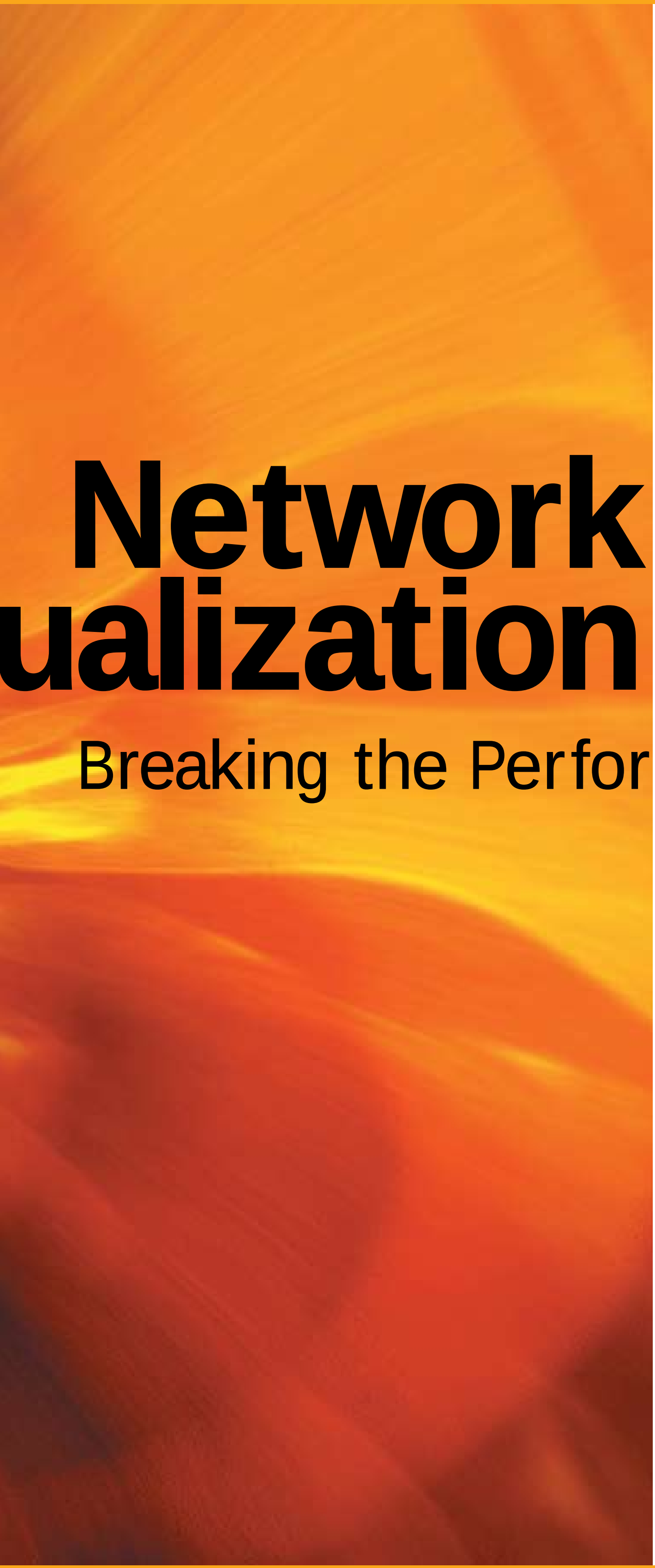
Virt

SCOTT RIXNER, RICE UNIVERSITY



Network ualization

Breaking the Performance Barrier

Shared I/O in virtualization platforms has come a long way, but performance concerns remain.

The recent resurgence in popularity of virtualization has led to its use in a growing number of contexts, many of which require high-performance networking. Consider server consolidation, for example. The efficiency of network virtualization directly impacts the number of network servers that can effectively be consolidated onto a single physical machine. Unfortunately, modern network virtualization techniques incur significant overhead, which limits the achievable network performance. We need new network virtualization techniques to realize the full benefits of virtualization in network-intensive domains.

Network Virtualization
Breaking the Performance Barrier

To share a network interface among a set of virtual machines, the VMM (virtual machine monitor) must accomplish two key tasks. First, the VMM must provide shared access to the network interface. This means that the virtual machines' outgoing network traffic must be multiplexed together before being sent over the network; similarly, incoming network traffic must be demultiplexed before being delivered to the appropriate virtual machines. Second, the VMM must protect the virtual machines from each other. This means that no virtual machine can be allowed to transfer data into or out of another virtual machine's memory. Therefore, the challenge in network virtualization is to provide efficient, shared, and protected access to the network interface.

I/O virtualization is by no means a new concept. The IBM System/360 allowed virtual machines to access I/O devices. In fact, many of the System/360's pioneering mechanisms are still in use today. The prevailing I/O virtualization architectures can be broadly classified into two categories: private assignment of each I/O device to a particular virtual machine or shared assignment of a particular I/O device to multiple virtual machines.

PRIVATE I/O DEVICES

IBM's System/360 was the first widely available virtualization solution.[1] The System/370 extended the virtualization support for processors and memory but continued to use the same mechanisms as the System/360 for I/O virtualization.[2] The initial I/O virtualization architectures developed for these systems did not permit shared access to physical I/O resources. Instead, physical I/O devices, such as terminals, were assigned exclusively to a particular virtual machine. Therefore, each virtual machine required its own physical I/O devices. These systems used channel programs to transfer data to/from I/O devices. The channel programs used programmed I/O to transfer data between memory and the I/O device. On a virtualized system, the channel programs were executed inside the VMM, allowing the VMM to ensure that a virtual machine could access only its own memory and devices.

More recent virtualization systems have also relied on private device access, such as IBM's first release of the LPAR (logical partitioning) architecture for its Power4 processors.[3] The Power4 architecture isolated devices at the PCI-slot level and assigned them to a particular virtual machine instance for management. Each virtual machine required a physically distinct disk controller for disk access and a physically distinct network interface for network access.

Modern I/O devices use DMA (direct memory access) instead of programmed I/O, so it is not possible for the VMM to perform I/O data transfers. Instead, the Power4 LPAR architecture uses an IOMMU (I/O memory management unit) to restrict the memory that can be accessed by each device. To use an IOMMU in this manner, the VMM creates an I/O page table for each device, with memory mappings for the pages owned by the virtual machine to which that device is assigned. The IOMMU then consults the appropriate page table for every DMA operation. If a device tries to access memory without a valid mapping in its I/O page table, then the IOMMU will disallow the access.

Private I/O access has obvious benefits and drawbacks. It yields high-performance I/O access, as each virtual machine can communicate directly with the physical devices that it owns. It is a costly solution, however, since it requires replicated physical devices for each virtual machine. These devices are likely to be underutilized, and for systems with large numbers of virtual machines it may be impossible to include a sufficient number of devices.

SHARED I/O DEVICES

A good way to decrease the costs inherent in providing private I/O devices is to allow I/O devices to be shared among virtual machines. The first shared I/O virtualization solutions were part of the networking support for System/360 and System/370 between physically separated virtual machines.[4] This networking architecture supported shared access to the network using a virtualized spool-file interface that was serviced by a special-purpose virtual machine, or *I/O domain*, dedicated to networking. The other virtual machines in the system could read from or write to virtualized spool files. The VMM would interpret these reads and writes based on whether the spool locations were physically located on the local machine or on a remote machine. If the data were on a remote machine, the VMM would transfer control to the I/O domain, which would then use its physical network interfaces to transfer data to/from a remote machine. The remote machine would use the same virtualized spool architecture and its own dedicated I/O domain to service requests.

