Concurrent Direct Network Access for Virtual Machine Monitors

Paul Willmann[†] Jeffrey Shafer[†] David Carr[†] Aravind Menon[‡]
Scott Rixner[†] Alan L. Cox[†] Willy Zwaenepoel[‡]

[†]Rice University
Houston, TX
{willmann,shafer,dcarr,rixner,alc}@rice.edu

[‡]EPFL
Lausanne, Switzerland
{aravind.menon,willy.zwaenepoel}@epfl.ch

Abstract

This paper presents hardware and software mechanisms to enable concurrent direct network access (CDNA) by operating systems running within a virtual machine monitor. In a conventional virtual machine monitor, each operating system running within a virtual machine must access the network through a software-virtualized network interface. These virtual network interfaces are multiplexed in software onto a physical network interface, incurring significant performance overheads. The CDNA architecture improves networking efficiency and performance by dividing the tasks of traffic multiplexing, interrupt delivery, and memory protection between hardware and software in a novel way. The virtual machine monitor delivers interrupts and provides protection between virtual machines, while the network interface performs multiplexing of the network data. In effect, the CDNA architecture provides the abstraction that each virtual machine is connected directly to its own network interface. Through the use of CDNA, many of the bottlenecks imposed by software multiplexing can be eliminated without sacrificing protection, producing substantial efficiency improvements.

1 Introduction

In many organizations, the economics of supporting a growing number of Internet-based services has created a demand for server consolidation. Consequently, there has been a resurgence of interest in machine virtualization [1, 2, 4, 7, 9, 10, 11, 19, 22]. A virtual machine monitor (VMM) enables multiple virtual machines, each encapsulating one or more services, to share the same physical machine safely and fairly. In principle, general-purpose operating systems, such as Unix and Windows, offer the same capability for multiple services to share the same physical machine. However, VMMs provide additional advantages. For example, VMMs allow services implemented in different or customized environments, including different operating systems, to share the same physical machine.

Modern VMMs for commodity hardware, such as VMWare [1, 7] and Xen [4], virtualize processor, memory, and I/O devices in software. This enables these VMMs to support a variety of hardware. In an attempt to decrease the software overhead of virtualization, both AMD and Intel are introducing hardware support for virtualization [2, 10]. Specifically, their hardware support for processor virtualization is currently available, and their hardware support for memory virtualization is imminent. As these hardware mechanisms mature, they should reduce the overhead of virtualization, improving the efficiency of VMMs.

Despite the renewed interest in system virtualization, there is still no clear solution to improve the efficiency of I/O virtualization. To support networking, a VMM must present each virtual machine with a virtual network interface that is multiplexed in software onto a physical network interface card (NIC). The overhead of this software-based network virtualization severely limits network performance [12, 13, 19]. For example, a Linux kernel running within a virtual machine on Xen is only able to achieve about 30% of the network throughput that the same kernel can achieve running directly on the physical machine.

This paper proposes and evaluates concurrent direct network access (CDNA), a new I/O virtualization technique combining both software and hardware components that significantly reduces the overhead of network virtualization in VMMs. The CDNA network virtualization architecture provides virtual machines running on a VMM safe direct access to the network interface. With CDNA, each virtual machine is allocated a unique *context* on the network interface and communicates directly with the network interface through that context. In this manner, the virtual machines

This work was supported in part by the Texas Advanced Technology Program under Grant No. 003604-0078-2003, by the National Science Foundation under Grant No. CCF-0546140, by a grant from the Swiss National Science Foundation, and by gifts from Advanced Micro Devices, Hewlett-Packard, and Xilinx.

that run on the VMM operate as if each has access to its own dedicated network interface.

Using CDNA, a single virtual machine running Linux can transmit at a rate of 1867 Mb/s with 51% idle time and receive at a rate of 1874 Mb/s with 41% idle time. In contrast, at 97% CPU utilization, Xen is only able to achieve 1602 Mb/s for transmit and 1112 Mb/s for receive. