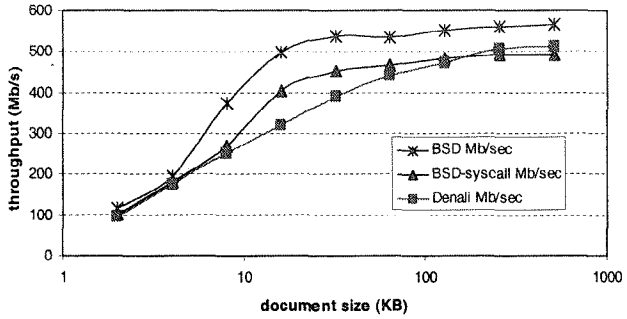


Figure 3: **Comparing web server performance on Denali and BSD:** performance is comparable, confirming that virtualization overhead is low. The "BSD-syscall" line corresponds to a version of the BSD web server in which an extra system call was added per packet, to approximate user-level packet delivery in Denali.

ported our web server to BSD by compiling and linking the unmodified source code against a BSD library implementation of the Ilwaco system call API. Figure 3 shows the results.

Denali's application-level performance closely tracks that of BSD, although for medium-sized documents (50-100KB), BSD outperforms Denali by up to 40%. This difference in performance is due to the fact that Denali's TCP/IP stack runs at the user-level, implying that all network packets must cross the user/kernel boundary. In contrast, in BSD, most packets are handled by the kernel, and only data destined for the application crosses the user-kernel boundary. A countervailing force is system calls: in Denali, system calls are handled within the user-level by the Ilwaco guest OS; in BSD, system calls must cross the user-kernel boundary.

For small documents, there are about as many system calls per connection in BSD (`accept`, `reads`, `writes`, and `close`) as there are user/kernel packet crossings in Denali. For large documents, the system bottleneck becomes the Intel PRO/1000 Ethernet card. Therefore, it is only for medium-sized documents that the packet delivery to the user-level networking stack in Denali induces a noticeable penalty; we confirmed this effect by adding a system call per packet to the BSD web server, observing that with this additional overhead, the BSD performance

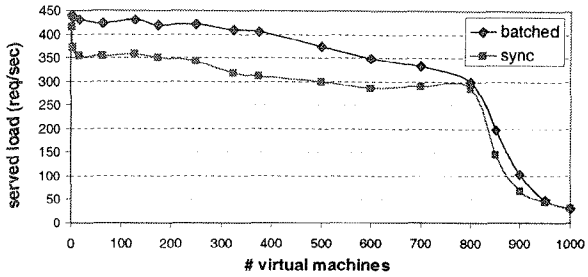


Figure 4: **Benefits of batched, asynchronous interrupts:** Denali's interrupt model leads to a 30% performance improvement in the web server when compared to synchronous interrupts, but at large scale (over 800 VMs), paging costs dominate.

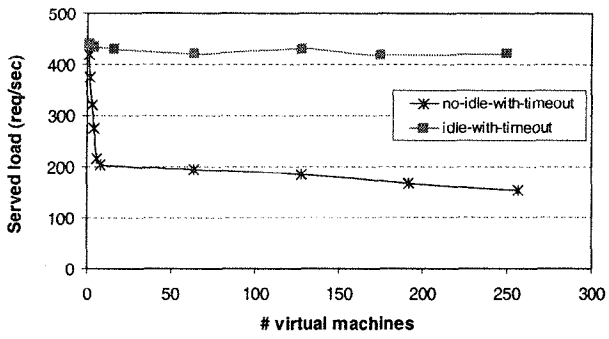


Figure 5: **Idle-with-timeout benefits:** idle-with-timeout leads to higher performance at scale, when compared to an idle instruction with no timeout feature.

closely matched that of Denali even for medium-sized documents (Figure 3).

