Denali: Lightweight Virtual Machines for Distributed and Networked Applications

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

Abstract

The goal of Denali is to safely execute many independent, untrusted server applications on a single physical machine. This would enable any developer to inject a new service into third-party Internet infrastructure; for example, dynamic content generation code could be introduced into content-delivery networks or caching systems. We believe that virtual machine monitors (VMMs) are ideally suited to this application domain. A VMM provides strong isolation by default, since one virtual machine cannot directly name a resource in another. In addition, VMMs defer the implementation of high-level abstractions to guest OSs, which greatly simplifies the kernel and avoids "layer-below" attacks. The main challenge in using a VMM for this application domain is in scaling the number of concurrent virtual machines that can simultaneously execute on it.

The distinction between Denali and existing VMMs is that we make aggressive use of *para-virtualization* techniques. Para-virtualization entails selectively modifying the virtual architecture to enhance scalability, performance, and simplicity. By using para-virtualization, we believe Denali will be able to scale up to an order-of-magnitude more virtual machines than existing VMMs. We have implemented a prototype virtual machine monitor that runs in ring 0 on bare x86 hardware. In addition, we have built a simple guest OS tailored to writing Internet services.

1 Introduction

Improvements in networking and computing technology are pushing application functionality into the wide-area infrastructure. This computing model has many advantages: services are immediately available to clients without cumbersome software distribution, services are always available and can be accessed from any device, services can be administered centrally, and administration or maintenance can be out-sourced to an infrastructure service provider rather than handled in-house.

Many of today's services are maintained by large organizations, such as Hotmail. However, the benefits of infrastructure computing should apply just as well to small services. A popular vision that we share is that any individual should be able to inject a new service into the Internet infrastructure for a small fee. As an example, a group of game players could deploy a server to a well-connected point in the Internet for the duration of a multi-player game session. As another example, the owners of a web service that includes dynamically generated content could inject both static and dynamic portions of their site into a content-delivery network.

These scenarios have significant trust implications: infrastructure providers cannot trust consumers' services, and services generally do not trust each other. Correspondingly, a mechanism must exist to enforce strong isolation between services and the infrastructure, both in the security sense (preventing one service from corrupting another) and in the performance sense (fairly multiplexing physical resources such as CPU, memory, and network bandwidth). The simplest approach to providing this isolation would be to run each service on its own physical machine. In addition to isolating services from each other, this would also allow each service to choose its own operating system and software. However, dedicating physical machines to services is wasteful, as it eliminates the possibility of statistically multiplexing a machine across many services. It is also not cost-effective, as we believe there will be many services that neither require nor can afford the cost of an entire physical machine.

1.1 Statistically multiplexing services

The benefits of statistically multiplexing services are re-enforced by Zipf's law, which states that the frequency of an event is proportional to $x^{-\alpha}$, where x is the rank of the event compared with all other events. Many studies of web servers, documents, web caches, and other network services have shown that popularity is almost always driven by Zipfian distributions [7]. Based on this, we expect that the popularity distribution of infrastructure services will also be driven by Zipf's law.

Zipfian distributions have two significant implications (Figure 1). First, most requests go to a small number of popular services. Second, most

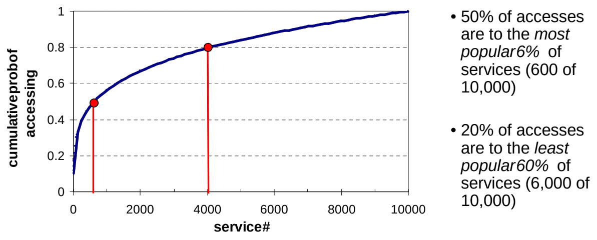

Figure 1: **Zipfian service popularity distribution:** This figure shows the CDF of requests to 10,000 hypothetical services driven by a Zipfian probability distribution, with $\alpha = 0.75$.

services are relatively unpopular, but a non-trivial fraction of requests go to these unpopular services. Because the amount of resources that a service requires is typically proportional to the workload it supports, popular services will require significant computational and networking resources. In contrast, there will be a large number of services that require scarcely any resources, motivating the desire to multiplex many of them on a single computer for reasons of affordability and manageability.

Fortunately, Moore's law has resulted in commodity components with enormous processing power, storage, and network bandwidth. A single modern computer can support a large amount of service traffic: recent SPECweb results show that single CPU servers can serve 2,000 HTTP requests per second, or 172 million requests per day. Correspondingly, we believe that if isolation can be enforced without introducing prohibitive overhead, a single computer can host a large number of concurrent services (hundreds, or perhaps thousands) while supporting an aggregate throughput that is comparable to a single-service computer.

1.2 Denali: supporting lightweight protection domains

The Denali project seeks to implement lightweight protection domains that allow many untrusted services to execute inside the network infrastructure. In particular, Denali's protection domains must have the following properties:

- **Strong isolation:** arbitrary code executing in a protection domain is prevented from perturbing code executing in another domain, both in terms of security and performance.
- **Scales to many protection domains:** in our application domain, we will need to execute hundreds or thousands of protection domains simultaneously on a single physical computer.
- **Rapid swapping:** to support a workload consisting of many requests to unpopular services, the act of swapping in a service that hasn't executed in a long time must be fast.

We also believe services will be *relatively independent*, and therefore sharing across protection domains is infrequent. Thus, a mechanism that obtains these three properties at the cost of increased sharing overhead is acceptable.

In this paper, we argue that virtual machine monitors (VMMs) are one of the few practical mechanisms that provide strong enough isolation for our desired application domains. Our research agenda includes mechanisms and design techniques to enhance the ability of a VMM to scale up in the number of concurrently executing virtual machines (VMs). This paper represents an initial exploration of this agenda, in which we use a prototype implementation of a lightweight VMM system (described in Section 3) to explore design possibilities. Section 4 provides performance measurements from several micro- and macro-benchmarks. We discuss future work in Section 5, related work in Section 6, and then conclude.

2 An Argument for VMMs

A virtual machine monitor is a software layer that virtualizes all of the resources of a physical machine, thereby defining and supporting the execution of multiple virtual machines (VMs) [13, 28, 29]. The interface exported by a VMM is a virtualized hardware/software interface, including a CPU, physical memory, and I/O devices. A VMM typically executes directly on physical hardware, and more specifically, below the level of operating systems. Within each VM, a "guest" operating system provides the customary set of high-level abstractions such as files or network sockets (Figure 2).

We believe VMMs are capable of providing strong isolation between virtual machines, both in the security and performance sense. However, for VMMs to work in our application domain, they must demonstrate adequate performance as the number of concurrently executing VMs scales up. In the remainder of this section, we discuss the isolation properties of VMMs and then introduce some issues of scale that arise when executing many VMs.

2.1 Security isolation

One of our isolation objectives is to *sandbox* untrusted code to prevent services from directly reading or modifying the state of other services or of the underlying protection system.[1] The requirements

[1]We are not pursuing stronger properties such as monitoring information flow or eliminating covert channels.

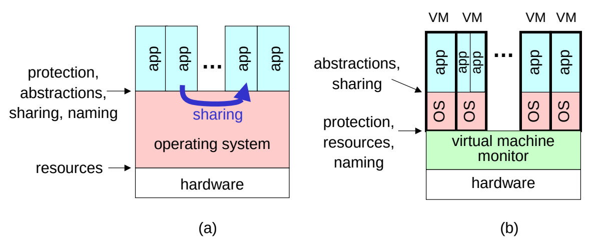

Figure 2: **OSs compared with VMMs:** (a) An OS shares and protects high level abstractions built out of low-level physical resources. (b) With a VMM, protection is below abstraction. By exposing virtualized (protected) low-level resources, each VM can run its own OS to define high-level abstractions for its applications.

of our application domain are fundamentally different from those that guided the design of protection in conventional desktop or time-sharing operating systems. In Denali, the unit of protection is a service instead of a user, and there is little need to share data between services (and hence protection domains). Virtual machine monitors are well-suited to this application domain because they directly address problems that plague conventional OSs:

Simple, static sharing policy. VMMs impose a simple sharing policy: all data is private to a virtual machine unless it wishes to share that data over the network. The advantage of this approach is that it obviates the task of constructing an appropriate protection policy. The disadvantage is the increased cost of sharing data between applications. However, we believe this trade-off is justified, given that our application domain demands little sharing between applications.

In contrast, the principals in a conventional OS are users that share data through protected abstractions; this results in very complicated sharing policies (e.g., "allow Jim to read file X" or "allow all of Sally's programs to use the network"). The complexity of expressing an appropriate policy grows with the number of principals and protected abstractions. Even if an OS is flexible enough to allow all policies to be expressed, this complexity implies that, in practice, it is difficult to verify that a given policy behaves according to its author's intentions. By removing protected sharing, VMMs avoid the issue of expressing complex policy.

Protection is below abstractions. VMMs defer the implementation of high-level abstractions like file systems and network stacks to guest operating systems. This greatly simplifies the implementation of the VMM (which has positive security implications), and it also eliminates "layer-below vulnerabilities" to which conventional operating systems are susceptible.

In a conventional OS, policy is expressed in terms of high-level abstractions like files, instead of low-level resources like disk blocks. Unfortunately, expressing protection policy in terms of abstractions gives rise to layer-below phenomena in which an attacker illicitly accesses resources by tunneling below the abstraction layer. For example, an attacker could read raw disk blocks to bypass the file system reference monitor, use a packet sniffer to capture the password of a local account, or force a core dump to access protected in-memory data.

Private namespaces. With the exception of network addresses, all names exposed by a VMM are private to a VM. As a result, a VM *cannot even construct a name* that refers to the resource of another VM. Even if a VM is compromised by an attacker, it cannot access any other machine's data, assuming that the VMM's mapping from virtual to physical resources is implemented correctly. The only global (and hence shared) namespace that a VMM exposes is the set of MAC addresses on the virtual ethernet subnet. Security vulnerabilities that can be exploited over the network are beyond the scope of our project; however, we point out that any network-enabled application must be prepared to handle malicious traffic that arrives from the network, and that a VM that desires complete isolation need only drop all network traffic.

In comparison, an operating system typically exposes several global namespaces (such as the set of all file names) through which users share data. These global namespaces can jeopardize security if they are misconfigured or poorly protected; for example, attackers could use aliases such as symbolic links to gain illicit access to resources. Global namespaces also grant unfettered access to an attacker that has gained supervisor privileges.

2.2 Performance isolation

Although the term "isolation" typically refers to security, an equally important aspect of service isolation is performance isolation (as evidenced by the rash of recent denial-of-service attacks). Our goal is to provide approximate resource fairness across services, even in the presence of malicious services or heavy network load. We do not aim to provide precise guarantees of the sort that are required by real-time applications.

The need to support high-level abstractions prevents most OSs from providing strong performance isolation. High-level abstractions create contention points where applications compete for resources and synchronization primitives. This leads to the effect of resource "cross-talk" [19], in which applications' resource management decisions interfere with each other. An additional problem posed by high-level abstractions is that precise resource accounting is difficult because resources are tied up in the implementation of the abstractions themselves. For example, the file buffer-cache and TCP/IP socket buffers consume memory resources that aren't "charged" to any particular application. Likewise, network protocol processing is often performed in the context of the running process instead of the receiving process, which can lead to unfairness and receiver live-lock [15].

By deferring the implementation of abstractions to guest OSs, VMMs need not suffer from these deficiencies. As we will show in Section 3, virtual hardware devices within the VMM act as queues for VMs' resource accesses, making it possible for a VMM to implement policies such as fair queuing and stride scheduling. Because the VMM exposes hardware-level resources, there are fewer unaccounted resources than in conventional operating systems.

2.3 Our challenge: scaling a VMM

Our goal in Denali is to support a large number of protection domains efficiently. VMMs are known to introduce virtualization overhead, but as we confirm in Section 4, the performance degradation from this overhead is modest on today's machines. More importantly, there are many issues of scale that arise as we increase the number of concurrently executing virtual machines. For example, at the architectural level, as more VMs concurrently execute it becomes less likely that a given physical interrupt arrives when the VM it is associated with is running. Issues of scale also affect operating system design; running hundreds of VMs implies executing hundreds of TCP/IP stacks on the same physical processor, which has implications for timer design.

In the following section, we describe how we exploit the notion of *para-virtualization* to address issues of scale. Para-virtualization exposes a virtual architecture that is slightly different than the physical architecture. The differences in the architecture are driven by improvements in scalability or reductions in system complexity. Modifying the architecture breaks backwards compatibility with existing OS code, which is a major disadvantage. However, it enables us to co-design the virtual architecture with the operating system, which gives us considerable latitude when exploring issues of scale.

Para-virtualization has been used in previous VMMs, including VM/370 [29] and Disco [9]. These systems added a combination of instructions, registers, or devices to the virtual architecture to improve performance. However, because the goal of these systems was to run legacy OSs, their use of para-virtualization was minimized. Our contribution is to explore architectural modifications without regard to backwards compatibility of OS code.

A potential criticism of para-virtualization is that it blurs the line between a VMM and a conventional OS. While this is true, we chose the term "virtual machine monitor" because we find virtualized hardware to be a useful metaphor for implementing the isolating properties discussed above: no shared abstractions, a simple sharing model, and no global namespaces. We discuss the relationship between Denali and previous systems in Section 6.

3 Design and Implementation

In this section, we describe Denali's para-virtualized architecture. In addition, we describe a prototype VMM and guest OS that utilize this architecture.

3.1 Para-virtualization in Denali

The Denali architecture is based on the x86 instruction set; this allows most virtual instructions to execute directly on the physical processor. However, the Denali architecture differs from the underlying x86 architecture in a number of ways that improve scalability, reduce implementation complexity, and increase performance.

We have introduced purely *virtual instructions* that have no counterpart in the physical architecture; these are conceptually similar to OS system calls, except that they are non-blocking and they operate at the architectural level instead of at the level of OS abstractions. We have *modified existing instructions' semantics*; although we cannot remove instructions from the physical architecture, we classify certain rarely used instructions as deprecated and having undefined semantics. We have added *virtual registers* as a lightweight mechanism for passing data between the VMM and its VMs; these registers are mapped to a well-known region of a VM's address space. Our virtual I/O devices export a *simplified architectural interface*, designed in part to minimize VM/VMM boundary crossings. Finally, as we will describe, other architecture features are heavily modified (e.g., interrupt delivery) or eliminated (e.g., virtual memory).

3.1.1 Para-virtualization for scalability

A simple barrier to scaling up to hundreds of VMs is that an OS must execute an idle loop when it has no useful work to do. In a VMM system, these loops waste CPU cycles, degrading the performance of the system. Denali introduces an "idle" instruction that allows a VM to yield control of the processor. After invoking it, the VM remains unscheduled until a new virtual interrupt arrives for it. By invoking this instruction, the VM promotes higher CPU utilization and increases its own performance, as it is no longer charged for these cycles.

The idle instruction is similar to the x86's halt instruction, which puts a physical machine to sleep awaiting an interrupt. Denali's idle instruction enhances this functionality with a timeout parameter that allows a VM to bound its sleep time. This effectively introduces a yield primitive that allows for fine-grained sharing of the processor, such as when a VM is waiting for a TCP timeout to expire.

A second scalability obstacle relates to virtual interrupt dispatching; when a physical interrupt arrives, the VMM raises a virtual interrupt in the appropriate VM. As the number of VMs grows, it becomes increasingly unlikely that the physical interrupt is destined for the currently running VM: handling physical interrupts destined for an inactive VM is the common case. One possible policy is to context switch to the destination VM immediately upon physical interrupt arrival. This synchronous dispatch model preserves timely delivery of interrupts, but unfortunately incurs the large cost of two context switches, which can result in context-switch thrashing as the number of VMs grow. Additionally, synchronous interrupt delivery fails to provide performance isolation in the presence of a denial of service attack.

Instead, Denali exposes an asynchronous interrupt dispatch mechanism in which physical interrupts are queued until the target VM runs. Multiple interrupts destined for the same VM are batched, reducing VMM/VM boundary crossings and allowing the guest OS to handle virtual interrupts in an order of its own choosing. The importance of batching grows as the number of VMs (and hence the number of queued interrupts) increases.

We have also modified the semantics of interrupts to improve scalability. On physical hardware, interrupts generally imply that *something just happened.* In Denali, a virtual interrupt implies that *something happened in the recent past, possibly while you were context switched out.* This semantic shift is particularly useful in the implementation of virtual timers. An OS typically maintains a "ticks" variable that is incremented on each hardware timer tick; mimicking this behavior on a VM requires raising a virtual interrupt for each timer tick that occurs while the VM isn't running. Instead, Denali raises a "time has passed" virtual interrupt, and it exposes the number of physical timer ticks since system start in a virtual register. This eliminates additional VMM/VM crossings to determine elapsed physical time.

3.1.2 Para-virtualization for simplicity

Hardware architectures are complex. Precisely replicating a physical machine requires a VMM to emulate many hardware constructs: privileged machine instructions, virtual memory, the BIOS, and I/O devices. However, some of these features are not necessary for our application domain. Para-virtualization provides an opportunity to remove or modify these features, vastly simplifying our VMM.

An example of architectural complexity is the presence of non-virtualizable instructions in the x86 instruction set [37]. These instructions behave differently in user mode and kernel mode; because virtual machines execute with the physical processor in user mode, this breaks backwards compatibility with legacy code. As a result, x86 VMMs such as VMWare and Plex86 [32] require elaborate binary-rewriting and virtual memory protection techniques to prevent these instructions from being directly executed. Because we are not concerned with backwards compatibility, we are content to deprecate these instructions.[2] Although the effects of these instructions are undefined, they are confined to the issuing VM.

A more radical architectural simplification is that Denali does not expose virtual memory hardware. Denali's virtual machines are constrained to use single address spaces, implying the use of library OSs similar to those in the Exokernel [17]. We believe this change is warranted because Denali targets small applications that do not require internal protection mechanisms. If multiple protection domains within an application are required, language techniques can be employed (as in Pilot [18]).

We have eliminated other x86 components from our virtual architecture as well. The BIOS is used primarily to bootstrap a conventional OS and to determine system-specific parameters.[3] We have replaced the BIOS bootstrap functionality by simply having the VMM load a VM's image into memory, much like a process is loaded by an OS. System parameters such as CPU speed and the size of (virtual) physical memory are accessible in read-only virtual registers.

Our final set of architectural simplifications relate to I/O devices. Denali exports a small number of generic devices, rather than the large number of heterogeneous devices found on most systems; we currently support a network interface card, a serial de-

[2]The most commonly used of these are pushl and popl, which are used to enable and disable interrupts. We replaced this functionality with a virtual register that serves as an interrupt-enabled flag. This also eliminates a VMM/VM crossing when virtual interrupts are enabled or disabled.

[3]The BIOS also contains power management functions; Denali does not expose power management to VMs.

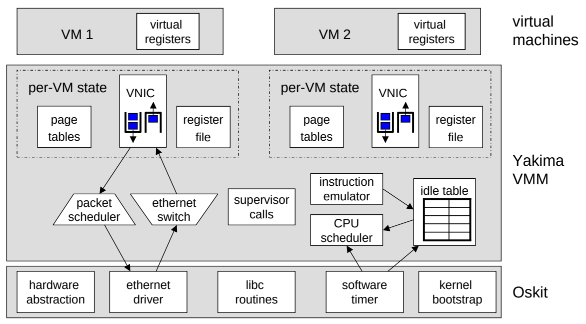

Figure 3: **The Yakima virtual machine monitor:** Arrows represent control and data flow. Some components (such as the virtual keyboard) are not shown.

vice, a timer, and a generic keyboard/console. Device interfaces are streamlined to minimize the number of VMM/VM crossings. For example, transmitting any number of ethernet frames requires a single virtual I/O instruction. By contrast, existing physical NICs can require a dozen I/O instructions to implement the same functionality [39]. Additionally, Denali's virtual devices do not require initialization during startup, simplifying guest OS device driver implementation and reducing OS boot time.

