acmqueue

# I/O Virtualization

**Decoupling a logical device from its physical implementation offers many compelling advantages.**

Mendel Rosenblum, Carl Waldspurger

The term *virtual* is heavily overloaded, evoking everything from virtual machines running in the cloud to avatars running across virtual worlds. Even within the narrowfigureer context of computer I/O, virtualization has a long, diverse history, exemplified by logical devices that are deliberately separate from their physical instantiations.

For example, in computer storage, a LUN (logical unit number) represents a logical disk that may be backed by anything from a partition on a local physical drive to a multidisk RAID volume exported by a networked storage array. In computer networking, a VPN (virtual private network) represents a logically isolated private network, where the isolation is provided using cryptographic methods to secure data that may in fact traverse the public Internet. In computer architecture, an IOMMU (I/O memory-management unit) translates I/O virtual memory addresses to corresponding physical memory addresses, making direct memory access by devices safe and efficient. Other examples of virtualization include VLAN (virtual LAN), NPIV (N_Port ID virtualization), VT-d (Intel Virtualization Technology for Directed I/O), and MR-IOV (Multiroot I/O Virtualization).

The common theme is decoupling the logical from the physical, introducing a level of indirection between the abstract and the concrete. Such indirection has proven to be remarkably powerful and versatile. Modern virtualization platforms exploit indirection and abstraction in numerous ways. A VM (virtual machine) is a software abstraction that behaves as a complete hardware computer, including virtualized CPUs, RAM, and I/O devices. A virtualization software layer, known as a *hypervisor*, provides the level of indirection that decouples an operating system and its applications from physical hardware. The term *guest* is commonly used to distinguish the layer of software running within a VM; a guest operating system manages applications and virtual hardware, while a hypervisor manages VMs and physical *host* hardware.

Although IBM invented and commercialized mainframe VMs many decades ago, virtual machines didn't make the leap to commodity hardware until the late 1990s, when VMware pioneered efficient virtualization on x86 platforms. Since then, virtualization has experienced a resurgence of interest in both industry and academia. Today, VMs are commonplace in many computing environments and nearly ubiquitous in enterprise data centers and cloud-computing infrastructures.

Since virtualization is a broad topic, and the universe of I/O devices is large and diverse, this article focuses on some representative I/O systems issues in VM-based systems, primarily in the context of a single physical host. After highlighting key benefits and challenges, we explore various implementation approaches and techniques that have been leveraged to enable flexible, high-performance I/O virtualization.

## BENEFITS

Many of the benefits of virtualized systems depend on the decoupling of a VM's logical I/O devices from its physical implementation. Examples range from the ability to multiplex many VMs on the

same hardware to advanced virtualization features such as live migration and enhanced security.

At the most basic level, decoupling enables time- and space-multiplexing of I/O devices, allowing multiple logical devices to be implemented by a smaller number of physical devices. Applications of virtualization such as server consolidation or running heterogeneous operating system environments on the same machine rely on this feature. As inexorable trends create ever more powerful hardware, it's not surprising that much of it remains seriously underutilized. The ability to multiplex logical I/O devices onto physical ones allows both administrators and automated systems to drive I/O devices at higher utilization and achieve better hardware efficiency. Much of virtualization's rapid adoption over the past decade can be attributed to the significant cost savings resulting from such basic partitioning and server consolidation.

Decoupling provides for flexible mappings between logical and physical devices, facilitating seamless portability. By supporting mappings of logical I/O devices to physical devices with different yet semantically compatible interfaces, virtualization makes VMs portable, even across heterogeneous systems. The same VM image can be run on computers with different I/O devices and configurations, with the I/O virtualization layer providing the necessary conversion.

Decoupling also enables popular VM features such as the ability to suspend and resume a virtual machine and the ability to move a running virtual machine between physical machines, known as live migration. In both of these applications, active logical devices must be decoupled from physical devices and recoupled when the VM resumes after being saved or moved.