4.2 Scale, Performance, and Simplicity and the Denali Virtual Architecture

The virtual architecture exposed by Denali was designed to enhance scalability, performance, and simplicity. In this section, we provide quantitative evidence to back these claims. Specifically, we demonstrate that batched asynchronous interrupts have performance and scalability benefits, that Denali's idle-with-timeout instruction is crucial for scalability, that Denali's simplified virtual NIC has performance advantages over an emulated real NIC, and that the source code complexity of Denali is substantially less than that of even a minimal Linux kernel.

4.2.1 Batched, Asynchronous Interrupts

Denali utilizes a batched, asynchronous model for virtual interrupt delivery. In Figure 4, we quantify the performance gain of Denali's batched, asynchronous interrupt model, relative to the performance of synchronous interrupts. To gather the synchronous interrupt data, we modified Denali's scheduler to immediately context switch into a VM when an interrupt arrives for it. We then measured the aggregate performance of our web server application serving a 100KB document, as a function of the number of simultaneously running VMs. For a small number of VMs, there was no apparent benefit, but up to a 30% gain was achieved with batched interrupts for up to 800 VMs. Most of this gain is attributable to a reduction in context switching frequency (and therefore overhead). For a very large number of VMs (over 800), performance was dominated by the costs of the isolation kernel paging VMs in and out of core.

4.2.2 Idle-with-timeout Instruction

To measure the benefit of the idle-with-timeout virtual instruction, we compared the performance of web server VMs serving 100KB documents across in two scenarios. In the first scenario, the VMs exploited the timeout feature: a guest OS with no schedulable threads invokes the idle-with-timeout instruction with a timeout value set to the smallest pending TCP retransmission timer. In the second scenario, VMs did not use the timeout feature, idling only when there were no schedulable threads and no pending TCP retransmission timers.

The performance difference was substantial: Figure 5 shows that as the number of VMs scales up, overall performance drops by more than a factor of two without the timeout feature. The precipitous drop in performance for small numbers of VMs happens because the entire offered load is focused on those few VMs, ensuring that all of them have active connections; an active connection means that retransmission timers are likely pending, preventing the VM from idling. As the number of VMs grows, the same aggregate workload is spread over more VMs, meaning that any individual VM is less likely to have active connections preventing it from idling. This results in an easing off of additional overhead as the system scales.

In general, a timeout serves as a hint to the isolation kernel; without this hint, the kernel cannot determine whether any software timers inside a VM are pending, and hence will not know when to reschedule the VM. As a result, without the timeout feature, a VM has no choice but to spin inside its idle loop to ensure that any pending software timers fire.

4.2.3 Simplified Virtual Ethernet

Denali's virtual Ethernet has been streamlined for simplicity and performance. Real hardware network adapters often require multiple programmed I/O instructions to transmit or receive a single packet. For example, the Linux pcnet32 driver used by VMWare workstation [31] issues 10 PIOs to receive a packet and 12 PIOs to transmit a packet.

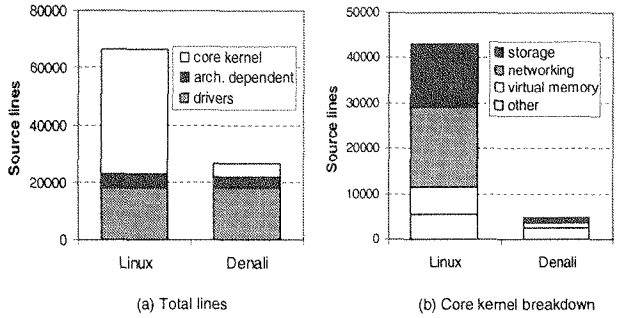


Figure 6: **Source code complexity:** number of source lines in Linux 2.4.16 and Denali. Denali is roughly half the size of Linux in total source lines. Denali's core kernel (without device drivers and platform-dependent code) is an order-of-magnitude smaller than Linux.

VMMs which support unmodified legacy OSs must trap and emulate these PIOs, resulting in additional overhead. By contrast, Denali's virtual Ethernet requires only a single PIO to send or receive a packet.

