Rethinking the Design of Virtual Machine Monitors

To overcome the poor scalability and extensibility of traditional virtual machine monitors that partition a single physical machine into multiple virtual machines, the Denali VMM uses paravirtualization to promote scalability and hardware interposition to promote extensibility.

Andrew Whitaker
Richard S. Cox
Marianne Shaw
Steven D. Gribble
University of Washington

A *virtual machine monitor* is a software system that partitions a single physical machine into multiple virtual machines. Traditionally, VMMs have created a precise replica of the underlying physical machine. Through faithful emulation, VMMs support the execution of legacy guest operating systems such as Windows or Linux without modifications. However, traditional VMMs suffer from poor scalability and extensibility.

Over the past several years, our research group has developed the Denali VMM (denali.cs.washington.edu/), working from the premise that it is both possible and useful to consider a virtual machine abstraction that differs from a physical machine. The two major results from this effort are *paravirtualization* and *hardware interposition*.

- In paravirtualization, the virtual hardware architecture differs from the underlying physical architecture. We have leveraged this characteristic to construct a scalable VMM that supports hundreds of concurrently executing virtual machines.
- Hardware interposition lets programmers extend the VMM with new implementations of virtual hardware components such as virtual disks and Ethernet devices. These new hardware components can differ dramatically from native devices. For example, a new virtual disk implementation could provide encryption or access a network storage device. Higher-level services can also be realized, such as the ability to migrate a running virtual machine.

VMMs provide many advantages: They are simple, encapsulate a complete software system, and support execution of multiple legacy operating systems on a single machine.

VMM ADVANTAGES

VMMs have a long history. IBM conceived the technology in the 1960s and achieved notable success with its VM/370,[1] which served as both a time-sharing system and a platform for operating system development.

Recently, VMMs have experienced a rebirth of popularity, due in large part to the success of the VMware virtual machine monitor for the x86. Buoyed by this success, research groups are exploring innovative ways to apply virtual machine technology.

The continued success of VMMs suggests that the technology possesses inherently useful traits. In our view, several factors contribute to the current popularity of VMMs:

- *Simple implementation.* Compared to a full-blown operating system like Linux or Windows, a VMM has a comparatively simple implementation. VMMs achieve simplicity by eschewing the implementation of high-level abstractions like TCP/IP sockets and file systems. This simplicity makes VMMs well suited

0018-9162/05/$20.00 © 2005 IEEE

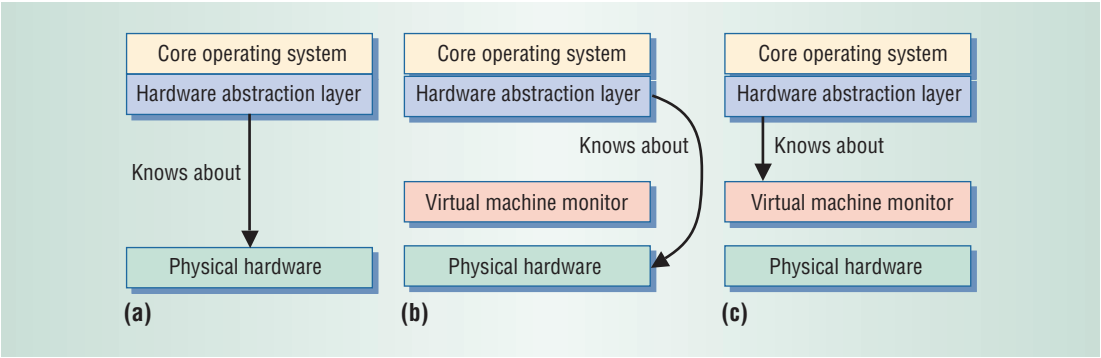

Figure 1. System architecture comparison. (a) On a conventional operating system, the hardware abstraction layer is the only OS subsystem with knowledge of the underlying physical hardware. (b) A traditional VMM exposes an unmodified replica of the physical hardware architecture. (c) A paravirtualized VMM exposes a modified hardware architecture.

for addressing system reliability and security issues.[2] In addition, their simplicity makes VMMs easier to extend and modify than traditional operating systems. For example, the Disco project used a VMM to provide system services for CC-NUMA machines.

- *Whole-system services*. A virtual machine captures a complete software system, including the operating system and its applications suite. This is important because many services of interest transcend traditional encapsulation mechanisms such as address spaces or OS processes. For example, virtual machine migration lets a user transfer a complete working environment to another physical machine.
- *Support for legacy guest operating systems*. The ability to run multiple legacy OSs on a single machine has proven useful over time. In the 1970s, this capability found application in time-sharing across users and for testing new operating system functionality. Today, system administrators use this capability to consolidate several underutilized servers onto a single machine and to enforce isolation for untrusted or insecure code.
- *Tolerable performance*. Historically, VMMs have suffered the drawback of slow performance relative to conventional system architectures. As processor speed has increased, however, this virtualization penalty has become tolerable in many settings. In addition, recent advances in VMM design[3] have driven the cost of virtualization still lower.

The Denali research group seeks to leverage VMMs to address problems in areas such as security, reliability, and system administration. During this process, we have discovered limitations that plague conventional VMMs. We have modified the Denali VMM to overcome two such challenges: scalability and extensibility.

SCALING A VMM

Scalability refers to the ability to run many virtual machines on a single physical machine. Two observations motivated our interest in scalability. First, application domains have emerged that require minimal or sporadic processor time. For example, a network measurement service inside the PlanetLab infrastructure may only require a CPU slice every few seconds. Second, Moore's law has produced an abundance of raw CPU power, enabling the collocation of many such services to decrease administrative overhead.

Our research has revealed that traditional VMMs suffer from scalability bottlenecks that artificially restrict the number of virtual machines a given system can support.[4] These bottlenecks exist because the notion of time is more complex on a VMM. A VMM runs multiple virtual machines in parallel, so each VM only runs on the real processor for 1/*N* of the total CPU time, on average. This effect creates a notion of *virtual time* that advances at a different rate than physical or wall-clock time.

As the number of virtual machines increases, the disjunction between virtual time and physical time increases, adversely affecting any timing-dependent aspect of the hardware, including interrupt delivery and timers. To address these challenges, we propose paravirtualization. The key idea in this technique is to expose a virtual hardware architecture that differs from the underlying physical hardware architecture. Small changes to the virtual architecture are sufficient to eliminate the artificial scalability bottlenecks that plague traditional systems. Figure 1 compares a paravirtualized VMM to other system architectures.

Denali's use of paravirtualization has parallels with earlier work in operating system design. Researchers in the 1970s proposed *impure* virtual machine architectures to improve performance or reduce implementation complexity. In a similar vein, microkernel systems such as Mach expose

low-level abstractions, which are similar to but differ from a hardware interface.

All these approaches have a downside: Legacy guest operating systems require modifications to run on the modified architecture. A major goal of our work was to avoid wholesale changes to legacy code bases. We achieve this goal by confining our architectural modifications to fit within the operating system's *hardware abstraction layer*. A primary purpose of the HAL is to ensure portability across a set of physical hardware architectures. Thus, by confining our virtual architecture changes to the HAL, porting an OS to Denali is no more difficult than porting that OS to a new physical hardware architecture. In contrast to Denali, microkernel systems have introduced more disruptive architectural changes, greatly complicating support for legacy guest operating systems and their applications.

To date, we have ported the NetBSD operating system to the Denali architecture. This port required only a one-line change outside the NetBSD's hardware abstraction layer. The Xen research group at Cambridge has applied paravirtualization techniques to a broader set of operating systems, including Linux and Windows. Their experience has been similar to ours:[3] A small team of programmers can accomplish the relatively straightforward task of porting an OS to a paravirtualized architecture.

Architecture changes for scale

Three aspects of the traditional hardware-software interface limit scalability: idling, interrupts, and timers.

The presence of *idle loops* within a guest operating system poses one barrier to scalability. On physical hardware, the OS executes an idle loop while waiting for some event of interest to transpire. On a VMM, these idle loops waste useful cycles that could be devoted to another virtual machine.

To avoid this performance degradation, Denali exposes an idle-with-timeout instruction, which lets the guest OS yield the CPU for a bounded time. This allows full processor utilization while ensuring that a virtual machine awakes to handle timer-related functionality.

As Figure 2 shows, the idle-with-timeout instruction prevents a 66 percent throughput degradation in aggregate throughput for a collection of Web server virtual machines. This experiment ran on a 1,700-MHz Pentium 4 with 256 Kbytes of L2 cache, 1 Gbyte of RAM, and an Intel PRO/1000 PCI gigabit Ethernet card.

On physical hardware, interrupt delivery occurs immediately after the arrival of some hardware event such as packet arrival or timer firing. On a VMM, preserving immediate interrupt delivery is difficult because a given virtual machine runs only 1/*N* percent of the time.