3.2 The Yakima VMM implementation

Our prototype VMM implementation, called Yakima, runs in ring 0 on bare x86 hardware. Yakima is event-driven and non-blocking, and the only thread owned by the VMM is an idle thread. Currently, Yakima only runs on uniprocessors, but we have designed it to be extensible to SMPs.

Figure 3 illustrates the major components of Yakima. At the lowest layer of the system are support libraries from the Flux OSKit [22]. We use the OSKit as a hardware abstraction layer to simplify interactions with devices, page tables, interrupt vectors, and the BIOS. We also use a small portion of the OSKit's libc library for dynamic memory management and interacting with the console. We do not use any of the OSKit's larger libraries for processes, network stacks, and similar abstractions.

The Yakima VMM multiplexes physical resources and exports the Denali architecture to each VM. At Yakima's core is a simple round-robin CPU scheduler coupled with a timer to prevent a VM from stealing the CPU; improving the scheduling policy is a topic for future work. Affiliated with the scheduler is an idle table, which contains a list of idle machines (those that invoked the idle instruction described in Section 3.1.1). Yakima wakes up an idling VM when either a virtual interrupt arrives or its idle timeout value is exceeded.

The instruction emulator implements instructions that cannot be directly executed by a VM, including I/O instructions and the halt instruction that terminates a VM. The instruction emulator also implements virtual instructions that have no counterpart in the x86 architecture. Denali's virtual instructions are mapped to illegal opcodes, which Yakima traps and emulates. At the moment, our only virtual instruction is the idle instruction.

Yakima emulates an ethernet subnet; each machine has its own MAC address and is outwardly indistinguishable from a physical machine. Yakima maintains receive and transmit FIFOs on behalf of each VM; these emulate the FIFOs that exist on real NICs. At a lower layer, a packet scheduler and a virtual ethernet switch perform network multiplexing and demultiplexing, respectively. Currently, the packet scheduler uses a simple round-robin scheduling policy. We plan to explore more sophisticated fair queuing policies in the future.

Yakima's approach to memory management is to statically allocate physical pages for each active virtual machine. Although static allocation is inefficient, it is simple to implement and avoids worst-case thrashing behavior. To date, static memory allocation has proven to be reasonable for our application domain: our web server VM requires only 12 megabytes of memory, which allows for over 80 concurrently active VMs on a physical machine with 1 gigabyte of physical memory.

The protection of each virtual machine's physical address space works in much the same fashion as a conventional OS. Yakima maintains page tables for each virtual machine; the address space visible to a VM contains a VM-accessible region, and a second region which is only accessible to the VMM.[4]

Yakima also includes support for a *supervisor virtual machine*, which is responsible for bootstrapping the VMM. The supervisor VM has access to privileged VMM calls to create and destroy other VMs. Currently, any user with physical access to the machine can issue supervisor calls. If more sophisticated security policies are desired, it would be straightforward to replace or enhance the supervisor VM with additional functionality.

3.3 Ilwaco: an example guest OS

We have developed a simple guest OS, named Ilwaco, which provides high-level abstractions to applications and shields applications from the details

[4]While this appears to violate the namespace isolation advocated in Section 2, we argue that isolation is not affected because protection is still based on statically mapped page table entries. Whether these entries reside only in the VMM address space or are mapped into each VM but protected from VM access is irrelevant. Exposing the VMM's address space in each VM eliminates TLB flushes on VMM/VM crossings, vastly reducing virtualization overhead.

of the Denali virtual architecture. Among the abstractions provided by Ilwaco are a TCP/IP stack, a threads package, a subset of the libc library, and the BSD sockets interface.

The Ilwaco TCP/IP stack is a port of the Alpine user-level TCP/IP infrastructure [16]. Alpine consists of the FreeBSD 3.3 stack and a support library that emulates the BSD kernel environment. We modified the support library to use Denali's interrupt and timer models, and linked the stack against a device driver for the Denali virtual NIC.

Ilwaco contains a threads package that includes basic primitives such as fork, kill, locks, and condition variables. Ilwaco threads are non-preemptive; this simplified development since the BSD TCP/IP stack assumes a non-preemptive thread environment. If a thread performs a timed sleep operation, the thread scheduler adds the sleep duration to a priority queue. If there are no runnable threads, the scheduler passes the smallest sleep duration to the Denali virtual idle instruction.

Finally, Ilwaco's supported subset of libc enables basic console I/O via printf and scanf, string manipulation, random number generation, and memory management. The majority of these functions were ported from OSKit libraries. Some functions were modified to interact with Denali's virtual hardware; for example, malloc reads the size of (virtual) physical memory from a virtual register.

3.4 Work in progress

Denali is a work in progress, and there are several pieces of functionality that have yet to be implemented. Our highest priority is the implementation of stable storage functionality. Despite the lack of disk, our system supports a non-trivial web server VM, as we will describe in the next section.

The resource management policies in our prototype VMM are overly simplistic. Both the packet and CPU schedulers use simple round-robin scheduling policies. Neither scheduler accounts for the amount of resources used during a round-robin iteration—the packet size for network traffic and the quantum for CPU scheduling. Although these schemes are sufficient to prevent starvation, they are not suitable for enforcing robust performance isolation. We are working to incorporate existing scheduling algorithms like fair queuing [14] and stride scheduling [41].

Our VMM can only execute as many VMs as will fit in physical memory: we have not yet implemented the swapping of an idle VM to disk. In addition, supporting a large number of inactive virtual machines will require changes to the guest OS. For example, the TCP stack registers timers that fire every 200 milliseconds and every 500 milliseconds. If left unmodified, this would force an inactive virtual machine to be swapped in 5 times per second, regardless of whether there are any pending connections. There are likely to be many such elements of a conventional OS that must be modified to improve the scalability of our system.

4 Measurements

In this section, we describe a set of micro-benchmarks and application level benchmarks designed to show the performance of the prototype Yakima VMM and of applications executing on our example guest operating system.

For all of the experiments described below, we ran our VMM and VMs on a 1700MHz Pentium 4 with 256KB of L2 cache, 1GB of RAM, and Intel PRO/1000 PCI gigabit ethernet cards connected with Intel 470T ethernet switches. Within Yakima, we ported version 3.0.7 of Intel's PRO/1000 device driver to the Flux OSKit Linux driver glue substrate. We used 1500 byte MTUs for our ethernet packets. To generate workloads for our server benchmarks, we used a mixture of several 1700MHz Pentium 4 and 930MHz Pentium III machines, thereby ensuring that the workload generation clients were not the bottleneck of the system.

4.1 Micro-benchmarks

Our first set of measurements attempt to characterize the performance of the Yakima VMM, independent of application-level behavior.