To estimate the benefit of Denali's simple virtual Ethernet, we modified the guest OS device driver to perform as many PIOs as the pcnet32 driver. Doing so increased the packet reception cost by 18,381 cycles (10.9 ms) and the packet transmission cost by 22,955 cycles (13.7 ms). This increases the overhead of receiving a single 100-byte UDP packet by 42%.

4.2.4 Source Code Complexity

As a final measure of the impact of Denali's virtual architecture, we quantify the size of the Denali source tree relative to Linux. This comparison therefore gives an indication of how complex it is to implement an isolation kernel that exposes Denali's architecture, as compared to the complexity of implementing an OS that exports high-level abstractions. Code size is important because it serves as an indication of the size of the trusted computing base, and it also impacts how easily the system can be maintained, modified, and debugged over time.

We compared Denali against Linux 2.4.16. For fairness of comparison, we choose a subset of Linux files that comprise a "bare-bones" kernel: no module support, no SMP support, no power management, only the ext2 file system, limited driver support, and so on. We use semicolon count as the metric of source lines, to account for different coding conventions.

Denali contains 26,634 source lines, while Linux has 66,326 source lines (Figure 6a). Only a small fraction (18%) of the Denali source is consumed by the "core kernel"; the remainder is dedicated to device drivers and architecture-dependent routines. Although drivers are known to be more buggy than core kernel code [8], the drivers used by Denali and "bare-bones" Linux consist of mature source code that has not changed substantially over time, e.g., the IDE driver, terminal support, and PCI bus probing.

In Figure 6b, we present a breakdown of the core kernel sizes of Denali and Linux. The Linux core kernel is an order-of-magnitude larger than Denali. The majority of the difference is attributable to Linux's implementation of stable storage (the ext2 file system) and networking (TCP/IP) abstractions. By deferring the implementation of complex abstractions to guest operating systems, Denali realizes a substantial reduction in core kernel source tree size.

4.3 Denali at Scale

In this section, we characterize Denali's performance at scale. We first analyze two scaling bottlenecks, which we removed before performing application-level scaling experiments. We then analyze two applications with fairly different performance requirements and characteristics: a web server and the Quake II game server.

4.3.1 Scaling Bottlenecks

The number of virtual machines to which our isolation kernel can scale is limited by two factors: per-machine metadata maintained by the kernel when a VM has been completely paged out, and the working set size of active VMs.

Per-VM kernel metadata: To minimize the amount of metadata the isolation kernel must maintain for each paged-out VM, wherever possible we allocate kernel resources on demand, rather than statically on VM creation. For example, page tables and packet buffers are not allocated to inactive VMs. Table 2 breaks down the memory dedicated to each VM in the system. Each VM requires 8,472 bytes, of which 97% are dedicated to a kernel thread stack. Although we could use continuations [10] to bundle up the kernel stack after paging a VM out, per-VM kernel stacks have simplified our implementation. Given the growing size of physical memory, we feel this is an acceptable tradeoff: supporting 10,000 VMs requires 81 MB of kernel metadata, which is less than 4% of memory on a machine with 2GB of RAM.

VM working set size: The kernel cannot control the size of a VM's working set, and the kernel's paging mechanism may cause a VM to perform poorly if the VM scatters small memory objects across its pages. One instance where memory locality is especially important is the management of the mbuf packet buffer pool inside the BSD TCP/IP stack of our Ilwaco guest OS. Initially, mbufs are allocated from a large contiguous byte array; this "low entropy" initial state means that a request that touches a small number of mbufs would only touch a single page in memory. After many allocations and

Component	Size (bytes)
thread stack	8192
register file	24
swap region metadata	20
paging metadata	40
virtual Ethernet structure	80
pending alarms	8
VM boot command line	64
other	72
Total	**8472**

Table 2: **Per-VM kernel metadata:** this table describes the residual kernel footprint of each VM, assuming the VM has been swapped out.

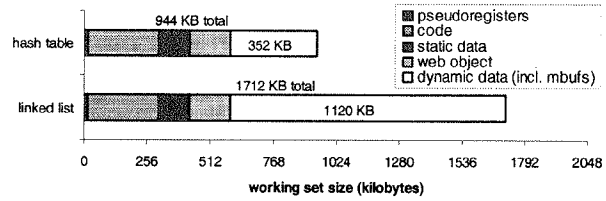
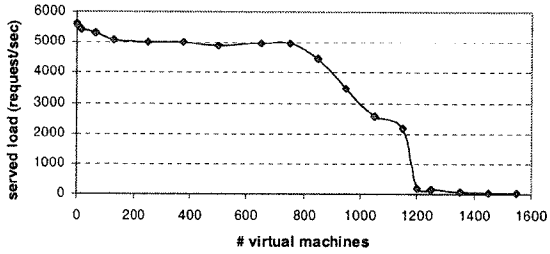


Figure 7: **Mbuf entropy and memory footprint:** eliminating mbuf entropy with a hash table can halve memory footprint.

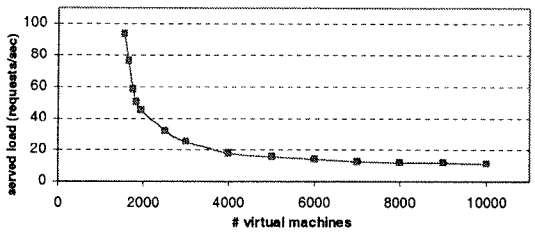
deallocations from the mbuf pool, the default BSD implementation of the mbuf pool scatters back-to-back mbuf allocations across pages: in the worst case, as many pages are necessary as referenced mbufs, increasing the memory footprint of a VM.

We have observed the effects of mbuf entropy in practice, especially if a VM is subjected to a burst of high load. Figure 7 shows the effect of increasing the offered load on a web server inside a VM. The memory footprint of the VM using the default, linked list BSD implementation of the mbuf pool increases by 83% as the system reaches overload. We improved memory locality by replacing the linked list with a hash table that hashes mbufs to buckets based on the memory address of the mbufs; by allocating from hash buckets, the number of memory pages used is reduced. With this improvement, the VM's memory footprint remained constant across all offered loads. The savings in memory footprint resulted in nearly a factor of two performance improvement for large numbers of concurrent web server VMs.

More generally, the mbuf entropy problem is indicative of two larger issues inherent in the design of a scalable isolation kernel. First, the paging behavior of guest operating systems is a crucial component of overall performance; most existing OSs are pinned in memory and have little regard for memory locality. Second, memory allocation and deallocation routines (e.g., garbage collection) may need to be re-examined to promote memory locality; existing work



(a) small document; small # virtual machines



(b) small document; large # virtual machines

Figure 8: **In-core vs. out-of-core:** (a) shows aggregate performance up to the "cliff" at approximately 1000 VMs; (b) shows aggregate performance beyond the cliff.

on improving paging performance in object-oriented languages could prove useful.

4.3.2 Web server performance

To understand the factors that influence scalability for a throughput-centric workload, we analyzed Denali's performance when running many web server VMs. We found that three factors strongly influenced scalability: disk transfer block size, the popularity distribution of requests across VMs, and the object size transferred by each web server.

To evaluate these factors, we used a modified version of the httperf HTTP measurement tool to generate requests across a parameterizable number of VMs. We modified the tool to generate requests according to a Zipf distribution with parameter α. We present results for repeated requests to a small object of 2,258 bytes (approximately the median web object size). Requests of a larger web object (134,007 bytes) were qualitatively similar.

The performance of Denali at scale falls into two regimes. In the *in-core* regime, all VMs fit in memory, and the system can sustain nearly constant aggregate throughput independent of scale. When the number of active VMs grows to a point that their combined working sets exceed the main memory capacity, the system enters the *disk-bound* regime. Figure 8 demonstrates the sharp performance cliff separating these regimes.