This virtualization layer may also change mappings to physical devices, even when the VM itself does not move. For example, by changing mappings while copying storage contents, a VM's virtual disks can be migrated transparently between network storage units, even while remaining in active use by the VM. The same capability can be used to improve availability or balance load across different I/O channels. For example, in a storage system with multiple paths between the machines and storage, the virtualization layer can rebind mappings to mask failures or avoid delays that might occur because of contention on paths.

I/O virtualization provides a foothold for many innovative and beneficial enhancements of the logical I/O devices. The ability to interpose on the I/O stream in and out of a VM has been widely exploited in both research papers and commercial virtualization systems.

One useful capability enabled by I/O virtualization is device aggregation, where multiple physical devices can be combined into a single more capable logical device that is exported to the VM. Examples include combining multiple disk storage devices exported as a single larger disk, and network channel bonding where multiple network interfaces can be combined to appear as a single faster network interface.

New features can be added to existing systems by interposing and transforming virtual I/O requests, transparently enhancing unmodified software with new capabilities. For example, a disk write can be transformed into replicated writes to multiple disks, so that the system can tolerate disk-device failures. Similarly, by logging and tracking the changes made to a virtual disk, the virtualization layer can offer a time-travel feature, making it possible to move a VM's file system backward to an earlier point in time. This functionality is a key ingredient of the snapshot and undo features found in many desktop virtualization systems.

Many I/O virtualization enhancements are designed to improve system security. A simple example is running an encryption function over the I/O to and from a disk to implement transparent disk

encryption. Interposing on network traffic allows virtualization layers to implement advanced networking security, such as firewalls and intrusion-detection systems employing deep packet inspection.

## CHALLENGES

While virtualization offers many benefits, it also introduces significant challenges. One is achieving good I/O performance despite the potential overhead associated with flexible indirection and interposition. Complex resource-management issues such as scheduling and prioritization are introduced by multiplexing physical devices across multiple VMs, further impacting performance. Another challenge is defining appropriate semantics for virtual devices and interfaces, especially when faced with complex physical I/O devices or system-level optimizations.

In many systems, a nontrivial performance penalty is associated with indirection. The same can be true for virtualized I/O, since I/O operations must conceptually traverse two separate I/O stacks: one in the guest managing the virtual hardware and one in the hypervisor managing the physical hardware. The longer I/O path affects both latency and throughput, and imposes additional CPU load. Indeed, I/O-intensive workloads on some early virtualization systems suffered a virtualization penalty larger than a factor of two. Since then, further research, optimizations, and hardware acceleration have reduced this penalty into the noise for an impressive set of demanding production workloads. Somewhat counterintuitively, virtualized systems have even outperformed native systems on the same physical hardware, overcoming native scaling limitations by instead running several smaller VM instances in a scale-out configuration.