This software I/O virtualization architecture is logically identical to most virtualization solutions today. Xen, VMware, and the Power5 virtualization architectures all share access to devices through virtualized software interfaces and rely on a dedicated software entity, such as the I/O domain, to perform physical device management.[5,6,7] The software entity controlling the I/O device can be either a separate I/O domain or the VMM's hypervisor. If there is a separate I/O domain, I/O devices are effectively private to that domain, so memory accesses by the device should be restricted to the I/O domain. This can be enforced either by trusting the I/O domain or by using an IOMMU.

NETWORK VIRTUALIZATION IN SOFTWARE

Since network interfaces are commonly shared in software, it would be instructive to look at a more concrete example of this I/O virtualization architecture. Xen is a popular open source VMM for the x86 architecture that shares I/O devices in software.[8] Although differences exist among VMMs, Xen's I/O virtualization architecture is representative of other systems that support shared I/O devices in software.

Figure 1 shows the organization of the Xen VMM. Xen consists of two elements: the hypervisor and the driver domain. The hypervisor provides an abstraction layer between the virtual machines and the actual hardware, enabling each virtual machine to execute as if it were running natively on the hardware. Shared I/O devices, however, are controlled by a special I/O domain, called the driver domain in Xen. It runs a modified version of Linux that uses native Linux device drivers to manage physical I/O devices. To communicate with the driver domain, and thus the physical I/O devices, each virtual machine is given a *virtual I/O device*, controlled by a special device driver, called a front-end driver. To access a physical device the virtual machine's front-end driver communicates with the corresponding back-end driver in the driver domain.

The back-end network interface drivers are connected to the physical NIC (network interface card) by an Ethernet bridge within the driver domain. When a packet is transmitted by a virtual machine, it is first copied (or remapped) from the front-end driver in the virtual machine to the back-end driver in the driver domain. Within the driver domain, the packet is routed through the Ethernet bridge to the physical NIC's device driver, which then enqueues the packet for transmission on the network interface as if it were generated normally by the operating system within the driver domain.

When a packet is received, the network interface generates an interrupt that is captured by the hypervisor and routed to the NIC's device driver in the driver domain as a virtual interrupt. The NIC's device driver transfers the packet to the Ethernet bridge, which routes the packet to the appropriate back-end driver. The back-end driver then copies (or remaps) the packet to the front-end driver in the target virtual machine. Once the packet is transferred, the back-end driver requests that the hypervisor send an additional virtual interrupt to the target virtual machine

Xen Network Virtualization Architecture

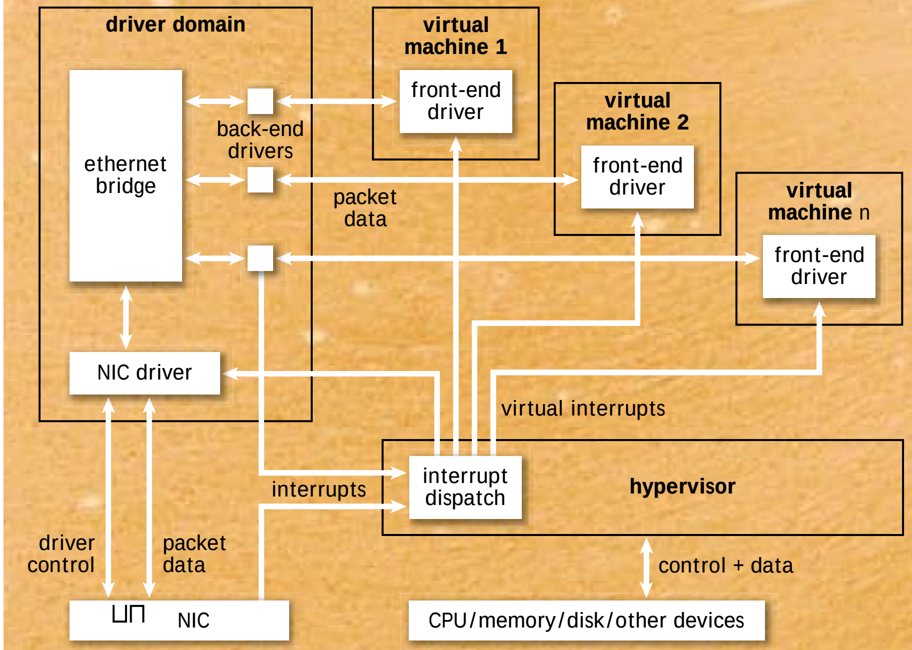


FIG 1

notifying it of the new packet. Upon receiving the virtual interrupt, the front-end driver delivers the packet to the virtual machine's network stack, as if it had come directly from a physical device.

In Xen, the driver domain is responsible for protecting I/O accesses. This means that the driver domain must be trusted to direct the network interface only to use buffers that are owned by the driver domain. Future x86 systems will include an IOMMU that could be used by the hypervisor to enforce that restriction. Regardless, the driver domain must also be trusted to transfer only the appropriate network traffic to each virtual machine.

The Xen network virtualization architecture incurs significant processing and communication overhead. For example, Linux is able to achieve only about one-third of the network throughput inside of Xen that it would be able to achieve running natively.[9] This overhead is not exclusive to Xen; the basic operations that must be performed in software to enable shared I/O devices are common among VMM implementations.

An often-overlooked component of the overhead involved with shared I/O devices is domain scheduling within the VMM. Even if there is only a single virtual machine, sending or receiving a network packet involves two domains: the driver and the virtual machine. These domains must be scheduled, and in the correct order, before a network packet can be sent or received. This means that poor scheduling decisions can result in increased network latency.[10]

Part of the scheduling problem can be eliminated by managing shared I/O devices directly in the hypervisor. VMware ESX[11] and early versions of Xen[12] used this approach. This eliminates the need for scheduling an additional I/O domain in order to transfer a network packet. Promptly scheduling the virtual machine's domain is still necessary, however, though it is not always possible, especially if multiple domains simultaneously have outstanding network traffic. Perhaps more importantly, this requires the device drivers for the network interfaces to be incorporated directly within the hypervisor. This negates one of the often-touted advantages of virtualization: removing the frequently buggy device drivers from the trusted code base of the machine. With the device drivers directly in the hypervisor, the complexity of the hypervisor is significantly increased, and its reliability is correspondingly decreased.

MULTIQUEUE NETWORK INTERFACES

Typically, modern network interfaces include a transmit queue and a receive queue. The transmit queue holds pointers to outgoing network packets, and the receive queue holds pointers to empty buffers that will be used to store incoming network packets. The device driver communicates with the network interface almost exclusively through these queues.

In recent years, network interfaces with multiple sets of these queues (sometimes referred to as *multiqueue network interfaces*) have emerged to improve networking performance in multicore systems. These devices support Microsoft's Receive Side Scaling architecture and Linux's Scalable I/O architecture. These architectures allow each core exclusive access to one of the sets of queues on the network interface. This enables each core to run the network interface's device driver concurrently without the need for additional synchronization. The use of multiqueue network interfaces in this manner increases the achievable parallelism within the network stack of the operating system, enabling more effective use of multiple cores for networking.

Researchers at Hewlett-Packard Laboratories and Citrix have recently come up with a way to use these multiqueue network interfaces to improve the performance of network virtualization in Xen.[13] They eliminate the Ethernet bridge within the driver domain by assigning network interface queues directly to back-end drivers in the driver domain. In this manner, the back-end drivers can communicate with the network interface concurrently without the need for synchronization. The network interface, rather than the Ethernet bridge within the driver domain, will then multiplex/demultiplex network traffic. To multiplex transmit network traffic, the network interface interleaves the network traffic of the virtual machines by servicing all of the transmit queues fairly. When the network interface receives network packets, the NIC uses each packet's destination Ethernet address to identify the appropriate receive queue.

