Constructing Services with Interposable Virtual Hardware

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

Abstract

Virtual machine monitors (VMMs) have enjoyed a resurgence in popularity, since VMMs can help to solve difficult systems problems like migration, fault tolerance, code sandboxing, intrusion detection, and debugging. Recently, several researchers have proposed novel applications of virtual machine technology, such as Internet Suspend/Resume [25, 31] and transparent OS-level rollback and replay [13]. Unfortunately, current VMMs do not export enough functionality to budding developers of such applications, forcing them either to reverse engineer pieces of a black-box VMM, or to reimplement significant portions of a VMM.

In this paper, we present the design, implementation, and evaluation of μDenali, an extensible and programmable virtual machine monitor that has the ability to run modern operating systems. μDenali allows programmers to extend the virtual architecture exposed by the VMM to a virtual machine, in effect giving systems programmers the ability to dynamically assemble a virtual machine out of either default or custom-built virtual hardware elements. μDenali allows programmers to interpose on and modify events at the level of the virtual architecture, enabling them to easily perform tasks such as manipulating disk and network events, or capturing and migrating virtual machine state. In addition to describing and evaluating our extensible virtual machine monitor, we present an application-level API that simplifies writing extensions, and we discuss applications of virtual machines that we have built using this API.

1 Introduction

Virtual machine monitors (VMMs) such as VM/370[8], Disco[6], and VMware [35] have demonstrated that is feasible to implement the hardware interface in software efficiently, making it possible to multiplex several virtual machines (VMs) on a single shared physical machine. Researchers have recognized that VMMs are a powerful platform for introducing new system services, including VM migration [25, 31], intrusion detection [18], trusted computing [17], and replay logging [13]. These services all exploit the unique ability of a VMM to observe and capture all of the events and state of a complete software system, including that of the virtual hardware, the operating system, and applications.

In addition to having this "whole-system" perspective, VMMs have the advantage that the interface they expose is simple in comparison to a operating system API. Virtual hardware events such as disk reads and writes, MMU faults, and network events have a narrow and stable interface, whereas operating systems tend to expose a large number of semantically complex system calls. The virtual hardware interface is an ideal place for deploying many kinds of services: the "whole-system" perspective makes these services powerful, and the simple virtual hardware interface makes them easy to build and able to operate on legacy software.

1.1 The VMM as a Service Platform

A VMM is a compelling platform for deploying an interesting class of system services. Unfortunately, today's VMMs provide precious little support for developing new *virtual machine services*. Because VMMs were not designed to be programmable or extensible, developers have had to reverse-engineer pieces of black-box VMMs to discover or expose the interfaces they need [25], or in extreme cases, they have had to reimplement significant portions of the VMM [13]. Worse, because there is no standard interface or extension framework supported by today's VMMs, VM service authors have not been able to cooperate with each other or re-use developed components. For example, ReVirt's replay ability [13] and Hypervisor-based fault tolerance [5] are both based on a similar logging primitive, but it would be difficult for these two projects to share this common functionality.

The current state of virtual machine services is similar to the distributed computing era before the advent of standardized transport protocols and remote procedure calls [4]. In this era, programmers used a variety of ad-hoc, home-grown techniques and services to communicate across machine boundaries, resulting in brittle, unreliable, and non-interoperable systems. Standardized transport and RPC allowed distributed systems programmers to focus on the logic of their systems, relying on underlying plumbing to solve common issues of reliable communications, naming, and type marshalling.

In this paper, we attempt to advance the state of VM service construction by exploring two questions. First, what should the programmatic interface exposed by VMMs to VM services look like? Second, what structure and mechanisms within a VMM are needed to support this interface well?

To address the first question, we propose a high-level software toolkit that allows programmers to build services in one VM that **interpose** on the events generated by another VM's virtual hardware devices, or to **extend** the virtual hardware exposed to virtual machines by implementing new virtual hardware devices. These two abilities (interposition and extension) allow programmers to develop software services that manipulate the virtual machine interface without worrying about the underlying virtualization mechanisms and plumbing. Because our toolkit exposes a well-defined API, extensions and interposition services would work on any VMM that exposes the same API.

To address the second question, we describe the design and implementation of a virtual machine monitor that supports virtual device interposition and extension. Our VMM, which we call μDenali, is built around a flexible virtual hardware event routing framework that borrows significant pieces of its design from Mach ports [12]. In our framework, virtual hardware events are defined as typed messages, and virtual hardware devices are simply sets of ports. Interposition is achieved by re-routing messages from one virtual machine to another. Extensibility is achieved by allowing a VM to expose ports that send and receive appropriately typed messages.

The remainder of this paper is structured as follows. In Section 2, we briefly describe the Denali VMM [36], which we modified to build μDenali. Section 3 gives an architectural overview of μDenali, describing the basic structure of the system and a high-level view of the extensibility and interposition interface which it exposes. In Section 4, we focus on the port-based routing framework within μDenali, and we describe the virtual hardware underlying VMs in terms of the message types and ports that define each virtual device. In Section 5, we describe a number of virtual hardware device extensions and virtual machine services which we have implemented. We present an evaluation of μDenali in Section 6, we discuss related work in Section 7, and we conclude in Section 8.

2 Denali Overview

The Denali isolation kernel [36] is an x86-based virtual machine monitor whose goal is to support a large number of concurrent virtual machines. Denali is a type-I VMM, meaning that it runs directly on physical hardware, as opposed to type-II VMMs (such as VMware workstation [35] or user-mode Linux [11]) which run as applications on a host operating system.

Denali relies on a technique called para-virtualization to enhance its performance, scalability, and simplicity. Rather than exposing a virtual architecture that faithfully reproduces the underlying physical architecture, Denali strategically modifies the virtual architecture, retaining the performance advantages of direct instruction execution but modifying key features of the virtual architecture such as interrupt processing, handling non-virtualizable instructions, and timers.

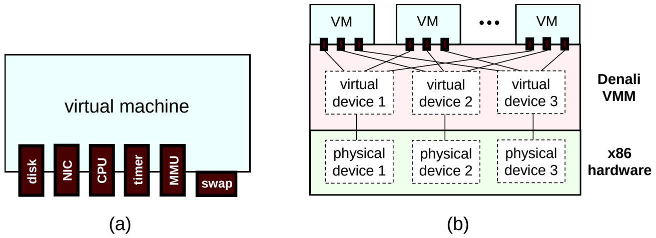


Figure 1: **Denali virtual machine monitor architecture.** The "old" pre-extensible architecture. Each virtual machine interacts with the virtual machine monitor through virtual hardware devices; each hardware device is hard-wired to a virtual device implementation inside the Denali kernel.