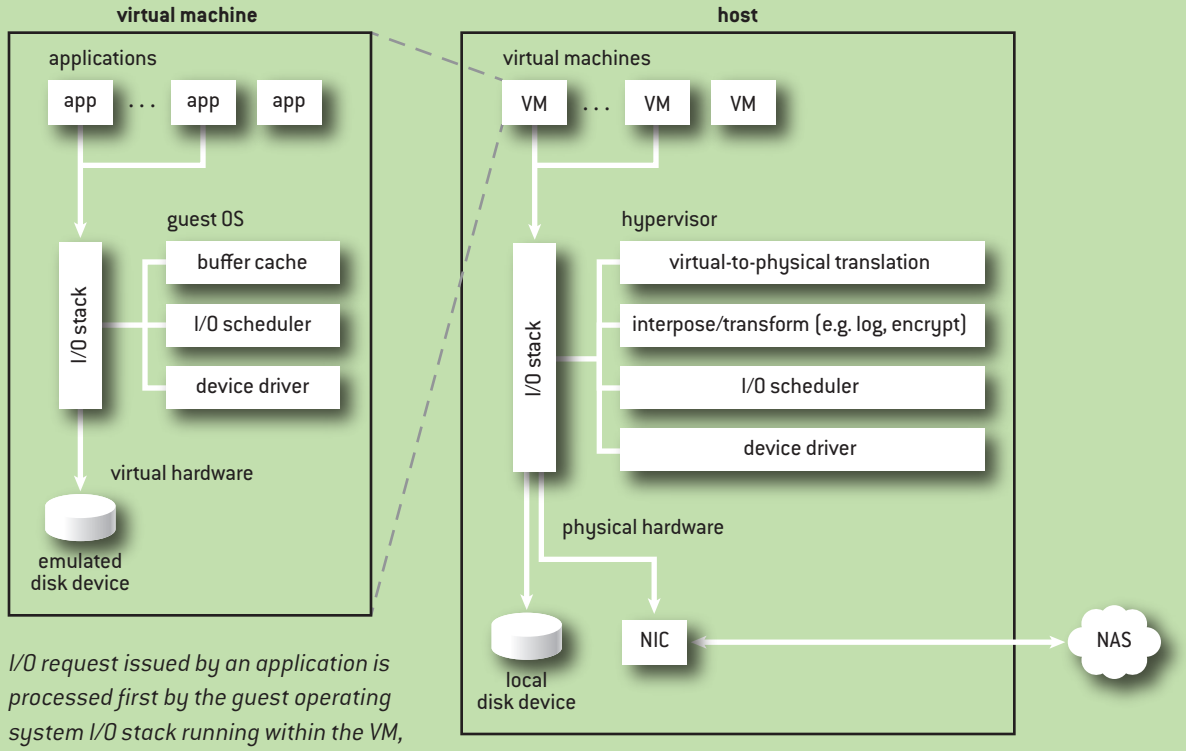
Figure 1 depicts the flow of an I/O request in a virtualized system. When an application running within a VM issues an I/O request, typically by making a system call, it is initially processed by the I/O stack in the guest operating system, which is also running within the VM. A device driver in the guest issues the request to a virtual I/O device, which the hypervisor then intercepts. The hypervisor schedules requests from multiple VMs onto an underlying physical I/O device, usually via another device driver managed by the hypervisor or a privileged VM with direct access to physical hardware.

When a physical device finishes processing an I/O request, the two I/O stacks must be traversed again, but in the reverse order. The actual device posts a physical completion interrupt, which is handled by the hypervisor. The hypervisor determines which VM is associated with the completion and notifies it by posting a virtual interrupt for the virtual device managed by the guest operating system. To reduce overhead, some hypervisors perform virtual interrupt coalescing in software, similar to the hardware batching optimizations found in physical cards, which delay interrupt delivery with the goal of posting only a single interrupt for multiple incoming events.

Interposition can incur additional overhead by manipulating I/O requests such as inspecting network packets to perform security checks or encrypting disk writes transparently. In some cases, the interposition costs are negligible, especially compared with high-latency operations such as I/O to traditional rotating media. In other cases, even making an extra in-memory copy of I/O data may be prohibitively expensive—for example, for fast networks with extremely high packet rates. To improve performance, some hypervisors parallelize portions of this processing, offloading work to additional processor cores. Of course, when there is contention for CPUs, this leaves fewer cores available for running VMs.

Managing resources in virtualized systems presents additional challenges. Although each VM is

FIGURE 1 **Processing an I/O Application Request from a VM**




*I/O request issued by an application is processed first by the guest operating system I/O stack running within the VM, and then by the hypervisor I/O stack managing physical hardware.*

presented with the illusion of having its own dedicated virtual hardware, in reality the hypervisor must multiplex limited physical hardware across multiple VMs of varying importance, mapping their virtual resources onto available physical resources. At the most basic level, contention for a physical device will result in scheduling delays for some VMs. At a minimum, the hypervisor must prevent VMs from monopolizing resources and denying service to others.

More generally, the hypervisor should provide some measure of performance isolation or quality-of-service controls to reflect the relative importance or absolute requirements of diverse VM workloads. The ability to express resource-management policies is especially important when physical resources are shared across multiple users or organizations, as is common in multitenant cloud-computing environments. Several hypervisors support a relative weight control, where a VM's allocation is directly proportional to its weight. Some also provide absolute reservation and limit settings, which bound a VM's minimum and maximum allocation, regardless of system load.

For I/O devices that can be accessed concurrently by VMs on different hosts, such as networked storage arrays, resource management requires distributed algorithms to schedule requests fairly and efficiently. Virtualization platforms have only recently started offering sophisticated solutions capable of providing end-to-end quality of service for VM I/O bandwidth and latency.

Scheduling may also impact VM performance in subtler ways. For example, contention for CPU resources can cause problems for TCP networking performance. TCP connections rely on accurate RTT (round-trip time) estimates in order to perform flow control and adjust window sizes appropriately. A VM, however, may be descheduled for tens or even hundreds of milliseconds while a packet is pending. As a result, CPU time-multiplexing can distort a VM's RTT values, causing its congestion windows to grow too slowly, which degrades throughput significantly. To solve this problem, some researchers have proposed offloading more TCP functionality to the hypervisor. Another option would be to present VMs with virtual NIC (network interface controller) hardware that supports optional TOE (TCP Offload Engine) functionality, as found in some physical NICs.

The very idea of adding TCP offload capabilities to a virtual NIC highlights the difficulty of choosing appropriate semantics for virtual hardware. This is especially true for devices with even more complex interfaces, such as modern graphics cards. At one extreme, virtual hardware can have an interface identical to a physical device. This approach has the compelling advantage of compatibility with all software that already supports (or will support) the physical device. Unfortunately, such transparency usually comes at the cost of emulating a fairly complex virtual device interface that wasn't designed to support virtualization efficiently. At the other extreme, virtual hardware can have a completely new hypervisor-specific interface, designed explicitly to be simple and efficient.

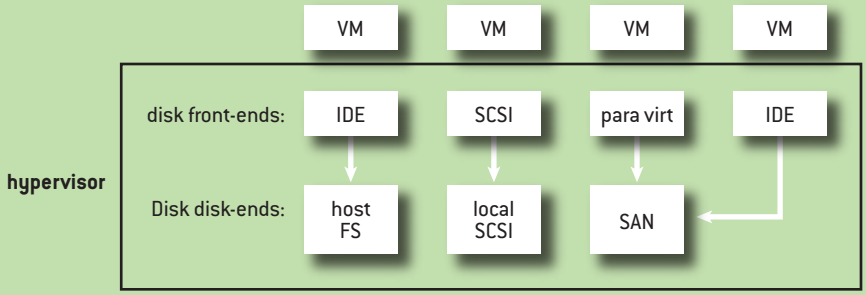
A related challenge is ensuring that virtualization faithfully preserves the semantics that software expects of physical devices. For example, some virtualization systems boost I/O performance by leveraging a hypervisor-level buffer cache between the VM and physical storage. While caching reads doesn't introduce any problems, caching writes would violate the durability semantics relied upon by guest file systems, databases, and other software. In a native, unvirtualized system, an I/O completion indicates that a write has been committed. Hypervisors must use a write-through cache to preserve this property, although some provide an explicit option to trade off safety for performance by relaxing this constraint.

DMA (direct memory access) illustrates additional safety and performance issues. It enables an I/O device to read and write host RAM directly without involving the CPU, which is critical for achieving high-performance I/O rates. Unfortunately, giving devices the ability to use DMA to reach arbitrary physical memory locations is risky, especially since the majority of operating system bugs result from misbehaving device drivers. As discussed in the next section, virtualization systems can ensure strict isolation between VMs by employing various approaches, such as leveraging hardware IOMMU functionality at both the guest and hypervisor levels.

## APPROACHES

The classic way of implementing I/O virtualization is to structure the software in two parts: an emulated virtual device that is exported to the VM and a back-end implementation that is used by the virtual-device emulation code to provide the semantics of the device. Modern hypervisors support an I/O virtualization architecture with a split implementation, as shown in figure 2, where a virtual machine can select among different virtual device interface emulation front-ends as well as multiple different back-end implementations of the device. For example, the virtual machine can be configured with IDE, SCSI, or a paravirtualized disk device that is implemented either as a file, local disk, or SAN (storage area network). This section describes the way these are implemented in modern

FIGURE 2 **Modern Hypervisors Split Device Virtualization**




virtualization systems and discusses some of the available optimization options.

To achieve the classic virtualization capability of running in a VM the same software environment that ran on a physical machine, the abstractions exported to the VM must be similar enough to the physical machine that the software will run. This can be surprisingly complex. For example, consider the commodity PC architecture that VMware inherited for legacy VMs. Ensuring compatibility meant exporting several PC device abstractions, including architecturally-defined abstractions such as x86 IN and OUT instructions, uncached load and store accesses to I/O device addresses, DMA, and interrupts. The virtualization layer and the virtual-device emulation code must support the architectural and device semantics faithfully enough for the code in the VM to execute correctly.

As a concrete example, consider a legacy PC I/O device such as an IDE disk. The operating system in the VM would call a device driver to launch disk read or write requests. The device driver would include OUT instructions that program the operation type, device number, disk-sector number, length, and buffer-memory location for the operation. The driver assumes that the device will use DMA to transfer the contents between memory and the disk device and then raise an interrupt when it is done. For the emulated device to work correctly, the emulation software must catch and interpret the OUT instructions to determine the correct operation and its arguments; perform an emulation of the operation by storing or fetching the requested block of storage, using the emulation of the architecture's DMA capability to read or write memory; and finally raise the proper interrupt signal on the VM to notify the driver that the request is finished.

While the device emulation code is specific to the particular device being emulated (e.g., an IDE disk), the semantics of the operations being performed are general and frequently constructed so that the same device emulation can access multiple different back-end implementations. For example, with virtual disks, the back-end implementation could be as simple as forwarding the request untransformed to a native physical IDE controller or as complex as implementing the storage for the virtual disk as a file in a host operating system file system, as in many desktop virtualization products. In the latter case, the back-end must generate host operating system file-system read and write operations for the file containing the virtual disk contents in order to perform emulated virtual disk-sector read and write operations.

A pluggable structure for back-end implementations makes it easy to generate new capabilities for virtual devices. A disk-storage back-end can implement an emulated CDROM device for a VM simply by accessing a file containing its ISO (International Organization for Standardization) image. Similarly, the snapshot and undo capabilities in modern virtualization systems can be implemented by logging rather than overwriting a virtual-disk file, enabling the virtualization layer to control which version of the disk is visible to the VM.

Along with flexibility and innovative features come potential performance penalties, which can vary greatly. A back-end optimized for a server machine that efficiently shuttles emulated reads and writes to a portion of a local disk may have very low virtualization overheads. In contrast, the penalty may be relatively large for a desktop-virtualization back-end where the data is stored in an encrypted file on a network file system used by the host operating system. Not only would every read and write request traverse the file system and networking code of the host operating system and the network file system, but additional encryption and decryption overhead would also be incurred for each sector.

Much current research in I/O virtualization is focused on either new interposition functionality that solves some problem or optimizations for reducing the overheads associated with the virtualization. Optimization has been an especially important goal in server virtualization and consolidation, where overheads directly impact metrics such as the number of VMs supported per server. Beyond back-end optimizations, reducing I/O-virtualization overhead requires decreasing virtual-device emulation costs. Several recent optimizations have tried exactly this using either software or modifications to the I/O device hardware.

One of the software techniques for lowering emulation costs is to reduce the number of trap-and-emulate operations the hypervisor is required to do to perform an I/O operation. For example, the legacy PC IDE interface uses eight-bit OUT instructions to communicate with the disk controller. Communicating the sector number, buffer address, and length requires multiple such instructions, causing repeated traps into the hypervisor to run the device emulation code. Using the driver for an alternative disk device such as a SCSI disk can achieve the same functionality with far fewer traps, greatly reducing the emulation overheads.

Even further reduction can be achieved by optimizing the communication between the VM software and the device emulation. In modern operating system environments such as Windows and Linux, it is possible to install device drivers that communicate the request's arguments to the hypervisor's device emulation code directly, with minimal overhead. This approach of using virtual hardware optimized for the virtualization layer rather than matching any particular real device is referred to as *paravirtualization*. In practice, most modern virtualization platforms support an emulated legacy device for compatibility, as well as providing an optional paravirtual device for higher performance.

As an example, a paravirtualized disk interface could have the device emulation code accept commands via a memory segment shared between the driver and the emulation, allowing communication of commands with practically zero overhead. The emulation code simply passes the command to the optimized back-end implementation. Examples include Xen's virtual block-device front-end driver and VMware's PVSCSI guest disk driver.

For an I/O device used by only a single VM, where the back-end implementation is mostly passing through the driver commands from the VM to the device, it is tempting to pass through the device

directly, assigning it to the VM exclusively. Consider the example of a high-performance NIC that is used by only one of the VMs running on a computer. It is relatively easy to configure the CPU virtualization so the x86 instructions that talk to the device can be connected directly to the device and incur zero I/O virtualization overheads. This *pass-through mode* can eliminate both the device emulation and back-end implementation overheads.

Although pass-through mode can remove I/O virtualization overheads, it introduces several limitations and implementation challenges that have slowed its deployment. Aside from the obvious limitation that each pass-through device can be used by only a single VM, pass-through forms a coupling between the hardware and the VM. As a result, many of the portability benefits of virtualization are lost, along with key benefits such as live migration and features that depend on the ability to interpose on I/O.

One of the biggest challenges of pass-through mode affects devices that use DMA. The fundamental problem is that the driver in the VM will program device DMA using the guest's notion of memory addresses, which differ from the real memory addresses in which the VM's memory resides. This is not only incorrect, but also a large safety and security problem since the device could read and write memory potentially belonging to the hypervisor or some other VM. To make this work, the VM's driver must translate memory addresses to use the correct real memory before programming them into the device. This exposes the driver to the details of hypervisor memory virtualization and still has safety problems since bugs in the driver can result in incorrect translations.

To eliminate both the limitations and the challenges of pass-through, device builders have modified their hardware to be aware of the virtualization layer. To handle the limitation of exclusive pass-through-only devices, such virtualization-aware hardware exports multiple interfaces, each of which can be attached to a different VM. As a result, each VM is given its own directly accessible pass-through copy of the device. For example, a virtualization-aware NIC could have many personalities that look and act as if many separate NICs had been directly mapped into different VMs.

Additional hardware support is needed to address the challenges of performing DMA operations directly involving VM memory. A memory management unit is employed to map the DMA operation's memory addresses into the correct locations in the VM's memory. This mapping hardware—the IOMMU—is programmed for each VM attached to a device with the mappings of where the VM resides in memory. Each DMA request is run through the IOMMU, which routes the request to or from the correct location in the real machine memory or generates an error if the request is not valid. The IOMMU allows the driver in the VM to program device DMA using its virtualized notion of memory addresses, while still allowing the hypervisor to decide where VM memory is actually located in physical machine memory. The IOMMU also provides a level of safety, ensuring that even buggy driver software in the guest cannot generate DMA accesses to locations outside the VM.

Although IOMMUs can safely and efficiently allow virtualization-aware I/O devices to access the memory of a virtual machine directly, there are implications for some of the more sophisticated memory virtualization operations in modern hypervisors that rely upon dynamic page remapping. Consider features such as overcommitting memory, where the hypervisor can reclaim RAM via techniques such as demand-paging VM memory to secondary storage; memory compression; or

transparent memory sharing where identical pages can be de-duplicated by sharing them read-only between multiple virtual machines. These features require that certain accesses to VM memory take faults and invoke hypervisor actions before they are allowed to proceed. DMA devices need to respect this, and hence the devices need to support something similar to page faults on DMA operations where the hypervisor is invoked before the DMA is permitted to finish. The ability to tolerate arbitrary delays on DMA operations can have a much deeper impact on the changes needed to make an I/O device virtualization-aware.