The obvious benefit of using multiqueue network interfaces in this fashion is the elimination of the traffic-multiplexing overhead within the driver domain. A less obvious benefit is the elimination of copying between the driver domain and the virtual machines. Because the network traffic for each virtual machine will interact with the network interface only through a single set of queues, the driver domain can direct the network interface to transfer data directly to/from memory owned by that virtual machine. The virtual machine need only grant the driver domain the right to use its network buffers.

The use of multiqueue network interfaces in this way should improve network virtualization performance by eliminating the multiplexing and copying overhead inherent in sharing I/O devices in software among virtual machines. This technique still has a few drawbacks, however. First, the driver domain must now protect each virtual machine's network traffic by ensuring that the buffers enqueued to a particular queue on the network interface belong to the virtual machine assigned to that queue. Second, overhead and complexity are inherent in managing the buffers that virtual machines have granted to the driver domain to use for networking. Finally, the scheduling problems described earlier remain, as the driver domain must still be scheduled to communicate with the network interface by enqueuing/dequeuing buffers and receiving interrupts.

CONCURRENT, DIRECT NETWORK ACCESS

CDNA (concurrent, direct network access) takes the use of multiqueue network interfaces one step further. In the CDNA network virtualization architecture, the hypervisor directly assigns each virtual machine its own set of queues in a multiqueue network interface.[14] This eliminates the driver domain altogether from networking in Xen.

Figure 2 shows the CDNA network virtualization architecture. As before, the network interface must support multiple sets of queues directly in hardware. Instead of assigning ownership of the entire network interface to the driver domain, however, the hypervisor treats each set of queues as if it were a physical network interface and assigns ownership of each set directly to a virtual machine. Notice the absence of the driver domain from the figure: each virtual machine can transmit and receive network traffic using its own private set of queues without any interaction with other virtual machines or the driver domain. The driver domain, however, is still present to perform control functions and allow access to other I/O devices.

In the CDNA network virtualization architecture, the communication overhead between the driver domain and the virtual machines is eliminated entirely. The hypervisor must now provide protection across the sets of queues on the network interface and deliver interrupts directly to the virtual machines.

Similar to the use of multiqueue network interfaces by the driver domain, CDNA eliminates the software-multiplexing overhead within the driver domain by multiplexing/demultiplexing network traffic on the network interface. As before, each virtual machine is assigned a set of queues and an associated Ethernet address to enable the network interface to perform this task. Interrupts from the network interface, however, are no longer delivered to the appropriate back-end driver in the driver domain. Instead, they are delivered directly to the appropriate virtual machine.

To do so, the hypervisor must understand that the network interface is "owned" by multiple virtual machines. Normally, the hypervisor assumes that each device is owned exclusively by a particular virtual machine. In