The Denali implementation has progressed significantly since what was reported in [36]. Most notably, because of the addition of a virtual MMU device, Denali is now able to run full-fledged operating systems in a manner similar to Xen [2]. Most modern OSs run on multiple architectures, isolating the architecture-dependent pieces of the implementation from the architecture-independent pieces. We successfully ported the NetBSD operating system to Denali by implementing architecture-dependent components appropriate for the Denali virtual architecture. Because of differences between our virtual MMU and the x86 MMU, we cannot run x86 binaries directly, but with recompilation we can run NetBSD and any of its applications.

2.1 Denali's Architecture

Figure 1 illustrates the Denali virtual architecture. The interface between each virtual machine and Denali is a set of virtual hardware devices. Within the Denali VMM, these virtual devices are "hard-wired" to virtual device implementations that (1) multiplex the virtual devices of the many VMs onto their physical counterpart (including namespace virtualization) and (2) implement physical resource management policies. Denali supports the following virtual devices:

Virtual CPU: The virtual CPU executes the instruction streams of virtual machines, emulates privileged instructions, exposes virtual interrupts, and handles virtual programmed I/Os. The virtual CPU also exposes a number of purely virtual registers, such as a register which contains the current wall-clock physical time.

Virtual MMU: The Denali virtual MMU exposes a software-loaded TLB with a fixed number of virtual address-space IDs (ASIDs). The virtual MMU is a completely different abstraction than the underlying x86 hardware-based page tables; we made this design choice to simplify the virtualization of virtual memory. Borrowing terminology from Disco [6], we refer to true physical memory as *machine memory*, virtualized physical memory as *physical memory*, and virtualized virtual memory as *virtual memory*. The virtual MMU allows guest OSs to implement multiple virtual address spaces, and to page or swap between virtual memory and physical memory. Denali itself overcommits physical memory and implements swapping between machine and physical memory transparently to VMs.

Virtual timers: Denali exposes a virtual timer that supports an "idle-with-timeout" instruction, allowing virtual machines to relinquish their virtual CPUs for a bounded amount of time.

Virtual network: The Denali virtual NIC appears to guest OSs as a simple Ethernet device. The NIC supports packet transmission and reception at the link level.

Virtual disk: Stable storage is exposed to guest OSs as a block-level virtual disk. Guests can query the size of the disk, and read and write blocks to disk.

Because it supports neither extensibility nor interposition, implementing the VM services that have appeared in recent literature would be difficult using Denali. For example, to checkpoint and migrate a virtual machine, all virtual device state must be captured, including virtual registers, physical memory pages (whether resident in machine memory or swapped to physical disk by Denali), virtual MMU entries, virtual disk contents, and any virtual disk operations that are in flight. However, none of this state is exposed outside of the VMM. In the next section of this paper, we describe the architecture of μDenali, a significant reimplementation of the Denali VMM which supports both extensibility and interposition to support these kinds of services.

3 μDenali: an Extensible VMM

A VMM provides three basic functions: it virtualizes the namespace of each hardware device in the physical architecture, it traps and responds to virtual hardware events to implement virtual hardware device abstractions, and it manages the allocation of physical resources to virtual machines. In the Denali VMM, these three functions were co-mingled. The major insight of μDenali is that these functions can be separated from each other, and by doing so, virtual hardware event interposition and extensibility become a simple matter of routing one VM's virtual hardware events to another VM.

Physical resource management: μDenali borrows significant code from Denali to interact with and manage physical devices. We utilize low-level code and device drivers provided by the Flux OSKit [15], and we implement our own global resource management policies across VMMs, such as round-robin CPU scheduling across VMMs, fair queuing of received and transmitted Ethernet packets, and a static allocation of physical disk blocks. These global resource management policies, which we do not allow to be extended or overridden, provide performance isolation between VMs.

Device namespace virtualization: Each virtual device in the Denali architecture is capable of generating and receiving several device-specific events. For example, virtual disks receive disk read and write requests, and generate request completion events. As another example, virtual CPUs generate interrupts and faults. In μDenali, we associate a typed message with each of these device-specific events. Implicit in the message type definitions are the namespaces associated with devices: disk event messages contain block offsets, and CPU faults contain the memory address of the faulting instruction.

Virtual hardware event trapping and routing: Denali directly executes the instructions of virtual machines, but privileged instructions and virtual hardware device interactions are trapped to and emulated by the VMM. In μDenali, we borrow Denali code to trap on these events, but instead of directly handing them off to virtual device implementations within the VMM, we encode these events within typed messages and route the messages to an endpoint. μDenali contains a routing infrastructure which associates destination **ports** with events generated by the hardware devices of each VM. Default virtual hardware devices within μDenali have ports, as do special **interposition devices** associated with each VM. By binding a hardware device of one VM (the child) to the interposition device of another (the parent), the parent VM gains the ability to interpose on the child's virtual device. The parent VM can either reroute the events back to μDenali's default virtual device implementation, or handle them itself, in which case the parent becomes an extension to the virtual architecture of the child.

Figure 2 illustrates the μDenali architecture. In Figure 2(a), we show the structure of a virtual machine. Each VM sees a collection of virtual hardware devices with which it interacts, similar to Denali. Unlike Denali, a μDenali VM can also interact with its interposition device. A parent VM receives virtual hardware events from children VMs on which it interposes, and the parent can send response events (either to μDenali or to the child) over this device. Additionally, the interposition device exposes various control functions to a parent, such as the ability to instantiate a new child VM, suspend or

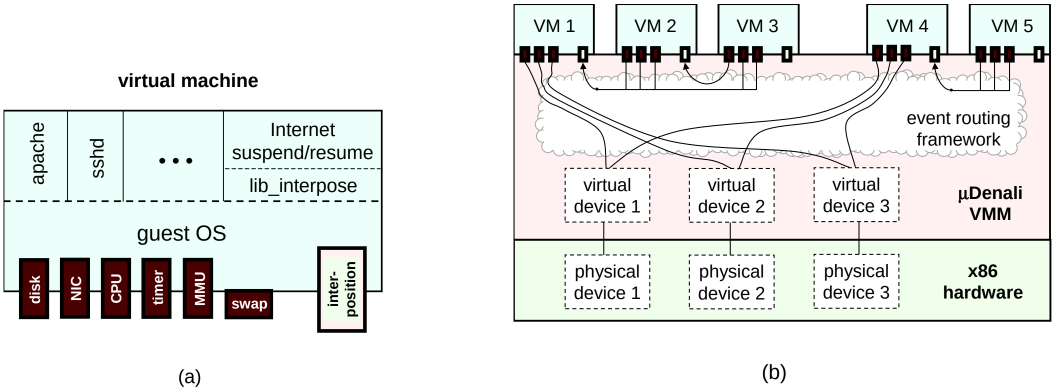


Figure 2: **μDenali virtual machine monitor architecture.** (a) A new virtual hardware device, the "interposition device", is visible to each VM. The guest OS exposes events routed through this interposition device to user-level virtual machine services, like Internet suspend/resume, with the help of a C library called *lib_interpose*. (b) A VM's virtual hardware devices can be bound to other VMs' interposition devices, or to default virtual devices implemented in the VMM. An event routing framework (based on Mach ports [12]) routes virtual device events to appropriate endpoints, enabling both event interposition and virtual device extensions.

resume execution of a running child VM, or extract the hardware state of a child VM from μDenali.

The guest operating system within a virtual machine can choose to handle virtual hardware and interposition device events as it sees fit. In our port of the NetBSD operating system to μDenali, we have written NetBSD device drivers for the "regular" virtual hardware devices, and we have exposed the interposition device to user-level applications through a high-level interposition library written in the C programming language. Using this interposition library (*lib_interpose*), user-level applications can implement virtual machine services, such as Internet Suspend/Resume, which manipulate the device state and events generated by other VMs.

In Figure 2(b), we show the μDenali VMM architecture. Similar to Denali, μDenali runs directly on x86 hardware, and contains a set of default virtual device implementations. μDenali also contains an event routing framework, which manages bindings between ports associated with virtual hardware devices of running VMs, and routes typed messages according to these bindings. The figure shows five virtual machines. VM1 is the parent VM of two children (VM2 and VM3). VM1 receives and handles all events generated by all virtual devices of VM2, and some virtual devices of VM3. VM2 is a partial parent of VM3, receiving events from the remaining virtual devices of VM3. VM1 has no parent, meaning that all of its virtual device events are bound to default virtual devices within μDenali. Similarly, VM4 is the parent of VM5.

3.1 The NetBSD Interposition Library

The μDenali interposition device defines the set of extensibility and interposition operations that a parent VM can perform on a child. The interface to this device consists of a set of downcalls that a parent initiates when it wishes to perform an action on a child (for example, to instantiate a new child VM), and a set of events and responses that are exchanged between the parent and μDenali when an event of interest happens within the child VM.

The interposition device interface is implemented in terms of architecture-level programmed I/O operations and interrupts. Our NetBSD interposition library, with the help of the NetBSD guest OS, exposes this interface to applications in terms of function calls in the C programming language. In the remainder of this section, we describe the key components of the interposition device interface by showing fragments of the interposition library API.

One key design decision we made was to focus on *local* extensibility and interposition—that is, the interposition device only exposes events generated by child virtual machines. We do not expose global events, such as μDenali scheduler decisions or machine memory allocation and deallocation events. Because of this, the μDenali extensibility framework does not suffer from security issues plaguing extensible operating systems, which permit extensions to modify global system behavior [3]. μDenali extensions are isolated from the μDenali VMM and from other VMs as a side-effect of being implemented inside their own virtual machine, and extensions can only affect children VMs.

3.1.1 Virtual Machine Control

The control API allows a parent VM to create, destroy, start, and stop child virtual machines. Figure 3 shows a subset of the NetBSD C library functions and

```
typedef struct {

// the swapDevice provides backing store for the
// child's physical pages when Denali swaps.
SwapDevice *swapDevice;

// a Disk provides block-addressable storage.
Disk *disk;

// (other virtual devices omitted, for brevity.)
} VirtualMachine;

// create and destroy child VMs.
int createVM(VirtualMachine *vm, char *macAddr);
int destroyVM(VirtualMachine *vm, char *macAddr);

// associate a local /dev/virtethX block device
// with a new virtual Ethernet in the child.
int createEthernet(VirtualMachine *vm);

// suspend and resume a child VM.
int suspendVM(VirtualMachine *vm);
int resumeVM(VirtualMachine *vm);
```

Figure 3: **Virtual machine control functions.** The VM control portion of the interposition API allows parent VMs to create, destroy, start, and stop child VMs.

structures associated with these control operations.

The create and destroy operations permit a parent to dynamically assemble a child out of constituent virtual hardware devices, and then cause μDenali to instantiate and begin executing the assembly. For each device type, the parent must either provide callback functions to handle messages associated with that device, or provide an alternate routing port (such as a parent VM's port, or μDenali default device ports) for the device to bind to.

The suspend and resume operations act on already active virtual machines. These commands are often used in conjunction with other interposition commands. For example, a checkpoint application would stop a virtual machine before extracting the checkpoint to ensure that it obtains a consistent snapshot of virtual device state.

3.1.2 I/O Device Interposition

The I/O device API allows a parent VM to interpose on virtual device events raised by child VMs, and to either re-route or respond to those events. This API allows parents to interact with virtual disks, Ethernets, and swap store backing (virtual) physical memory. As in Denali, μDenali device events are simplified idealizations of the underlying physical devices. For example, the virtual disk interposition interface supports two principle operations: reading and writing disk blocks. Figure 4 shows a representative subset of the I/O device API.

```
// the virtual Disk device callback functions
typedef struct {

  // the child generated a write event.
  int (*diskWrite)(char *buffer, int offset,
                   int num_sectors);

  // the child generated a read event.  If the
  // parent chooses to handle the event, it
  // puts the appropriate data in "buffer".
  int (*diskRead)(char *buffer, int offset,
                  int num_sectors);

  // the child is asking the disk to report
  // how many sectors it contains.
  int (*getSectorSize)(void);

} Disk;
```

Figure 4: **I/O device interposition functions.** The I/O device interposition API permits parents to interpose on and respond to their children's device operations. In this figure, we show only the interface associated with the virtual disk; other devices have similar interfaces.

We chose to special-case the interposition interface to children's virtual Ethernet devices. The interposed Ethernet is exposed to the parent as a network device within NetBSD, (e.g., `/dev/virteth0`). This allows us to reuse existing networking software like NAT proxies, routers, and intrusion detection systems within a virtual machine service. From the point of view of the parent VM, it is connected to its child over a a dedicated network interface.

3.1.3 Exposing μDenali Internal State

In practice, not all virtual machine state can be directly exported through the interposition device. For performance reasons, we chose to cache run-time state such as hardware registers, MMU mappings, in-flight virtual device operations, and the status of swap buffers inside the μDenali kernel itself. However, to implement services such as checkpoint and migration, parent virtual machines must be able to flush and access this state upon request. In practice, only "hard" state must be saved. For example, Ethernet packets that are queued for transmission or delivery inside μDenali can be discarded, because doing so is consistent with the best-effort semantics of Ethernet networks.

Because much of this run-time state is platform specific, μDenali encapsulates it inside in an opaque data structure. Extensions such as debuggers that require low-level access to platform-specific structures must manipulate this run-time state with knowledge of μDenali's encapsulation syntax. Figure 5 shows a subset of the flush and state-capture functions that are exposed by the interposition device.

```
// functions to flush cached state from
// Denali, and extract/restore kernel
// internal associated with a VM.

int flushSwap(VirtualMachine *vm);

int extractMMUmappings(VirtualMachine *vm,
                       MMUState *ms);
int restoreMMUmappings(VirtualMachine *vm,
                       MMUState *ms);

int extractCPUstate(VirtualMachine *vm,
                    CPUState *cs);
int restoreCPUstate(VirtualMachine *vm,
                    CPUState *cs);
```

Figure 5: **Flushing and extracting VMM state.** μDenali internal state associated with a VM can be flushed and extracted using this part of the API. Here, we only show a portion of the full API.

3.1.4 Tracking Non-Determinism

Tracking and logging non-deterministic events is necessary to implement replay services such as ReVirt [13]. We have not yet finished the implementation of this functionality, but we have a complete design that takes advantage of μDenali's message routing layer.

To facilitate logging and replay services, μDenali must expose two types of information: the precise instruction-level timing of asynchronous events (such as virtual interrupts), and the content of events that are obtained from external input, such as user keystrokes or network packets. Both types of information are readily available in the μDenali architecture. Asynchronous events are defined to occur when the message routing infrastructure delivers an event to a VM, and the content of external events can be observed through interposition. For example, an external Ethernet packet arrives when a virtual Ethernet packet message is delivered to a virtual Ethernet device, and the content of that message can be observed and logged by a parent through interposition.

In our design, μDenali assists with the logging of non-deterministic events by recording the timing of messages delivered to a child VM, and periodically flushing this log to the parent's interposition interface. Each log entry contains a three-tuple: a unique event ID, the event type (which is simply the type of the message delivered to the child port), and a delivery time. As with ReVirt, delivery time is measured by a software instruction counter [27] which includes the instruction address and a counter of the number of backward branches executed during the lifetime of the child VM.

Replay requires the ability to interrupt a child VM before the execution of a logged non-deterministic event. To support this, the parent VM must be able to pass a previously recorded log file to μDenali, which uses the log to trap the child VM at the precise moment that a logged non-deterministic event should occur. Our implementation of logging and replay is underway, but not complete.

3.2 Summary

In this section, we described the architecture of the μDenali extensible virtual machine monitor, and presented the high-level interposition and extension interface that it exposes. This interface, which can be accessed through an application-level library implemented in the C language, provides programmers with a powerful and natural way to implement new virtual machine services. In the following section of the paper, we drill down into the design of the message routing infrastructure inside μDenali.

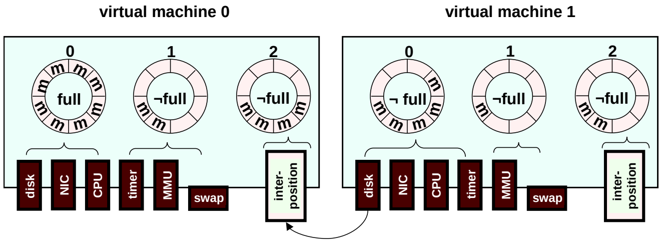
4 Event Routing in μDenali

As described in the previous section, μDenali contains an event routing framework that is responsible for receiving, routing, and delivering virtual device events. In essence, the event routing framework is a virtual bus that provides a communication channel between a VM's virtual devices and the implementation of those devices.

Our routing framework design was inspired by Mach ports and messages [12]. A port is an abstraction of a protected communication channel that facilitates the reception and transmission of typed messages and port capabilities. Each virtual device in each virtual machine has a set of ports associated with it, and ports correspond to operations supported by virtual devices. By binding a virtual device port from a child VM to the interposition device port of a parent VM, events generated by that child's virtual device are converted into typed messages and delivered to the parent. Figure 6 illustrates the routing framework design.

4.1 Port Capabilities and the Port Table

Ports are protected by capabilities. Like Mach, μDenali defines three types of capabilities: the receive right allows the holder to receive messages delivered to that port, the send right allows the holder to send messages to the port, and the send-once right allows the holder to send a single message to the port (for example, to allow a message recipient to send a reply). A capability can be transmitted over ports in order to grant a privilege to the recipient. Only send rights can be copied; if a send-once right is passed in a message, the sender loses that right. The holder of a receive right (defined to be the port owner) may create send and send-once capabilities for that port.



(a)

VM 1's port table

port name	port rights	port type	refcount	ringbuf
0	R	disk read reply	0	VM1:0
1	R	disk write reply	1	VM1:0
2	S	disk read request	1	VM0:2
3	S	disk write request	1	VM0:2

(b)

Figure 6: **μDenali port tables.** (a) VM1's virtual disk is managed by VM0. (b) A subset of VM1's port table. Because VM1 can generate virtual disk read and write requests, VM1 has send rights on the ports associated with those operations (ports 2 and 3). VM0 manages the associated receive ports, and uses its ring buffer #2 to buffer messages arriving on it. Because VM0 manages the disk, it needs to be able to send disk read and write replies back to VM1. VM1 has receive ports (ports 0 and 1) to receive these replies, and it uses its ring buffer #0 to buffer disk reply messages.

The μDenali VMM maintains a table of port capabilities on behalf of each virtual machine. Because this port table is managed inside the VMM, port capabilities are unforgeable, similar to UNIX file descriptors. **From the point of view of μDenali, a virtual machine is simply a port table.** Much like a MCS capability list [10] or Hydra local name spaces [37], a port table simultaneously defines the operations that can be performed on an object (virtual devices) and the namespace in which those objects are embedded (the virtualized namespaces of virtual devices).

When a parent VM creates a new child VM, the parent sets up the initial port table of the child. To do this, the parent creates individual ports in the child's port table, grants port capabilities as necessary, and binds the ports as appropriate. A parent can bind a child's port to its own interposition device, to the interposition device of another VM for which it has a port send capability, or to the port associated with the default virtual device implementations inside μDenali.

4.2 Port Queues and Message Buffers

Like hardware buses, the μDenali event routing framework itself never stores messages. All message queuing must be implemented by the virtual device which owns the port which receives the message. To accomplish this, each virtual machine maintains one or more ring buffers associated with its ports. When a message arrives on a port, μDenali atomically delivers and advances the ring buffer. If the ring buffer is full, all ports associated with it enter the "full" state, and error messages are returned to source ports on future message delivery attempts. When the ring buffer is drained by the virtual machine, the receive ports associated with it leave the full state, and notifications are sent to all send ports bound to them.

Because the event routing framework has no storage, providing atomic message delivery is simple: the routing framework promises either successful delivery, or an immediate error. This lack of storage also means that the routing framework itself is stateless, meaning that it need not be involved in the checkpoint or recovery of a virtual machine. Note, however, that port tables and ring buffers must be included in checkpoints.

μDenali messages contain a small amount of in-band data (16 32-byte words). For several of our devices, such as virtual MMUs, this suffices. For those that deal with larger blocks of data (e.g., virtual disks), μDenali provides an out-of-band data passing mechanism to transfer up to 64KB of data with each message. This out-of-band channel is similar in spirit and usage to direct memory access (DMA) on modern physical hardware devices.

4.3 Example Virtual Devices, Ports, and Message Types

μDenali defines a standard set of virtual devices and associated operations and events. Operations and events are exposed to virtual machines by the routing framework by ports and message types, respectively. We now discuss a few virtual devices and their ports and message types. Note that there is a one-to-one correspondence between ports and messages in μDenali's event routing framework, and function calls and arguments in the *lib_interpose* interposition library described in Section 3.1. Though we only discuss a subset of the virtual devices supported by μDenali, the other virtual devices have similar and consistent designs.

4.3.1 Virtual CPU

A VM's virtual CPU device (vCPU) is responsible for executing the instructions of the guest operating system and its applications, managing the registers of VMs, and exposing ports and messages to VMs. All virtual machines currently rely on the default vCPU implementation inside μDenali.

The vCPU shares most of its specification with the underlying x86 physical CPU. With the exception of a handful of non-virtualizable instructions [29] which have undefined (though secure) results, all unprotected instructions are available to the guest. All of the unprotected x86 registers are also available to the guest.

μDenali augments the x86 CPU with a set of purely virtual instructions and registers that have no physical counterpart, as discussed below.

The vCPU maintains each VM's port table, and also handles the transmission and reception of messages on ports. To transmit a message, a VM places the message and destination port-descriptor in a fixed memory location and invokes a virtual instruction. The vCPU device traps this instruction, type-checks the message against the port table, swizzles any port references in the message, converts the physical addresses referring to out-of-band data into machine addresses, and attempts to place the message in the receive port ring buffer. From the point of view of the VM transmitting the message, this process corresponds to a single atomic instruction (albeit a high latency one). On reception of a message, the vCPU interrupts the receiver VM and jumps to a guest-specified interrupt handler, assuming virtual interrupts are enabled.

The vCPU provides the mandatory implementation of all VMs' virtual MMU devices. We implemented this device in μDenali because it is the most frequently used device in the system, and its implementation is closely intertwined with physical page table and machine memory management. We have not yet considered whether it is possible or pragmatic to permit virtual MMUs to be interposed on or extended by parent VMs.

4.3.2 Virtual Swap Device

Interposing on the virtual swap device permits a parent VM to manage the swap file backing a child VM's physical memory pages. μDenali decides when to reclaim a machine page that is bound to a physical page of a VM, and when it does so, it generates swap events on behalf of the child. A swap device is conceptually similar to external pagers in Mach and V [38, 7]. By interposing on a swap device, a parent has the ability to checkpoint all of the physical memory of its child.

A swap device has three ports, and supports four message types. Whenever a VM generates a physical-to-machine page fault within μDenali, the VMM generates a *fault* message and sends it to the VM's fault receive port. The handler is responsible for processing the message and sending a reply back when the fault has been serviced. When μDenali needs to swap out a physical page, it generates a *swap-out* message and sends it to the swap receive port of the VM. Finally, a parent VM can generate a *flush* or *flush-all* message to flush one or all dirty physical pages back to the swap device.

4.3.3 Virtual Ethernet Device

The virtual Ethernet device manages the transmission and reception of Ethernet packets. A virtual Ethernet device has two ports, and supports one message type. When an Ethernet frame arrives for an Ethernet device, an *Ethernet packet* message is created and sent to the virtual Ethernet device's receive port. When a VM transmits an Ethernet frame, another *Ethernet packet* message is created and sent on the virtual Ethernet device's send port.

A VM's virtual CPU facilitates the creation of new virtual Ethernet devices. A parent VM can send a *create new virtual Ethernet* message to the Ethernet creation port of the virtual CPU, causing a new virtual Ethernet device to be created with a specific MAC address.

4.4 Summary

In this section, we described μDenali's event routing framework, whose design borrows heavily from Mach ports and messages. We also described the ports and message types associated with virtual devices supported by μDenali. In the next section of the paper, we describe several virtual device extensions and virtual machine services that we have implemented on μDenali.

5 Application Case Studies

In this section of the paper, we demonstrate how μDenali's extensibility and interposition mechanisms can be used to develop powerful virtual device extensions and virtual machine services.

5.1 Internet Suspend/Resume

The Internet Suspend/Resume project [25] uses the Coda file system to migrate checkpointed VMware virtual machines across a network. We have re-implemented this functionality in μDenali, albeit using the less efficient NFS rather than Coda.

Migrating a VM consists of three phases: checkpointing and gathering the complete state of the VM, transmitting this state over the network, and unpackaging the state into a new VM on the remote host. In μDenali, we use virtual device extensions to maintain and gather the state of a child VM's virtual devices (for example, a swap disk extension maintains the complete physical memory image of the child), and we rely on the state extraction routines in the interposition library to extract μDenali internal hard state associated with the child.

To transmit VM state over a network, we rely on the NFS file system. The parent stores all gathered state inside an NFS file system also accessible to the remote host on which the VM will be resumed.

We found the development of migration on top of `lib_interpose` to be straightforward. The entire implementation consists of 289 C source lines. The bulk of this complexity is dedicated to serializing C structures into a byte stream, a task handled automatically by more modern languages such as Java.

5.2 "Drop-in" Network Services

As an example of a virtual machine service that exploits network device interposition, we have created a dynamically-insertable network intrusion detection system (IDS) embedded within a virtual machine. Our system (which we call VM-snort) acts as a parent that interposes on other VMs' virtual Ethernet devices. Any network packet flowing in or out of a child VM must pass through VM-snort. We developed VM-snort using the Snort network intrusion detection system.

Because VM-snort is embedded in a virtual machine checkpoint, it is possible for a third party to create a "pre-packaged" IDS image that can be dynamically instantiated at customer sites, using the checkpoint/restore functionality we previously described. Thus, VM-snort is the virtual equivalent of a dedicated physical intrusion detection appliance.

This model of embedding a network service inside a VM is generalizable to other network services. Many active services (i.e., those that modify in addition to monitor packets) are possible, such as drop-in VPN servers, transparent Web proxy caches, or application-level firewalls. Other passive services besides VM-snort are also possible (i.e., services that only passively monitor packets), such as a dynamically deployed distributed worm detector or an Internet weather service sensor.

5.3 Continuous Rejuvenation

In practice, it is extremely difficult to remove all bugs from complex software systems. Many bugs in production systems are "heisenbugs" [20], which depend on intricate sequences of low-probability events. Other errors like memory leaks can emerge slowly, thereby defeating many testing procedures. Researchers have proposed pro-active restarts to forestall the effect of these latent errors [22]. However, simply restarting a machine creates short-term unavailability, which may be unacceptable for some services. Another alternative is to replicate the service across physical machines, but this can increase cost and administrative overhead.