While production hypervisors employ hardware IOMMUs and other hardware-enforced memory-mapping techniques to guarantee isolation between VMs, they have not yet included IOMMUs in the virtual hardware presented to VMs. A vIOMMU (virtual IOMMU) would allow a guest operating system to defend against its own buggy device drivers, as in a native system.

Recently, researchers have developed new IOMMU emulation techniques for efficiently providing vIOMMUs to guests. Even more significantly, the same approach facilitates a more flexible form of device pass-through, where a VM is allowed to interact with a directly-assigned I/O device without hypervisor intervention. Since the guest exposes which regions of its memory are currently involved in DMA operations to the vIOMMU, the hypervisor is able to modify mappings for other memory regions safely. By interposing only on vIOMMU operations, it is possible to achieve near-native I/O performance while preserving the hypervisor's ability to manage, remap, and overcommit memory.

## CONCLUSIONS

Decoupling a logical device from its physical implementation offers many compelling advantages. A single physical device can be multiplexed, allowing it to act as many virtual devices, improving hardware utilization. Abstracting away details about specific hardware and physical location makes seamless migration possible. The level of indirection between virtual and physical also provides a convenient hook for interposing on I/O operations transparently, enabling new capabilities such as replication, load balancing, encryption, and security checks.

A key challenge in I/O virtualization is achieving these benefits with minimal overhead. A number of clever software and hardware approaches have been devised to achieve high-performance indirection and interposition, including paravirtualization and virtualization-aware devices. Resource-management issues, such as scheduling and prioritization, become important when device multiplexing is used to consolidate different workloads onto the same physical hardware. Defining clean interfaces and appropriate semantics for virtual devices is also challenging.

I/O virtualization remains an active area of research and development in both academia and industry. Although we focused here on systems issues from the perspective of an individual physical machine, the broader context of I/O virtualization includes an enormous range of work on distributed systems and the fabrics that connect their virtual and physical components. The increasing prevalence and commercial success of systems based on virtual machines is certain to fuel demand for new virtualization optimizations and I/O capabilities.

### SUGGESTED READING

Ahmad, I., Gulati, A., Mashtizadeh, A. 2011. vIC: interrupt coalescing for virtual machine storage device IO. In *Proceedings of the 2011 Usenix Annual Technical Conference* (June).

Amit, N., Ben-Yehuda, M., Tsafrir, D., Schuster, A. 2011. vIOMMU: efficient IOMMU emulation. In

*Proceedings of the 2011 Usenix Annual Technical Conference* (June).
Gamage, S., Kangarlou, A., Kompella, R. R., Xu, D. 2011. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *Proceedings of the Second ACM Symposium on Cloud Computing* (October).
Gulati, A., Ahmad, I., Waldspurger, C. 2009. PARDA: Proportional allocation of resources for storage access. In *Proceedings of the Seventh Conference on File and Storage Technologies* (February).
Sugerman, J., Venkitachalam, G., Lim, B-H. 2001. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Usenix Annual Technical Conference* (June).

## LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

**CARL WALDSPURGER** is an advisor and consultant to several startups. For most of the previous decade, he was a principal engineer at VMware, where he was responsible for core resource management and virtualization technologies, including scheduling, memory management, and distributed systems. Prior to VMware, he was a researcher at the DEC Systems Research Center. He has a Ph.D. in computer science from MIT and is an ACM Distinguished Engineer.

**MENDEL ROSENBLUM** is an associate professor in the computer science and electrical engineering departments at Stanford University. His research interests include system software, distributed systems, and computer architecture. He was cofounder of VMware Inc. and active in the research and development of VMware's virtualization platform. He has a Ph.D. in computer science from UC Berkeley and is an ACM Fellow.