CDNA Network Virtualization Architecture

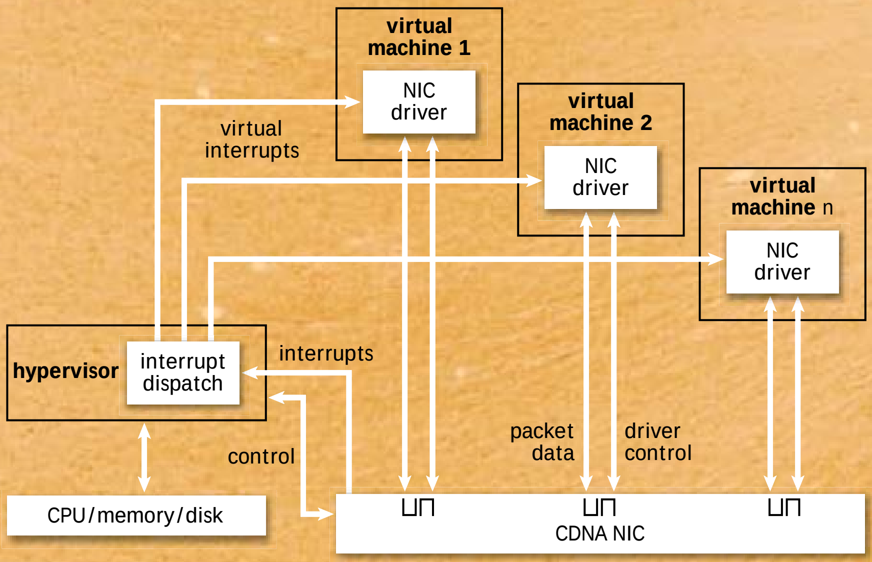


FIG 2

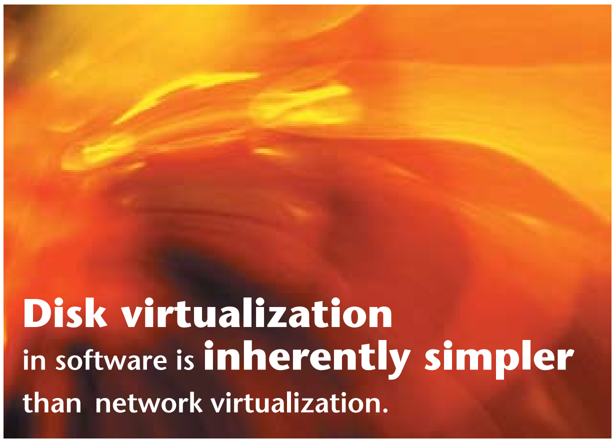
the prototype implementation of CDNA, the network interface delivers a bit vector to the hypervisor and then generates a physical interrupt. Each bit in the bit vector corresponds to one set of queues on the network interface. The hypervisor then scans this bit vector and delivers virtual interrupts to the appropriate virtual machines. Similar behavior could be accomplished using message-signaled interrupts, at the cost of an increased physical interrupt rate. Delivering interrupts directly to the virtual machines, without an intervening trip through the driver domain, reduces the virtual machines' interrupt response time.

The CDNA network virtualization architecture does improve the scheduling issue to some degree. For a virtual machine to transmit or receive network traffic, it does not need the driver domain to be scheduled. This improves overall responsiveness, which leads to improved network performance. The problem of scheduling multiple virtual machines with outstanding network traffic remains, however.

Although the CDNA network virtualization architecture eliminates the overhead of the driver domain, it complicates memory protection. Since the trusted driver domain is no longer moderating access to the network interface, the virtual machines could potentially direct the network interface to access arbitrary memory locations. This problem is particularly acute in the x86 architecture, in which device drivers provide I/O devices with physical addresses to memory. If a faulty or malicious virtual machine were to provide a physical address to the network interface that does not belong to its address space, it could co-opt the network interface into reading or writing memory that is not owned by that virtual machine.

To guarantee memory protection in CDNA, the hypervisor must perform two tasks. First, it must validate all buffers before they are delivered to the network interface. The hypervisor validates a buffer only if it is owned by the virtual machine using it, ensuring that the virtual machine does not enqueue a buffer to the network interface that it does not own. If the network interface receives an unvalidated buffer, that buffer will not be used. Second, the hypervisor must ensure that the ownership of any buffer enqueued on the network interface does not change. In Xen, memory ownership is maintained by using reference counts to a page. In CDNA, a page's reference count is incremented when a buffer within that page is enqueued to the network interface, and it is decremented when the buffer is dequeued. This guarantees that the page's ownership does not change, as Xen will allow the ownership of a page to change only if its reference count is zero.

While the CDNA prototype performs these protection tasks in software, they can be simplified with the use of an IOMMU. In a system with an IOMMU, the hypervisor would create an I/O page table for each virtual machine, with memory mappings for the pages that the virtual machine is allowed to use for I/O operations (potentially all of the virtual machine's memory). The hypervisor would update these I/O page tables whenever memory ownership changed. This would obviate the need to


Disk virtualization
in software is inherently simpler
than network virtualization.

validate each buffer individually and to track the ownership of network buffers. Instead, the IOMMU would consult the I/O page tables for every network transfer to ensure that each memory access made by the network interface was allowed. For this to work, however, the IOMMU must be able to differentiate memory accesses made by the network interface on behalf of the different virtual machines. The IOMMU needs this information to determine which I/O page table should be consulted for a particular access. Current PCI bus implementations offer no obvious method for a network interface to differentiate its memory accesses in this way, but the upcoming PCI I/O virtualization specification should include such a capability.