4.1.1 Context switch time

We measured the time to context switch between virtual machines. To quantify the effect of cache pollution, we considered two workloads: the "worst-case test" cycles through a large memory buffer between each context switch; the "best-case test" does not touch memory between context switches. Preemption was disabled for these tests.

Figure 4 describes the context switch time as a function of the number of virtual machines. For the worst-case workload, context switch time starts at 3.9 microseconds for a single virtual machine, and increases to over 9 microseconds for multiple VMs. For the best-case workload, the context switch time starts at 1.4 microseconds for a single virtual machine and exhibits small peaks as we exhaust the capacity of the L1 and L2 caches.

Taken as a whole, these numbers suggest that Denali's context switch time is manageable. Even the worst-case time of 9 microseconds is small relative to the thousands of microseconds that are required

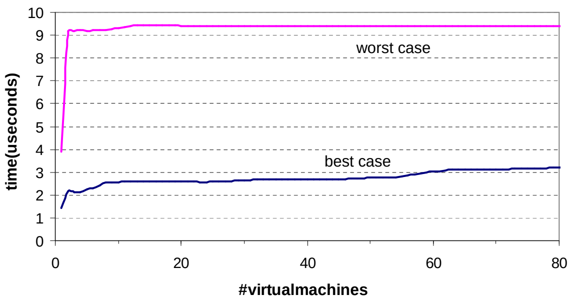

Figure 4: **Yakima context switch time:** The worst-case context switch time, with a memory intensive workload, tops out at 9.4 microseconds. The best-case context switch time, whose workload does not touch memory, tops out at 3.2 microseconds.

for TCP/IP protocol processing (refer to Figure 6 below). In addition, over 40% of the context switch time is devoted to simply entering and exiting the kernel, which suggests that a conventional OS would demonstrate similar performance.

4.1.2 Control flow from VM to VMM

VM to VMM control flow transfers can happen in two situations. First, the supervisor VM can invoke a privileged system call into the VMM by executing the `int` instruction. Second, the VMM will trap and emulate privileged instructions, which happens when a VM issues programmed I/O using the `inb/outb` instructions. The end result of both is the same: control is vectored to a kernel address specified in the x86 interrupt descriptor table[5].

We measured the transfer time from VMs using a null system call and a generic programmed I/O instruction. The null system call is slightly cheaper than a programmed I/O instruction: 1759 cycles for the system call versus 2129 cycles for the PIO. In retrospect, using the `int` instruction for all control transfers, instead of using `inb/outb` for PIOs, would provide slightly better performance. Fortunately, the performance difference is not noticeable in practice.

4.1.3 Packet processing overhead

Figure 5 shows the cost of packet processing for application-level UDP packet transmission and reception, for both 100 and 1400 byte packets.

A transmitted packet first traverses the TCP stack and then is processed by the guest OS device driver. This driver signals the virtual NIC using a PIO, resulting in a trap into the VMM. Inside the VMM, the virtual NIC implementation copies the packet out of the guest OS into a Tx FIFO. Once the VMM has decided to transmit the packet, the physical device driver is invoked.

Packet reception essentially follows the same path in reverse. When the physical NIC receives a packet, it raises an interrupt, causing a device driver to execute. The driver hands the packet to the VMM, which then demultiplexes it into an appropriate virtual NIC Rx FIFO. When the virtual NIC is ready to hand the packet to its VM, the VMM copies the packet into the guest OS, and raises a virtual interrupt. The guest OS's device driver processes the packet and gives it to the TCP stack, eventually resulting in the packet being handed to the application.

As indicated in Figure 5, the physical device driver and TCP stack incur significantly more cost than the VMM itself. Handling a received packet in the physical device driver represents 43.3% and 38.4% of the total packet processing costs for small and large packets, respectively.[6] A non-trivial portion 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. The TCP stack represents 37.3% and 41.8% of a small and large received packet's processing time, respectively. Of course, it is possible to optimize the stack within the guest OS to reduce overhead.

The transmit path currently incurs two packet copies and one VM/VMM boundary crossing; it may be possible to eliminate one or both of these copies using virtual memory copy-on-write techniques. The receive path incurs the cost of a packet copy, an mbuf deallocation within the Flux OSKit, and a VMM/VM crossing. The mbuf deallocation attempts to coalesce the mbuf memory back into a global pool, and is therefore fairly costly. With additional optimization, we believe we can lower this cost as well.

4.2 Application-level benchmarks

The second set of measurements that we gathered were end-to-end measurements of network applications running on top of our Ilwaco guest OS. These measurements show two things. First, the absolute performance numbers we obtain are comparable with those of a conventional operating system (Linux), and therefore that the overhead of virtualization is not prohibitive. Second, performance is upheld as we scale up the number of virtual machines running on a single system. These scaling numbers are only a start: as future work, we plan on exploring scaling issues in detail as we scale up

[5]PIO instructions raise a general protection fault when executed in user mode.

[6]While the device driver appears to be less expensive on the transmit path, the cost of interacting with the NIC is not included in these numbers, since this interaction is asynchronous and largely driven by the NIC itself.

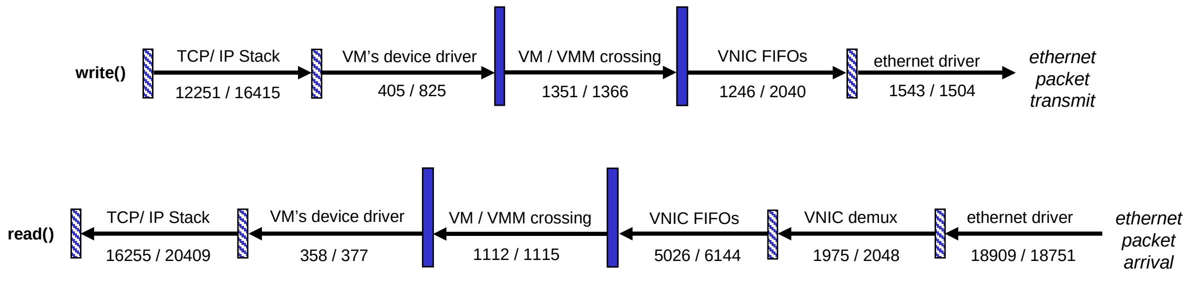

Figure 5: **Packet processing overhead:** These two timelines illustrate the cost (in cycles) of processing a packet, broken down across various functional stages, for both packet reception and packet transmission. Each pair of numbers represents the number of cycles executed in that stage for 100 byte and 1400 byte packets, respectively.

far beyond the roughly 10 VMs that we present in this paper.

4.2.1 TCP/IP performance

To measure TCP/IP throughput and latency, we wrote a simple application that opens a TCP connection to a remote server and sends data as quickly as possible. The remote server calculates aggregate TCP/IP throughput across all measured VMs. To measure end-to-end latency, we ping the supervisor machine while the throughput test is in progress.