By using restartable *virtual machines*, we can achieve many of the benefits of software rejuvenation without the accompanying downtime. To demonstrate this, we have constructed a service called *Apache**, which serves web requests from a (virtual) web server farm. The system consists of K child nodes, each running a standard copy of the Apache web server. The parent VM acts as a layer-3 switch, redirecting incoming requests to the child that is active at any given time. After a configurable time period, the parent VM redirects requests to the next child in the pipeline, and reboots the child[1].

*Apache** leverages μDenali's interfaces for disk, swap, and network interposition. Our implementation was greatly simplified by leveraging NetBSD's built-in NAT functionality to handle request distribution. We use an implementation of copy-on-write disks (described below) to isolate state changes inside the Apache virtual machines. The entire implementation (including COW disks) required 1131 lines of C code.

5.4 Disk and Swap Device Extensions

Because μDenali virtual swap devices are extensible, implementing novel bootloaders is simple. As a demonstration of this, we implemented a network boot loader, which fetches the child's swap image over HTTP. The boot loader saves the swap image in the NetBSD file system of the parent. When swap-in or swap-out events are generated by the child's virtual swap device, the bootloader serves the appropriate frames from this file. Our bootloader makes use of the `wget` Unix program to fetch files from Web servers. We implemented the portion of the bootloader that handles swap events and serves frames from a file; this required 94 lines of C code.

Using μDenali's disk interposition mechanism, we built a simple copy-on-write (COW) disk extension. Our COW disk extension permits multiple VMs to share a base disk image, but exposes the abstraction of a private mutable disk per VM. Like all standard COW implementations, our extension maintains a file containing differences between the base disk image and each VM's current disk image. A COW disk is useful in many situations, including efficiently creating multiple "cloned" virtual machines on a single physical machine. Our copy-on-write disk extension consists of 675 lines of C code.

We are currently working on a "time-travel" disk implementation, which records all disk updates to a log. This provides a recovery mechanism that allows a system to roll back to a previous working state.

6 Evaluation

Our evaluation explores three aspects of μDenali's performance. First, we measure the basic overhead introduced by μDenali's virtualization and extensibility. Next, we measure the performance of interposed virtual hardware devices (such as interposed disks and Ethernets), and compare it to non-interposed virtual devices. Finally, we measure the performance of our Internet Suspend/Resume and VM-snort services.

For our experiments, μDenali ran on a 3.2 GHz Pentium 4 with 1.5 GB of RAM, an Intel PRO/1000 PCI

[1] Actually, there is a delay before rebooting a virtual machine to allow existing connections to terminate.

operation	latency
native NetBSD null system call	0.38 µsec
µDenali NetBSD null system call	1.2 µsec
virtualization overhead of µDenali's virtual Ethernet (send, without interposition)	2.0 µsec
virtualization overhead of µDenali's virtual Ethernet (send, with interposition)	7.2 µsec

Figure 7: **VMM overhead.** This table compares microbenchmarks run on Linux executing directly on physical hardware, and on our ported NetBSD executing on μDenali. The "virtualization overhead of μDenali's virtual Ethernet" latencies show the overhead introduced when packets are sent through a virtual Ethernet device, excluding the physical Ethernet send costs. The cost of the interposition machinery is shown by comparing the "without interposition" and "with interposition" cases.

gigabit Ethernet card connected to an Intel 470T Ethernet switch, and an 80 GB IDE hard drive running at 7200 RPM. For any experiment involving the network, we used a 1500 byte MTU. We used the httperf [28] Web workload generation tool.

Our Ethernet card is not well-supported by conventional NetBSD on physical hardware. Therefore we compare μDenali's NetBSD performance with Linux in the results that follow.[2] We do not believe this qualitatively changes our results.

6.1 Basic Overhead

A major source of overhead for VMMs is the trapping and emulation of privileged instructions. For μDenali, these instructions include virtual hardware device programmed I/O instructions and system calls issued by a guest OS. The event routing framework in μDenali imposes additional overhead on interposed virtual device operations.

Figure 7 shows the cost of commonly used operations. Null system calls (getpid) are more than three times as expensive on μDenali as on conventional NetBSD. In addition to trapping on the original system call, μDenali must trap on the system call return. Our trap handling code is based on old Mach code, and has not been as carefully optimized as NetBSD.

Network communication is more expensive on a VMM because packet send operations must be trapped and emulated. μDenali incurs an overhead of 2.0 microseconds to send a 1400 byte packet through a (non-interposed) virtual Ethernet, relative to a physical Ethernet driver. This overhead includes the cost of a kernel trap and packet copy. For interposed Ethernets, the message routing framework within μDenali introduces an additional cost of 5.2 microseconds, which includes the cost of a message transfer, a context switch to the "parent" VMM, and message reception. This value represents an upper bound because μDenali is capable of batching multiple messages per receive operation.

	Apache, 2KB	Apache, 64KB	Apache, 128KB	Bulk TCP
Linux	6700 req/sec	2300 req/sec	614 req/sec	728 Mb/sec
µDenali NetBSD, no interposition	3200 req/sec	1400 req/sec	550 req/sec	684 Mb/sec
µDenali NetBSD, null interposed Ethernet	2200 req/sec	880 req/sec	325 req/sec	387 Mb/sec
µDenali NetBSD, Snort interposed Ethernet	1350 req/sec	344 req/sec	116 req/sec	214 Mb/sec

Figure 8: **μDenali network overhead.** Moving from left to right, these tests transition from system-call intensive to packet-intensive. μDenali performs better in the packet-intensive regime. The 3x system call penalty we previously described contributes to a substantial performance loss for small transfers, and a moderate performance loss for large transfers.

6.2 Device Interposition Overhead

In this section, we evaluate the performance of μDenali's virtual network and disk devices, and compare the performance of these devices with and without interposition. To evaluate network performance, we benchmarked the Apache Web server serving static documents of various sizes. We also measured bulk TCP throughput. Figure 8 shows results for four configurations: 1) Native Linux, 2) μDenali NetBSD without Ethernet interposition, 3) μDenali with "null" Ethernet interposition, which provides simple packet forwarding via a parent VM, and 4) μDenali with Snort running inside the interposing VM. We use the default Snort rule set. Our generated traffic stream consisted of normal (non-malicious) Web requests, so Snort performed very little logging.

Figure 9 shows the same network performance results graphically by normalizing μDenali's performance against Linux. Without interposition, μDenali's performance degrades relative to Linux as the TCP transfer size decreases. μDenali obtains 94% of Linux's bulk TCP bandwidth, but only 48% of Linux's 2k Web throughput. This discrepancy arises because TCP connection setup is more expensive on μDenali, owing to the 3X system call performance gap. For bulk TCP throughput and large Web documents, there are fewer system calls, and μDenali performs competitively with Linux.