Figure 3 shows the achievable aggregate bandwidth using both the Xen and CDNA network virtualization architectures as the number of virtual machines is varied. Only two network interfaces are used, limiting the bandwidth to 2 Gbps. As the figure shows, direct access to the network interface yields dramatic performance improvement. In fact, the figure understates the overall improvement, as there is idle time when using the CDNA network virtualization architecture. These idle processing resources could be used to achieve even higher network bandwidth if there were additional network interfaces in the system. In contrast, there is never idle time when using the Xen network virtualization architecture, so the figures show the peak network bandwidth of Xen.

DISK VS. NETWORK I/O

This article has focused on network virtualization, because it is the most complicated type of I/O virtualization. Since network traffic is unsolicited, it is more difficult to share a network interface efficiently than it is with other I/O devices. The system must be prepared to receive and respond to network traffic destined for any virtual machine at any time. This differs from the virtualization of many other hardware resources in that the VMM has far less control over the use of the network. Disk access, for example, is more controlled, in that disk reads and writes occur only when explicitly requested by a virtual machine.

In software I/O virtualization architectures, such as Xen, a virtual machine can always provide the driver domain with a buffer when it initiates a disk access. This eliminates the need to copy (or remap) data between the virtual machine and driver domain. Similarly, the equivalent of the Ethernet bridge that was required for networking is unnecessary, as the destination of all disk read traffic is known ahead of time. Therefore, disk virtualization in software is inherently simpler than network virtualization.

Furthermore, IBM used mini-disks and the DASD (direct access storage device) architecture to improve disk virtualization and avoid the need for an expensive, dedicated disk for each virtual machine. The System/360 partitioned the disk into logically separate, virtual mini-disks.[15] Though multiple virtual machines could access the same physical disk via the mini-disk abstraction, they did not concurrently share access to the same mini-disk region. This reduced the number of disks needed in the system, but was still a form of private I/O access, as disk blocks could not truly be shared among virtual machines.

The System/360 and System/370 also implemented the DASD architecture to enable concurrent, direct disk access.[16] As previously described, I/O devices in these early IBM systems were accessed using programmed I/O via channel programs. DASD-capable devices had several separately addressed channels, so multiple channel programs could execute in parallel. Disk access was still synchronous, however, so only one channel program at a time could actually access the disk. Additional channel programs could perform data comparisons or address calculations at the same time, significantly improving I/O throughput. On a virtualized system, the virtual machines' channel programs would be run inside the hypervisor, allowing the hypervisor to ensure that a virtual machine could access only its own memory and devices.[17]

Although these techniques improve disk virtualization performance, they cannot be used directly for modern network virtualization. First, the network cannot be partitioned in the same way as a disk, so the mini-disk abstraction has no logical equivalent for networking.