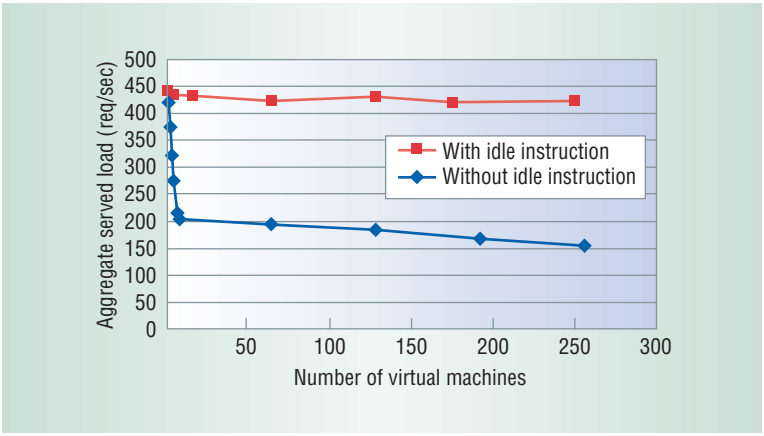

Figure 2. Benefit of idle-with-timeout instruction. The graph depicts the aggregate throughput across an increasing number of virtual machines, which run a custom Web server atop a lightweight operating system. Each Web server VM serves a static 130-Kbyte document. The x-axis reflects the number of VMs across which the workload is spread. Denali avoids a 66 percent throughput degradation for a large collection of Web server virtual machines.

To account for this limitation, Denali uses a batched, asynchronous interrupt model that queues virtual hardware events until the virtual machine's normal-scheduler quantum occurs. Using this approach, Denali avoids a 30 percent performance degradation for many VMs.

Operating systems use the arrival of *timer interrupts* to measure the passage of physical time. With each virtual machine limited to 1/*N* percent of the CPU, and therefore missing the majority of physical timer ticks, Denali exposes a global physical timer that operating systems can read to learn how much physical time has transpired since the VM's last scheduler quantum.

Architecture changes for simplicity

Beyond promoting scalability, we leveraged the freedom that paravirtualization offers to simplify our VMM implementation. Denali's virtual architecture omits several rarely used features, such as the BIOS, x86 segmentation hardware, and protection rings. Denali also replaces the x86's hardware-filled translation lookaside buffer with a software-filled TLB, resulting in a simpler and more efficient implementation.

Denali's use of paravirtualization also cleanly sidesteps the problem of unvirtualizable instructions. The x86 architecture contains several instruc-

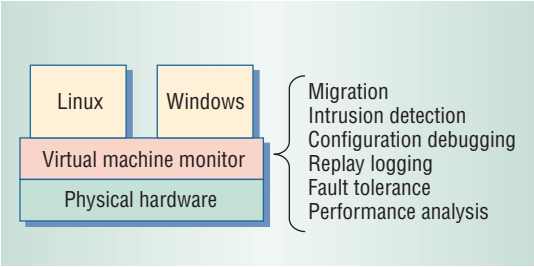

Figure 3. Virtual machine services. Virtual machine monitors have proven useful for realizing whole-system services that apply to virtual machines.

tions that behave differently in user and kernel mode, yet do not force a trap into the VMM.[5] This is problematic for conventional VMMs, which attempt to precisely emulate the underlying physical hardware's behavior.[6] Denali makes no attempt to precisely emulate physical hardware, thereby avoiding the complexity inherent in handling these instructions.

EXTENDING A VMM

The VMM's traditional role has been to multiplex a single physical machine across multiple users or applications. Recently, researchers have moved beyond multiplexing to explore novel applications of virtual machine technology.

As Figure 3 shows, these virtual machine services leverage the layer of indirection between virtual and physical hardware to realize functionality such as migration, intrusion detection, and performance analysis.[7]

A key advantage of implementing services within a VMM is that such services have a *whole-system perspective*: They capture the complete state of a running operating system and its suite of applications. This perspective is important because many services of interest cut across traditional OS encapsulation mechanisms, such as processes or address spaces. For example, with a virtual machine migration primitive, users can move a complete computing environment, including an operating system and application suite.

Virtual machine services also benefit from a VMM's simplicity relative to a full-blown OS. For this reason, virtual machine migration primitives have proven easier to implement and maintain than corresponding process migration primitives. The VMware ESX server supports VM migration. We know of no process migration implementation that has received major operating system support.

Despite the utility of virtual machine services, realizing such services within the current generation of VMMs can be difficult. Implementing a service requires the ability to change or extend parts of the VMM implementation.

Unfortunately, traditional VMMs support only a single, fixed implementation of hardware abstractions such as virtual disks and virtual Ethernet. As a result, service designers must expend considerable effort to refactor an existing VMM's implementation.