Although the virtualization overhead is highest for short TCP transfers requests, the additional overhead

[2] Note that NetBSD running on μDenali can exploit our physical gigabit card, since our VMM interacts with the true physical device and has the correct drivers, while a NetBSD VM interacts with a simple virtual device.

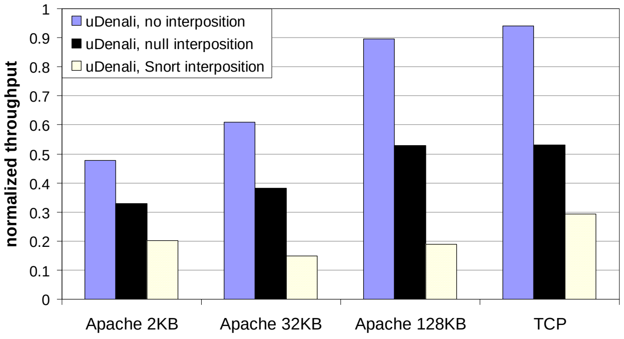


Figure 9: **Network interposition overhead:** Results are normalized against a native Linux machine. For small transfers, system call costs dominate. For large transfers, per-packet costs dominate. Interposing on network devices adds to per-packet latencies, resulting in a relative performance drop for large transfers.

	Sequential Read Throughput
Linux	45 MB/sec
µDenali NetBSD, native disk	39 MB/sec
µDenali NetBSD, interposed disk	37 MB/sec

Figure 10: **Disk interposition overhead.** This table shows the performance overhead of μDenali's virtual disks, with and without interposition.

due to interposition is *smallest* for short transfers. For example, the additional overhead for null interposition versus no interposition is 16% for 2k web requests and 41% for bulk TCP transfers. μDenali only imposes additional overhead on packet delivery, and therefore system call-intensive workloads are less affected by interposition. For Snort interposition, the per-packet overhead is large due to Snort's data copies and packet inspection routines. Snort is known to carry a high per-packet cost; even well-tuned installations can have difficulty keeping up with a heavily utilized 100 Mb/s Ethernet.

To evaluate disk performance, we used the UNIX dd utility to perform large, contiguous disk reads. In Figure 10, we compare the throughput of Linux, μDenali using a native (non-interposed disk), and μDenali using an interposed disk. The interposed disk is implemented as a file inside the parent VM's local file system. The parent's file system was initially empty, and therefore the child's blocks are likely to be nearly contiguous on disk. The sequential read workload is highly disk-bound, and therefore μDenali's numbers (with or without interposition) do not differ significantly from Linux.

6.3 Virtual Machine Services

We evaluated our Internet Suspend/Resume service using an Apache Web server as the migrating VM. We configured a Web client to submit requests for a 32 kilobyte document 200 times per second. As shown in Figure 11, the Web server satisfies these requests before and after its migration. The instability that is evident immediately after migration is due to a batch of requests that are waiting inside the VM's Ethernet ring buffer. The long migration latency (12 seconds) is primarily due to our use of NFS as a transport mechanism.

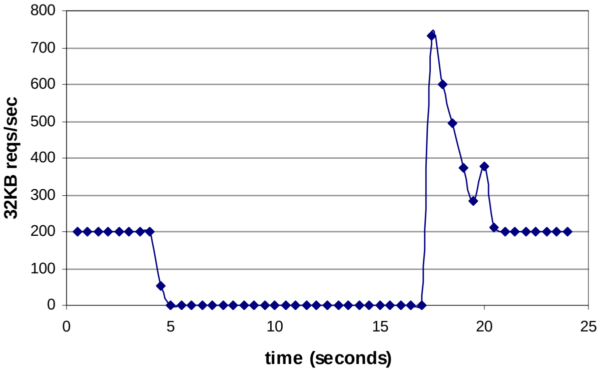


Figure 11: **Migrating an Apache server.** This timeline shows the behavior of an Apache Web service, before, during and after migration.

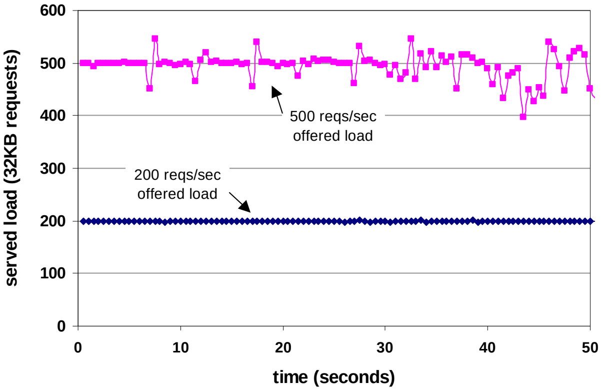


Figure 12: ***Apache** service performance.** This timeline shows the behavior of Apache before, during and after migration. The two lines correspond to two different workloads: 500 requests per second (open loop), and 200 requests per second (open loop).

We evaluated our *Apache** service using a three virtual machine Apache cluster. The parent VM was configured to garbage collect the active child VM after 10 seconds. Figure 12 shows the time-varying performance of *Apache** against offered loads of 200 requests per second and 500 requests per second for a 32KB Web document. For 200 requests per second, the throughput remains constant over time, which indicates no service disruption across VM restarts. At 500 requests per second, the system is less stable, suggesting the system is near its performance limit.

7 Related Work

μDenali is related to research into applications of virtual machines, novel virtual machine architectures, and extensible systems.

7.1 Virtual Machine Applications

Many researchers have been building novel applications that make use of virtual machine monitors. Internet Suspend/Resume [25] utilizes the checkpoint and resume functionality within VMware [35] to migrate virtual machines across a network. Sapuntzakis et al. [31] build a similar system, and show how to optimize performance with novel compression techniques. ReVirt [13] provides efficient logging and replay of virtual machines, and using this, King and Chen show how to identify sequences of events that lead to an intrusion [24]. The Stanford Collective project [30] uses VMMs to implement virtual appliances that facilitate software deployment and maintenance.

To implement these virtual machine services, the authors have had to reverse engineer relevant interfaces from a black-box VMM, or reimplement significant portions of the VMM in order to provide the interfaces they require. The goal of the μDenali VMM is to facilitate these kinds of services by exposing a carefully designed interposition and extension interface.

7.2 Novel Virtual Machine Monitors

Several research projects have focused on building novel virtual machine monitors. The Denali isolation kernel [36] relies on paravirtualization to implement lightweight virtual machines; μDenali is an enhancement to Denali. Xen [2] provides similar functionality to Denali, and focuses on providing high performance and strong isolation. Many user-level ports of Linux exist, including UMLinux [11]. Commercial virtual machine monitors have existed for several decades for mainframe computers [8], and have recently begun to achieve widespread usage on desktop workstations [35]. None of these virtual machine monitors provide comparable extensibility and interposition abilities to μDenali.