The results of these tests are shown in Figure 6. TCP/IP throughput reaches a peak value of 560 Mb/s for 2 virtual machines.[7] The low value for a single VM is due in part to TCP dynamics: because our TCP/IP stack runs within a VM, the system must cross the VMM/VM boundary before sending data in response to an ACK. This overhead cannot be optimized without pushing knowledge of TCP into the VMM. Fortunately, these effects are masked with multiple VMs because the VMM maintains an outbound transmit FIFO for each virtual machine; this means that the VMM can overlap packet transmission from one VM with computation within another. The bandwidth drop for more than 2 VMs is likely due to cache contention: the size of the TCP socket buffers is exactly half the L2 cache.

When only the supervisor VM was running, the baseline ping time was 212 microseconds. Each additional TCP-intensive virtual machine caused the ping time to increase by roughly 3 milliseconds. This suggests that latency may prove to be more problematic than bandwidth when running a large number of concurrent VMs, and that additional techniques for reducing each VM's compute time may be required. Eliminating unnecessary memory copies should prove beneficial in this respect.

[7]We did not explicitly consider fairness between VMs in this test. Informally, we observed a low variation of bandwidth across VMs.

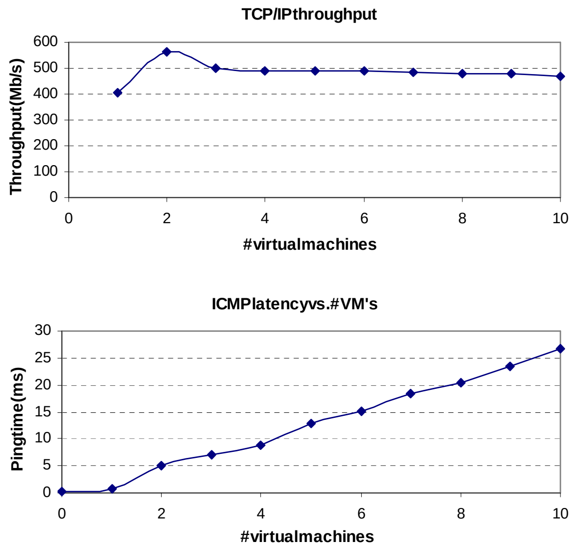

Figure 6: **TCP throughput and latency vs. # VMs:** The top graph shows the average TCP throughput for a set of n virtual machines. Throughput reaches a peak value of 560 Mb/s for 2 VMs . The bottom graph shows end-to-end latency, which increases by roughly 3 milliseconds per virtual machine.

4.2.2 Web server performance

Our final set of benchmarks test the performance of a web server running on top of Ilwaco. We implemented our own simple multi-threaded server that dispatches incoming requests to a pre-allocated thread pool. The web server serves static content; because we do not yet have a file system in Ilwaco, we used a Perl script to embed a document tree in a Unix file system into a set of C source files that was statically compiled into the application.

We used the httperf [34] tool running on Linux 2.4.7 to generate workloads for our benchmarks. Using this tool, we subjected the web server to two different workloads, varying the rate at which we generated requests and measuring the throughput, latency, and error rate of the server under these dif-

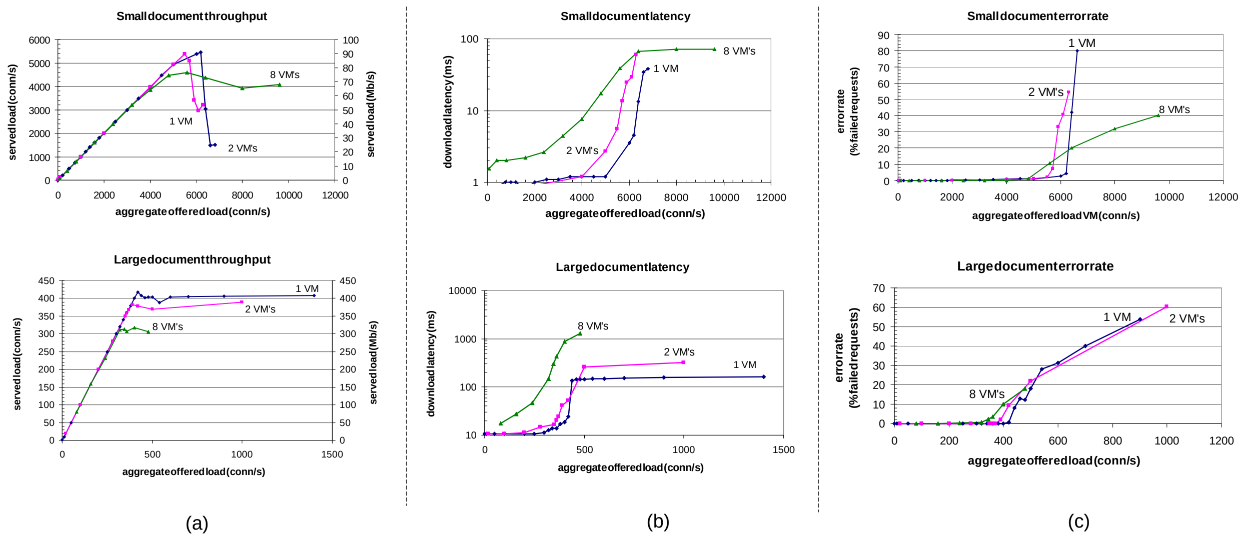

Figure 7: **Web server performance:** As a function of offered load and the number of concurrent VMs, these graphs illustrate (a) the throughput (in connections/s and megabits/s) of our web server, (b) the download latency of our web server, and (c) the error rate of our web server. The "small" document workload has an average document length of 2079 bytes, and the "large" document workload has an average document length of 134,128 bytes.

ferent loads. We also varied the number of simultaneously running web server VMs; the reported numbers below represent the aggregate load summed up over all VMs. The *small document* workload consists of repeated requests to a set of small HTML files averaging 2,079 bytes including HTTP headers. The *large document* workload consists of repeated requests to a single 134,128 byte PDF file.

Figure 7 illustrates our results. For the small document workload, a single VM achieved a peak sustained response rate of 5,379 documents/second. This rate greatly surpasses the saturation point of most popular web servers (including Apache), but this is not a fair comparison, since Apache is more fully featured than our server. For the large document workload, a single VM achieved a peak sustained data throughput of 449.6 Mb/s, which is commensurate with the peak aggregate TCP throughput observed in Section 4.2.1.

As we scaled up the number of VMs, we noticed that the aggregate bandwidth served by the entire system dropped, although we observed that bandwidth was fairly split across the VMs. For the small document workload, the aggregate throughput for 8 VMs was 72.8 Mb/s, compared with 86.6 Mb/s for the single VM case. For the large document workload, the 8 VM aggregate throughput was 324 Mb/s, compared with 427.7 Mb/s for the single VM case. We believe that this drop in performance is due to a combination of context switching overhead and perturbations in TCP dynamics because of context switching across machines; proving this hypothesis is the subject of future work.

For comparison, we reran our benchmarks against the same web server running on Linux. We compiled the server against a Linux library implementing Ilwaco's system call API. For the small document workload, we observed a peak response rate of 5,490 documents/second, and for the large document workload, we measured a sustained throughput of 507.9 Mb/s. The fact that Linux achieves only slightly higher TCP/IP throughput demonstrates that the overhead of virtualization introduced by Yakima is relatively small.

5 Future Directions

As we discussed in Section 3.4, there are several short-term directions that we are pursuing, including adding disk support, swapping of idle VMs, the exploration of more sophisticated resource management algorithms, and exploring issues of scale in depth. Besides this, we have a longer-term research agenda. We believe that lightweight virtual machines are a tool that enables many powerful, new applications that we plan to explore.

Inserting services below the OS: VMMs provide a layer of indirection between the guest OS and hardware; this facilitates the insertion of new system services transparently to the OS. Two services of interest are checkpointing and migration; VMs are uniquely suited to provide these because all OS and application state is accessible to a VMM. OS state such as file descriptors and socket buffers can be captured by the VMM in the memory and disk footprint of a VM. The only additional state to be captured is virtual hardware state, such as requests queued in virtual disk FIFOs.

VMMs also provide the opportunity to hot-swap virtual devices transparently to a VM. For example, a generic virtual block device interface could be mapped to a conventional hard disk, a RAID array, or a distributed disk like Petal [33]. Other potential services include NUMA memory management [9], fault tolerance [8], and secure logging [11].

Virtual machines for content distribution: A significant challenge for Internet services is dealing with client load that can vary over several order of magnitudes. The problem of flash-crowds on the web motivated replication mechanisms such as content delivery networks and proxy caches; unfortunately, these systems can only handle static data to date. According to a recent study by Wolman et al. [43], between 20–40% of web documents are dynamically generated and are therefore not amenable to these systems.

We propose using VMMs inside a CDN to generate dynamic content at the edge of the network. In addition to providing improved availability through replication, CDNs can mitigate wide-area network failures, which are a significant cause of service outages [10]. VMMs enforce security and performance isolation, allowing service providers to provide guaranteed service levels to their clients. Lightweight VMs would let a CDN host a large volume of dynamic content, and VM migration would enable the demand-loading of active content into CDNs.

Virtual clusters: VMMs introduce the possibility of emulating many virtual clusters on top of a physical cluster of workstations [3]. In such a system, a service author would provide a collection of processes that would execute on a collection of virtual cluster nodes. Cooperating VMMs executing on each physical node in the cluster would map each service's virtual machines onto the physical resources in the cluster. This would enable better multiplexing among a large set of services with bursty request streams. As the relative load on a virtual cluster increases, the system could increase that virtual cluster's relative share of physical resources. We also anticipate having the VMMs interact with L2/L4 load balancing switches, in order to map externally visible IP addresses onto virtual clusters.

The ability to migrate VMs can be used as a load-balancing mechanism in virtual clusters. If two popular virtual clusters map to the same physical machines, some of the virtual machines can be moved to take advantage of idle resources elsewhere. Also, migration would enable under-utilized machines to be switched off, which has positive repercussions for power conservation.

6 Related Work

6.1 Operating System improvements

Many projects have sought to improve the OS as a reference monitor to isolate untrusted code. Privilege subsetting defines restricted rights for untrusted code, distinct from normal user privileges [6, 12, 30]. Although this provides *mechanism* to isolate untrusted code, the problem of expressing appropriate *policy* is not specifically addressed. These proposals typically do not address layer-below attacks or vulnerabilities due to global namespaces.

To address file system global namespace vulnerabilities, some OSs provide a `chroot` system call to contain a process to a subtree in the global file system. In theory, additional OS extensions could provide similar containment in other shared namespaces, such as PIDs and network addresses. However, this would not eliminate layer-below vulnerabilities as the OS would still retain the significant complexity associated with high-level abstractions. For example, it has been shown that it is possible to use cached file descriptors to break out of a `chroot`'ed namespace.

Several systems propose mechanisms to supplement the OS reference monitor. Janus [27] allows a user-level server to intercept systems calls from untrusted processes. Software wrappers [24] provide better performance by pushing the interception layer into the kernel. These systems provide mechanisms, but again do not address the problem of appropriate policy. Unfortunately, OSs have hundreds of system calls, implying policy is complex. Additionally, these systems require knowledge of application behavior, while virtual machines do not.

A promising approach to isolation is inferring application behavior using system call traces [1] and machine learning techniques [25]. However, this approach confines applications to "typical" patterns of behavior and doesn't respond well to non-malicious changes in application behavior. Virtual machines provide the ability to isolate untrusted code without knowledge of application semantics.

WindowBox [4] seeks to isolate untrusted code to a virtual desktop inside Windows 2000. However, because it is implemented inside a conventional operating system, WindowBox's security is limited by high-level abstractions and global namespaces. By virtualizing at a layer below operating system abstractions, VMMs are more secure.

Fluke [23] proposes a recursive VM model, in which a parent can re-implement OS functionality on behalf of its child processes. In Denali, we virtualize at a layer *below* OS abstractions, whereas Fluke's virtual architecture includes high-level IPC calls. By virtualizing at the level of hardware, we avoid imposing a fixed set of protection abstractions and nullify layer-below vulnerabilities.

Server and multimedia systems have led to OS improvements for performance isolation. Resource containers [5] demonstrates that OS abstractions for resource management (processes and threads) are poorly suited to applications' needs. Our work differs in that resource management is below OS abstractions, which makes precise resource accounting more tractable and accurate.

A limitation of VMs as resource principals is that they do not span multiple protection domains, making it difficult to have a common service (like a DNS resolver) shared by all VMs. The virtual services work [36] is notable for tracking resource usage across server boundaries. However, this work suffers from significant implementation complexity; it also relies on intercepting system calls, which is subject to the same caveats as the Janus work.

Many proposals for fair resource allocation policies exist: fair queueing for network bandwidth [14], stride scheduling for CPU allocation [41], and the Cello framework for disk bandwidth allocation [38]. This work is complementary to Denali; we plan on incorporating some of these policies in our VMM.

6.2 Software virtual machines

Software virtual architectures such as Java, OmniWare [2], and the Microsoft Common Language Runtime have been proposed to isolate untrusted code. However, running multiple applications in a single VM has many of the same problems as running multiple applications on an OS. Libraries (e.g., Java's class library) provide shared abstractions that can be subverted through layer-below attacks. The trend toward extensible security architectures [42] means that security policy must be expressed in two places (the host OS and the software virtual machine). Finally, resource management within a single VM is complicated by the ability to share resources through pointers.