Continues on page 52

Performance of CDNA and Xen Network Virtualization Architectures

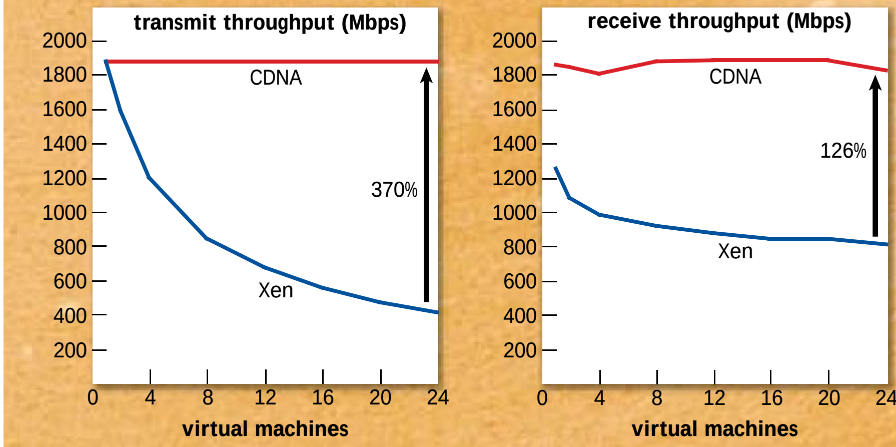


FIG 3

Network Virtualization

Breaking the Performance Barrier

Continued from page 43

Second, DASD requires a channel program to run for each synchronous disk access. This is not compatible with the asynchronous receipt of network packets. Furthermore, CDNA builds upon the DASD concept by allowing protected, asynchronous, and concurrent network access.

THE PROMISE OF CDNA

For the past several decades, virtual machine monitors have shared I/O devices in software. Although such software I/O virtualization architectures are simple and flexible, they create a significant network performance barrier, which severely limits the value of virtualization for network-intensive domains. To realize the full potential of virtualization, we need more efficient network virtualization architectures.

Given the unsolicited nature of network traffic, it is valuable to perform traffic multiplexing and demultiplexing directly on the network interface. This capability has already shown promise in improving the network efficiency of multicore systems. CDNA is an upcoming network virtualization architecture that further uses this capability to provide many of the performance benefits of using private network interfaces, without requiring a large number of underutilized network interfaces. This will dramatically increase the efficiency of virtualized systems and increase the number of network-intensive virtual machines that each physical machine can support. Q

REFERENCES

1. Parmelee, R. P., Peterson, T. I., Tillman, C. C., Hatfield, D. J. 1972. Virtual storage and virtual machine concepts. *IBM Systems Journal* 11(2): 99–130.
2. Gum, P. H. 1983. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development* 27(6): 530–544.
3. Jann, J., Browning, L. M., Burugula, R.S. 2003. Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pseries servers. *IBM Systems Journal* 42(1): 29–37.
4. MacKinnon, R. A. 1979. The changing virtual machine environment: Interfaces to real hardware, virtual hardware, and other virtual machines. *IBM Systems Journal* 18(1): 18–46.
5. Armstrong, W. J. Arndt, R. L. Boutcher, D. C., Kovacs, R. G., Larson, D., Lucke, K. A., Nayar, N., Swanberg, R.C. 2005. Advanced virtualization capabilities of Power5 systems. *IBM Journal of Research and Development* 49(4/5): 523–532.
6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A. 2003. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles* (October).
7. Sugerman, J., Venkitachalam, G., Lim, B. 2001. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the Usenix Annual Technical Conference* (June).
8. See reference 6.
9. Willmann, P., Shafer, J., Carr, D., Menon, A., Rixner, S., Cox, A. L., Zwaenepoel, W. 2007. Concurrent direct network access for virtual machine monitors. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (February).
10. Ongaro, D., Cox, A. L., Rixner, S. 2008. Scheduling I/O in virtual machine monitors. In *Proceedings of the International Conference on Virtual Execution Environments* (March).
11. VMware Inc. 2006. VMware ESX server: Platform for virtualizing servers, storage and networking; http://www.vmware.com/pdf/esx_datasheet.pdf.
12. See reference 6.
13. Santos, J. R., Janakiraman, G., Turner, Y., Pratt, I. 2007. Netchannel 2: Optimizing network performance. Xen Summit Talk (November).
14. See reference 9.
15. See reference 1.
16. Brown, D. T., Eibsen, R. L., Thorn, C. A. 1972. Channel and direct access device architecture. *IBM Systems Journal* 11(3): 186-199.
17. See reference 2.

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

SCOTT RIXNER is an associate professor of computer science and electrical and computer engineering at Rice University. He received a Ph.D. in electrical engineering from the Massachusetts Institute of Technology and is a member of the ACM. His research interests include network systems architecture, memory systems architecture, and the interaction between operating systems and computer architectures.