7.3 Extensibilty and Interposition

Numerous systems have injected novel functionality at the hardware interface. The storage device interface has been particularly fruitful. Petal virtual disks [26] offer a block-based 64-bit storage abstraction implemented on a cluster of workstations. Logical disks [9] provide an abstract disk interface based on logical block numbers and block lists, designed to support many different file system implementations. These systems do not attempt to be extensible or complete: they provide a fixed implementation of one virtual hardware device.

Other systems have explicitly dealt with adding extensibility to the hardware interface. User-mode pagers [7, 38] can be viewed as interposing on the implementation of the memory abstraction exposed to processes or operating systems. In particular, μDenali's virtual swap device closely resembles Mach's extensible paging for memory regions. μDenali moves beyond these systems to allow interposition on a much more complete spectrum of hardware abstractions.

Alpha PALCode [34] provides the ability to modify or extend the behavior of the Alpha instruction set architecture. Although this is a powerful primitive, PALCode faces several severe restrictions. PALcode is designed for small, low-level handlers such as TLB refills. PALCode is not suitable for implementing complex abstractions such as copy-on-write disks. A second limitation is that PALCode offers no abstraction: PALCode routines are architecture-dependent **and** implementation dependent. Finally, PAL instructions are differentiated from normal instructions in the instruction set, and therefore, it is impossible to interpose on arbitrary functionality.

The Java [19] virtual machine architecture is similar to μDenali at a high level, as it exposes a hardware-like architecture that is backed by a software-based implementation. As a result, Java VMs can support novel implementations, such as a distributed virtual machines [33]. However, JVMs are less complete than VMMs. They do not virtualize I/O devices such as Ethernet adapters or disks. As a result, JVMs do not support software systems with non-Java components. Additionally, applications such as migration that require the clean extraction of all I/O device state will be no easier to implement in a JVM than in a conventional OS.

μDenali's use of interposition as a means for achieving extensibility has been used by many previous systems, including interposition agents [23], Fluke's recursive virtual machines [16], and Mach's interposition enabled ports [12]. μDenali differs from previous systems in that we apply interposition to the virtual hardware interface exposed by a VMM. Thus, the class of applications and services enabled by μDenali is fundamentally different from previous systems, whose aim has been to interpose on the user/kernel boundary of a conventional operating system.

μDenali's architecture is similar to that of many microkernel-based operating systems, such as Mach [1] and L4 [21]. μDenali's port-based event routing framework is similar to the IPC mechanisms in these systems. Modern microkernels are able to run entire operating systems inside user-mode servers, including L^4linux [21]. μDenali differs from microkernels in many respects, providing the ability to extract full virtual hardware state, the ability to interpose on all virtual hardware events, and (currently in design only) the ability to log and reply non-deterministic events. In spirit, there is little difference between a microkernel and a VMM; both expose a virtual machine architecture. In practice, though, microkernels have been designed to provide extensible OS services such as user-mode pagers and file systems, whereas virtual machine monitors are being used to provide whole-machine services such as consolidation and migration.

There has been considerable past research into extensible operating systems [3, 14, 32]. SPIN [3] allows untrusted operating system extensions written in a typesafe language to be downloaded into the operating system. Unlike SPIN, we do not attempt to graft extensions into our kernel to override global system policies and mechanisms, but rather concentrate on extensions local to a single virtual machine. The Exokernel [14] allows programmers to build library operating systems that are safely multiplexed on a thin OS layer that exposes physical names. Instead of exposing physical names, μDenali exposes virtual names, sacrificing some performance to virtualization overhead in return for simplicity and potentially stronger isolation.

8 Conclusions and Future Directions

Virtual machine monitors have proven ideal for implementing a variety of system services, including migration, intrusion detection, and replay logging. All of these virtual machine services leverage the unique ability of a VMM to observe and interpose on the functionality and state of a complete software system. We believe there are many compelling applications of VMMs that are waiting to be discovered.

To facilitate these new services, we have redesigned the Denali VMM with the explicit goal of extensibility in mind. The resulting VMM, μDenali, has been architected to allow parent virtual machines to interpose functionality on behalf of their children. μDenali is structured on top of an event routing framework inspired by Mach ports. Using a high-level C library that exploits the extensibility features of μDenali, we implemented several virtual device extensions and virtual machine services that run in our framework, including Internet Suspend/Resume, a “drop-in” network intrusion detection virtual appliance, and a continuous rejuvenation framework for the Apache web server.

An exciting aspect of μDenali is that its extensibility mechanisms open up several avenues for future research. We are particularly interested in two future directions: using μDenali’s checkpoint primitives to simplify the management of machines and networks, and using ReVirt-style logging to monitor for bugs like race conditions, which are otherwise difficult to analyze.

9 Acknowledgements

We are grateful for the feedback of our shepherd, Mendel Rosenblum, and our reviewers. We would also like to thank Brian Bershad, Ed Lazowska, and Hank Levy for their thoughts. Brian Youngstom’s support in our lab was invaluable. This work was supported in part by NSF Career award ANI-0132817, funding from Intel Corporation, and a gift from Nortel Networks.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, 1986.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, Bolton Landing, NY, October 2003.

[3] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Dec 1995.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

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

[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 Systems Principles*, October 1997.

[7] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

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

[9] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles (SOSP’93)*, Asheville, NC, December 1993.

[10] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[11] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[12] Richard Draves. A revised IPC interface. In *Proceedings of the USENIX Mach Conference*, October 1990.

[13] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[14] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[15] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin 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.

[16] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, Shantanu Goel, and Steven Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[17] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, Bolton Landing, NY, October 2003.

[18] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Tenth Annual Network and Distributed Systems Security Symposium*, San Diego, CA, February 2003.

[19] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[20] Jim Gray. Why do computers stop and what can be done about it ? In *Proceedings of the 5th Symposium on Reliablity in Distributed Software and Database systems*, January 1986.

[21] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[22] Y. Huang, C. Kintala, and N. Kolettis. Software rejuvenation: analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995.

[23] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*, Asheville, NC, December 1993.

[24] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, Bolton Landing, NY, October 2003.

[25] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the fourth IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY, June 2002.

[26] 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.

[27] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, April 1989.

[28] 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.

[29] 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*, August 2000.

[30] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)*, October 2003.

[31] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[32] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith Smith. Dealing with disaster: Surviving misbehaved kernel extensions, October 1996.

[33] Emin Gun Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, Kiawah Island, SC, December 1999.

[34] Richard L. Sites. Alpha architecture reference manual, 1992.

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

[36] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[37] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[38] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, Texas, November 1987.