In-core regime: To better understand the performance cliff, we evaluated the effect of two variables: disk block transfer size, and object popularity distribution. Reducing the block size used during

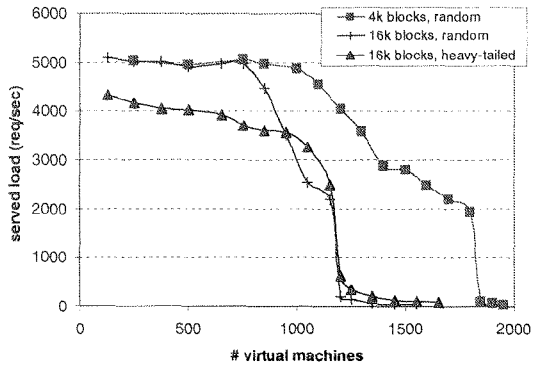


Figure 9: **Block size and popularity distribution:** this graph shows the effect of varying block size and popularity distribution on the "cliff"; the web servers were serving a 2,258 byte document.

paging can improve performance by reducing internal fragmentation, and as a consequence, reducing a VM's in-core footprint. This has the side-effect of delaying the onset of the performance cliff (Figure 9): by using a small block size, we can push the cliff to beyond 1000 VMs.

Disk-bound regime: To illustrate Denali's performance in the disk-bound regime, we examined web server throughput for 4,000 VMs serving the "small" document; the footprint of 4,000 VMs easily exceeds the size of main memory. Once again, we considered the impact of block size and object popularity on system performance.

To explore the effect of heavy-tailed distributions, we fixed the disk block transfer size at 32 kilobytes, and varied the Zipf popularity parameter α. As α increases, the distribution becomes more concentrated on the popular VMs. Unlike the CPU and the network, Denali's paging policy is purely demand driven; as a result, Denali is able to capitalize on the skewed distribution, as shown in Figure 10.

Figure 11 illustrates the effect of increased block size on throughput. As a point of comparison, we include results from a performance model that predicts how much performance our three disk subsystem should support, given microbenchmarks of its read and write throughput, assuming that each VM's working set is read in using random reads and written out using a single sequential write. Denali's performance for random requests tracks the modeled throughput, differing by less than 35% over the range of block sizes considered.

This suggests that Denali is utilizing most of the available raw disk bandwidth, given our choice of paging policy. For heavy-tailed requests, Denali is able to outperform the raw disk bandwidth by caching popular virtual machines in main memory. To improve performance beyond that which we have reported, the random disk reads induced by paging

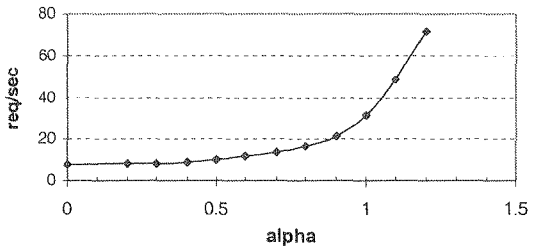


Figure 10: **Out-of-core performance vs. α:** increasingly skewed popularity distributions have better out-of-core performance; this data was gathered for 4,000 VMs serving the small web object, and a block size of 32KB.

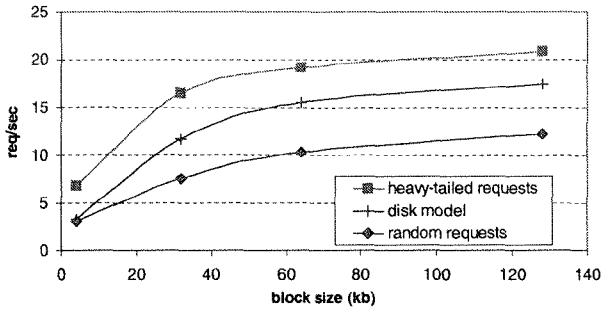


Figure 11: **Out-of-core performance vs. block size:** increased block size leads to increased performance in the out-of-core regime.