Alternatively, each application could be isolated in its own software VM, similar to a hardware VM architecture. Software VMs require complex software runtimes, which, in the case of Java, has led to numerous security vulnerabilities. We believe a VMM that is nearly identical to the underlying hardware is simpler to build and more robust.[8].

[8]We are intrigued by the possibility of using transparent instruction set mapping, as is done on the Transmeta Crusoe processor.

6.3 Small kernel architectures

VMMs have served as the foundation of several "security kernels" [26, 31, 35]. More recently, the NetTop initiative has sought to create secure virtual workstations running on VMWare [40]. Our work differs from these efforts in that we aim to provide scalability as well as isolation. Our work also assumes a weaker threat model: we are not concerned with covert channels between VMs.

Denali is similar in many respects to microkernel operating systems [20]. Indeed, Denali's virtual machines could be viewed as single-threaded applications on a low-level microkernel. However, the main focus of microkernel research is in pushing OS functionality into shared servers, which are themselves susceptible to the security vulnerabilities discussed in section 2. While it may be possible to remove shared servers from a microkernel system, our work addresses issues related to scaling to a large number untrusted applications. On the other hand, fast IPC is not a major focus of our research because data sharing is not a requirement of our applications domain.

Exokernels [17, 21] eliminate high-level abstractions to enable OS extensibility. Although this enables optimizations based on physical names, it discourages isolation because all resources exist inside a single, globally-visible namespace. This necessitates complex mechanisms to download protection policy based on high-level abstractions into low-level Exokernel protection mechanisms. Moreover, the virtual name spaces exposed by a VMM facilitate rapid swapping, since virtual-to-physical name bindings can be transparently modified by the VMM. Swapping applications is more difficult on an Exokernel system because there is no way to transparently remap library OS address translations.

Nemesis [19] shares our goal of eliminating application resource "crosstalk". Nemesis adopts a similar approach to us, pushing most kernel functionality (including protocol processing and device drivers) into application space. Our systems differ in that Nemesis is not designed to sandbox untrusted code; Nemesis applications share a global file systems and a single virtual address space.

6.4 Virtual hosting platforms

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 operating system or application, and therefore cannot provide the same degree of isolation as a VMM.

Two commercial VMMs provide virtual hosting services: VMWare's ESX server and IBM's z/VM system. By allowing ourselves to change the virtual architecture and co-design the OS (para-virtualization), we believe that Denali will scale to many more VMs on similar hardware than these products. Our work also addresses resource management in the face of a large number of concurrent VMs; we are not aware of any publications on this subject related to these products.

7 Conclusions

In this paper, we argued that virtual machine monitors are well-suited to the task of hosting many untrusted applications on a single physical machine. VMMs defer the implementation of high-level abstractions and sophisticated protection policies to guest operating systems. This makes the monitor simpler, and facilitates strong security and performance isolation, at the cost of increased sharing overhead.

To scale up to a large number of virtual machines, Denali utilizes *para-virtualization*, which entails selectively modifying the virtual architecture. Using para-virtualization techniques, we co-designed a VMM and a guest operating system capable of supporting a non-trivial web-server application, which can serve over 5,379 HTTP requests per second.

Although Denali is a work-in-progress, our work thus far demonstrates that it is possible to achieve strong isolation and reasonable performance using a virtual machine monitor. We also demonstrated that some of the issues that impact scaling up the number of concurrent virtual machines demand a reconsideration of the virtualized architecture, and indeed, the co-design of the virtual architecture with guest operating systems.

References

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the Ninth USENIX Security Symposium*, August 2000.

[2] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the Symposium on Programming Language Design and Implementation*, May 1996.

[3] T. E. Anderson, D. E. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54–64, February 1995.

[4] 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.

[5] 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.

[6] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the Winter USENIX Conference*, 1995.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching, and Zipf-like distributions: Evidence, and implications, Mar 1999.

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

[9] 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.

[10] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end wan service availability. In *Proceedings of the Third Usenix Symposium on Internet Technologies and Systems (USITS01)*, San Francisco, CA, USA, March 2001.

[11] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.

[12] C. Cowan, S. Beattie, G. Kroach-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *Proceedings of the Systems Administration Conference*, December 2000.

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

[14] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the ACM SIGCOMM Conference*, Austin, TX, September 1989.

[15] Peter Druschel and Gaurav 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, October 1996.

[16] 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.

[17] D.R. Engler, M.F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource managment. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, December 1995.

[18] D. D. Redell et al. Pilot: An operating system for a personal computer. In *Proceedings of the 7th*

ACM Symposium on Operating Systems Principles (SOSP), pages 106–107, 1979.

[19] 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.

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

[21] 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.

[22] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. 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.

[23] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[24] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1999.

[25] A.K. Ghosh, A. Schwartzbard, and M. Schattz. Learning program behavior profiles for intrusion detection. In *Proceedings of the Workshop on Intrustion Detection and Network Monitoring*, 1999.

[26] B. Gold, R. Linde, R. Peeler, M. Schaefer, J. Scheid, and P. Ward. A security retrofit of VM/370. In *Proceedings of the national computer conference*. AFIPS Press, June 1979.

[27] 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.

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

[29] R.P. Goldberg. A survey of virtual machine research. *IEEE computer magazine*, 7(6), June 1974.

[30] S. Ioannidis and S.M. Bellovin. Sub-operating systems: A new approach to application security. University of Pennsylvania Technical Report MS-CIS-01-06, 2001.

[31] P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn. A retrospective on the VAX VMM security kernel. 17(11), November 1991.

[32] K.P. Lawton. Running multiple operating systems concurrently on an ia32 pc using virtualization techniques. http://www.plex86.org/research/paper.txt.

[33] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[34] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. In *Proceedings of the First Workshop on Internet Server Performance (WISP '98)*, Madison, WI, June 1998.

[35] G.J. Popek and C.S. Kline. Verifiable secure operating system software. In *AFIPS Conference Proceedings*, June 1973.

[36] J. Reumann, A. Mehra, K.G. Shin, and D. Kandlur. Virtual services: A new abstraction for server consolidation. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000.

[37] 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.

[38] P.J. Shenoy and H.M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98)*, Madison, Wisconsin, USA, June 1998.

[39] 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.

[40] VMware, Inc. Vmware virtual machine technology. `http://www.vmware.com/`.

[41] C.A. Waldspurger and W.E. Weihl. Stride scheduling: Deterministic proportional share resource management. Technical Memorandum MIT/LCS/TM-528, June 1995.

[42] D.S. Wallach, D. Balfanz, D. Dean, and E.W. Felten. Extensible security architectures for java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[43] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. 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.