An alternative to modifying source code is to reverse engineer a black-box VMM such as VMware. However, developers did not design conventional VMMs with extensibility in mind, and thus VMMs lack the necessary hooks to realize some services.[8]

Given the difficulty of realizing any particular virtual machine service, research groups have paid little attention to how they might cooperate. This has led to repetition in research and development. For example, although the Hypervisor[9] and Revirt[8] projects both rely on the same underlying logging primitive, it would be difficult for these groups to share this functionality with existing virtual machine technology.

μDenali,[10] a system that facilitates the rapid development of new virtual machine services, embodies our solution to these problems. The system achieves extensibility through two primary components. First, it exposes a set of programmatic interfaces. Developers can extend these interfaces to modify the implementation of hardware abstractions such as virtual disks without getting bogged down in the VMM's implementation details. Second, μDenali supports *hardware interposition,* which lets user extensions override the default system functionality.

Programmatic API

Traditional operating systems such as Unix provide programmatic interfaces that third-party developers can use to extend system functionality. For example, programmers can use the BSD virtual file system interface to supply new file system implementations. μDenali's programmatic API provides a similar level of programmability and extensibility to the VMM domain. Whereas traditional OS extensions provide high-level abstractions like a file system, VMM extensions provide low-level abstractions like a virtual disk device.

Supporting extensibility through a clean programmatic interface offers numerous benefits. For example, this interface shields programmers from the implementation details of the underlying VMM and avoids wasting effort on refactoring or reverse engineering. Using abstract interfaces also can serve as the basis for component-based programming, letting developers reuse service components. Finally,

the system can port extensions across any VMM that exposes the same programmatic API.

We designed µDenali's programmatic API based on a survey of existing virtual machine services. This survey identified four aspects of VMM behavior that would benefit from a clean extensibility mechanism:

- *Extending I/O devices*. Many services must be able to monitor or modify the behavior of I/O devices such as virtual disks and the Ethernet. For example, the Chronus tool implements a "time-travel" disk that facilitates system configuration debugging.
- *Exposing virtual machine state*. To enhance performance, µDenali caches some virtual machine state inside the hardware or within the VMM. For example, the current register contents of a virtual machine can reside inside the processor. µDenali provides an API to extract this state, which is used by services such as checkpointing and migration that require up-to-date knowledge of the complete virtual machine state.
- *Tracking nondeterminism*. Services such as virtual machine replay[8] and fault tolerance[9] require precise timing information for nondeterministic events such as timer interrupts.
- *Controlling virtual machines*. Developers can use µDenali's programmatic API to start, stop, and kill virtual machines.

The programmatic API consists of a set of C-based interfaces. Figure 4 shows µDenali's API for implementing a new virtual-disk abstraction. The virtual-disk interface contains a set of upcalls invoked in response to actions within the target virtual machine. For example, a disk read operation in the target virtual machine triggers a corresponding invocation of the diskRead function. Other programmatic interfaces contain downcalls that programs can use to exert active control over a virtual machine. For example, the virtual machine control API provides the ability to start, stop, and suspend virtual machines.

```
// 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 API permits parents to interpose on and respond to their children's device operations. Only the interface associated with the virtual disk is shown, but other devices have similar interfaces.

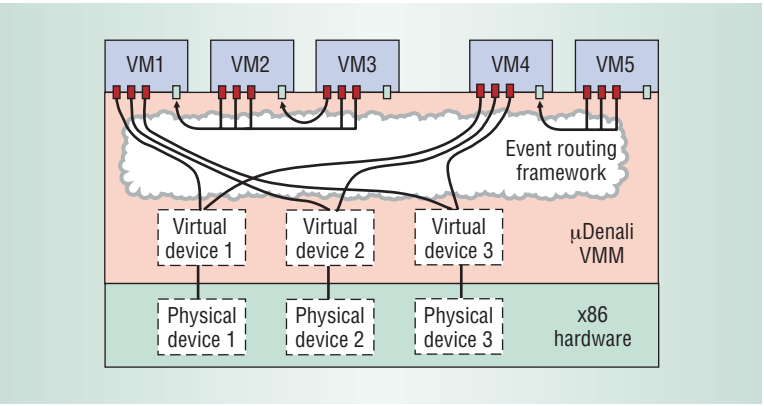

Figure 5. µDenali software architecture. An event-routing framework separates a virtual device's interface from its implementation. Some devices have default implementations within the VMM, while user-supplied extensions inside separate virtual machines implement other devices.

Hardware interposition

Denali's initial implementation contained hard-coded abstractions such as virtual disks. To support extensibility, it was necessary to separate the interface of virtual hardware abstractions from their implementations. We achieved this separation using a technique called *hardware interposition*.

The key idea in this approach is to transform the VMM into a general message-routing framework. The system transforms all virtual hardware events, such as disk reads, into messages that the VMM routes to an appropriate destination. Some events are handled by default implementations within the VMM, and some are handled by extension code running inside of virtual machines.

The message-routing framework's general nature makes it possible to build up arbitrary message-routing topologies. Thus, a child virtual machine could have multiple parents, each of which implements some portion of its virtual hardware functionality. Figure 5 depicts the µDenali's software architecture.

Service examples

We have implemented a range of virtual machine services atop µDenali's extensibility mechanisms. A service called *Apache** uses a virtual cluster of Apache Web servers to improve reliability in the face of software errors. The Chronus tool uses "time travel" debugging to help diagnose computer configuration errors. In addition to these novel services, we have used µDenali to reimplement previously proposed services such as virtual machine migration. Using µDenali's clean programmatic APIs, our migration implementation required only 289 lines of C source code.

A well-known maxim in computer science holds that any problem can be solved with a layer of indirection. Virtual machine monitors introduce such a layer beneath an entire computer system. This mechanism has proven useful for realizing a wide range of system services. Our work on the Denali VMM has expanded the applicability of VMMs by improving the scalability and extensibility of these systems.

In the future, we expect that innovations will arise from applying virtual machine technology in new and innovative ways. One opportunity will be to leverage the strong isolation of virtual machines to avoid configuration conflicts between applications running on a single system. Another opportunity would be to use VMMs to simplify software testing and debugging. ■

References

1. R.J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Research and Development*, vol. 25, no. 5, 1981, pp. 483-490.
2. P.A. Karger et al., "A Retrospective on the VAX VMM Security Kernel," *IEEE Trans. Software Engineering*, Nov. 1991, pp. 1147-1165.
3. P. Barham et al., "Xen and the Art of Virtualization," *Proc. 19th Symp. Operating System Principles* (SOSP 2003), ACM Press, 2003, pp. 164-177.
4. A. Whitaker, M. Shaw, and S.D. Gribble, "Scale and Performance in the Denali Isolation Kernel," *Proc. 5th Symp. Operating Systems Design and Implementation* (OSDI 02), Usenix, 2002, pp. 195-209.
5. J.S. Robin and C.E. Irvine, "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor," *Proc. 9th Usenix Security Symp.*, Usenix, 2000, pp. 129-144.
6. G.J. Popek and R.P. Goldberg, "Formal Requirements for Virtualizable Third-Generation Architectures," *Comm. ACM*, July 1974, pp. 412-421.
7. A.J. Whitaker, "Building Robust Systems with Virtual Machine Monitors," Univ. Washington Generals Examination, 2004; www.cs.washington.edu/homes/andrew/papers/general.pdf.
8. G.W. Dunlap et al., "Enabling Intrusion Analysis through Virtual Machine Logging and Replay," *Proc. 2002 Symp. Operating Systems Design and Implementation* (OSDI 02), Usenix, 2002, pp. 211-224.
9. T.C. Bressoud and F.B. Schneider, "Hypervisor-Based Fault Tolerance," *ACM Trans. Computer Systems*, vol. 14, no. 1, 1996, pp. 80-107.
10. A. Whitaker et al., "Constructing Services with Interposable Virtual Hardware," *Proc. 1st Symp. Network Systems Design and Implementation*, Usenix, 2004, pp. 169-182.

Andrew Whitaker *is a graduate student in the Department of Computer Science at the University of Washington. His research interests are operating systems, networking, and software engineering. Whitaker received an MS in computer science from the University of Washington. He is a member of the ACM and Usenix. Contact him at andrew@cs.washington.edu.*

Richard S. Cox *is a graduate student in the Department of Computer Science at the University of Washington. His research interests are operating systems, security, and the design of "future-proof" systems that will stand the test of time. Cox received an MS in computer science from the University of Washington. He is a member of Usenix. Contact him at rick@cs.washington.edu.*

Marianne Shaw *is a graduate student in the Department of Computer Science at the University of Washington. Her research interests are operating systems, networking, and security. Shaw received an MS in computer science from the University of Washington. She is a member of the ACM and Usenix. Contact her at mar@cs.washington.edu.*

Steven D. Gribble *is an assistant professor in the Department of Computer Science at the University of Washington. His research interests include the design and operation of robust, scalable Internet infrastructure and services, the measurement and design of wide-scale distributed systems, and virtual machine monitors. Gribble received an MS in computer science from the University of California, Berkeley. He is a member of the ACM and Usenix and cofounded ProxiNet, now a division of PumaTech. Contact him at gribble@cs.washington.edu.*