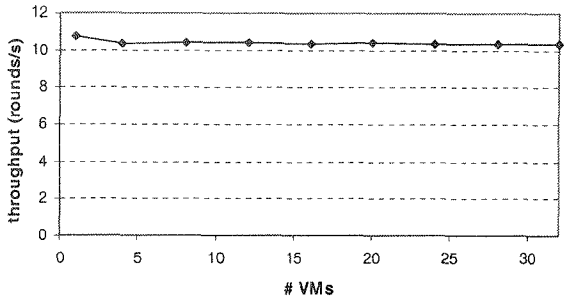
would need to be converted into sequential reads; this could be accomplished by reorganizing the swap disk layout so that the working sets of VMs are laid out sequentially, and swapped in rather than paged in.

4.3.3 Quake II server performance

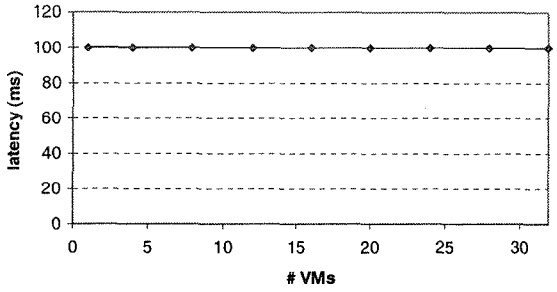
As a second workload to test the scalability of Denali, we ported the GPL'ed Quake II game server to Ilwaco. Quake II is a latency-sensitive multiplayer game. Each server maintains state for a single game session; clients participating in the session send and receive coordinate updates from the server. We use two metrics as a measure of the quality of the game experience: the *latency* between a client sending an update to the server and receiving back a causally dependent update, and the *throughput* of updates sent from the server. Steady latency and throughput are necessary for a smooth, lag-free game experience.

To generate load, we used modified Linux Quake II clients to play back a recorded game session to a server; for each server, we ran a session with four clients. As a test of scalability, we measured the throughput and latency of a Quake server as a function of the number of concurrently active Quake VMs. Figure 12 shows our results.

As we scaled up the number of VMs (and the number of clients generating load), the average throughput and latency of each server VM remained essentially constant. At 32 VMs, we ran out of client



(a) Quake II server throughput (per VM)



(b) Quake II server latency

Figure 12: **Quake II server scaling benchmarks:** even with 32 concurrent active Quake II server VMs, the throughput and latency to each server remained constant. At 32 servers, we ran out of client machines to drive load.

machines to generate load. Even with this degree of multiplexing, both throughput and latency remained constant, suggesting that the clients' game experiences would still be good.

Although the Quake server is latency-sensitive, it is in many ways an ideal application for Denali. The default server configuration self-imposes a delay of approximately 100 ms between update packets, in effect introducing a sizable "latency buffer" that masks queuing and scheduling effects in Denali. Additionally, because the server-side of Quake is much less computationally intensive than the client-side, multiplexing large numbers of servers is quite reasonable.

5 Related Work

We consider related work along two axes: operating system architectures, and techniques for isolating untrusted code.

5.1 OS architectures

The idea of hosting multiple isolated protection contexts on a single machine is not new: Rushby's separation kernel [28] is an instance of this idea. Denali puts these ideas into practice, and explores the systems issues when scaling to a large number of protection domains.

Exokernels [20] eliminate high-level abstractions to enable OS extensibility. Denali differs from Exokernels in its approach to naming: Denali exposes virtual, private name spaces, whereas Exokernels expose the physical names of disk, memory, and network resources. The Exokernel's global namespace allow resources to be shared freely, necessitating complex kernel mechanisms to regulate sharing.

Denali is similar to microkernel operating systems like Mach [1]. Indeed, Denali's VMs could be viewed as single-threaded applications on a low-level microkernel. However, the focus of microkernel research has been to push OS functionality into shared servers, which are themselves susceptible to the problems of high-level abstractions and data sharing. Denali emphasizes scaling to many untrusted applications, which was never an emphasis of microkernels.

Nemesis [23] shares our goal of isolating performance between competing applications. Nemesis adopts a similar approach, pushing most kernel functionality to user-level. Nemesis was not designed to sandbox untrusted code; Nemesis applications share a global file system and a single virtual address space.

The Fluke OS [13] proposes a recursive virtual machine model, in which a parent can re-implement OS functionality on behalf of its children. Like Denali, Fluke exposes private namespaces through its "state-encapsulation" property. The primary motivation for this is to support checkpointing and migration, though the security benefits are alluded to in [22]. Denali exposes a virtual hardware API, whereas Fluke virtualizes at the level of OS API. By virtualizing below abstractions, Denali's kernel is simple, and we avoid layer-below vulnerabilities.

Virtual machine monitors have served as the foundation of several "security kernels" [21]. More recently, the NetTop proposal aims to create secure virtual workstations running on VMWare [24]. Denali differs from these efforts in that we aim to provide scalability as well as isolation. We assume a weaker threat model; for example, we are not concerned with covert channels between VMs.

VMMs like Disco [6] and VM/370 [9] have the goal of supporting legacy systems, and therefore minimize architectural modifications to maintain compatibility. In comparison, isolation kernels rely on virtualization for isolation: backwards compatibility is not their primary goal. As a result, isolation kernels have the freedom to make strategic changes to the exposed virtual architecture for scalability, performance, and simplicity.

5.2 Enforcing isolation

Many projects provide OS support for isolating untrusted code, including system call interposition [16] and restricted execution contexts [32]. These proposals provide *mechanisms* for enforcing

the principle of least privilege. However, expressing an appropriate access control *policy* requires a security expert to reason about access permissions to grant applications; this is a difficult task on modern systems with thousands of files and hundreds of devices. Denali imposes a simple security policy: complete isolation of VMs. This obviates the policy problem, and provides robust isolation for applications with few sharing requirements.

WindowBox [3] confines applications to a virtual desktop, imposing a private namespace for files. Because it is implemented inside a conventional OS, WindowBox's security is limited by high-level abstractions and global namespaces. For example, all applications have access to the Windows registry, which has been involved in many vulnerabilities.

Software VMs (like Java) have been touted as platforms for isolating untrusted code. Experience with these systems has demonstrated a tradeoff between security and flexibility. The Java sandbox was simple and reasonably secure, but lacked the flexibility to construct complex applications. Extensible security architectures [33] allow more flexibility, but reintroduce the problem of expressing an appropriate access control policy. Denali avoids this tradeoff by exposing a raw hardware API, complete with I/O devices, which allows VMs to build up arbitrary abstractions inside a guest OS. In addition, Denali's virtual architecture closely mirrors the underlying physical architecture, avoiding the need for a complex runtime engine or just-in-time compiler.

The problem of performance isolation has been addressed by server and multimedia systems [4, 26, 23]. Resource containers demonstrate that OS abstractions for resource management (processes and threads) are poorly suited to applications' needs. Denali's VMs provide a comparable resource management mechanism. We believe that isolation kernels can provide more robust performance isolation by operating beneath OS abstractions and data sharing. As an example, Reumann et al. conclude that there is no simple way to account for the resources in the file system buffer cache [26].

Finally, numerous commercial and open-source products provide support for virtual hosting, including freeVSD, Apache virtual hosts, the Solaris resource manager, and Ensim's ServerXchange. All work within a conventional OS or application, and therefore cannot provide the same degree of isolation as an isolation kernel. Commercial VMMs provide virtual hosting services, including VMWare's ESX server and IBM's z/VM system. By selectively modifying the underlying physical architecture, Denali can scale to many more machines for a given hardware base. We are not aware of detailed studies of the scalability of these systems.

6 Conclusions

This paper presented the design and implementation of the Denali isolation kernel, a virtualization layer that supports the secure multiplexing of a large number of untrusted Internet services on shared infrastructure. We have argued that isolation kernels are necessary to provide adequate isolation between untrusted services, and to support scaling to a large number of Internet services, as required by cost-efficiency. Quantitative evaluation of our isolation kernel has demonstrated that the performance overhead of virtualization is reasonable, that our design choices were both necessary and reasonable, and that our design and implementation can successfully scale to over 10,000 services on commodity hardware.

We believe that isolation kernels have the potential to dramatically change how Internet services are deployed. An isolation kernel allows a service to be "pushed" into third party infrastructure, thereby separating the management of physical infrastructure from the management of software services and lowering the barrier to deploying a new service.

7 Acknowledgments

We are grateful for the help of our shepherd, Robert Morris, whose suggestions have greatly improved the final draft of this paper. We would also like to thank Brian Bershad, David Wetherall, Neil Spring, Tom Anderson, John Zahorjan, and Hank Levy for their valuable feedback. This work also benefitted greatly from many discussions with the Systems and Networking group at Microsoft Research, particularly Bill Bolosky, Marvin Theimer, Mike Jones, and John Douceur. This work was supported in part by NSF Career award ANI-0132817, funding from Intel Corporation, and a gift from Nortel Networks.

References

[1] M. Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, 1986.

[2] Collaborative advanced interagency research network (cairn). `http://www.cairn.net`, 1997.

[3] D. Balfanz and D.R. Simon. Windowbox: A simple security model for the connected desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.

[4] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating system design and implementation*, February 1999.

[5] L. Breslau et al. Web caching, and Zipf-like distributions: Evidence, and implications, Mar 1999.

[6] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[7] R. Carr and J. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the 8th Symposium on Operating System Principles*, Dec 1981.

[8] A. Chou et al. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, October 2001.

[9] R.J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), 1981.

[10] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, USA, October 1991.

[11] P. Druschel and G. Banga. Lazy receiver processing: A network subsystem architecture for server systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, USA, Oct 1996.

[12] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, March, 2001.

[13] B. Ford et al. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[14] B. Ford et al. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[15] A.F. Forin, D.B. Golub, and B.N. Bershad. An I/O system for Mach. In *Proceedings of the Usenix Mach Symposium (MACHNIX)*, Nov 1991.

[16] I. Goldberg, D. Wagner, R. Thomas, and E.A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the sixth USENIX Security Symposium*, July 1996.

[17] R.P. Goldberg. *Architectural Principles for Virtual Computer Systems.* PhD thesis, Harvard University, 1972.

[18] D. Gollmann. *Computer Security.* John Wiley and Son, Ltd., 1st edition, February 1999.

[19] Robert S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In *Proceedings of the Fourth Annual Usenix Tcl/Tk Workshop*, 1996.

[20] M.F. Kaashoek et al. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[21] P.A. Karger et al. A retrospective on the VAX VMM security kernel. 17(11), November 1991.

[22] J. Lepreau, B. Ford, and M. Hibler. The persistant relevance of the local operating system to global applications. In *Proceedings of the Seventh SIGOPS European Workshop*, Sep 1996.

[23] I. Leslie et al. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7), 1996.

[24] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. http://www.vmware.com/, 2000.

[25] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale Internet measurement. *IEEE Communications Magazine*, 36(8):48–54, August 1998.

[26] J. Reumann et al. Virtual services: A new abstraction for server consolidation. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, USA, June 2000.

[27] J.S. Robin and C.E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[28] J. Rushby. Design and verification of secure systems. In *Proceedings of the 8th Symposium on Operating System Principles*, December 1981.

[29] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[30] I.Stoica et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM 2001*, San Diego, USA, Aug 2001.

[31] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual Usenix Technical Conference*, Boston, MA, USA, June 2001.

[32] M.M. Swift et al. Improving the granularity of access control in Windows NT. In *Proceedings of the 6th ACM Symposium On Access Control Models and Technologies*, May 2001.

[33] D.S. Wallach et al. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Oct 1997.

[34] A. Wolman et al. Organization-based analysis of web-object sharing and caching. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS '99)*, Boulder, CO, Oct 1999.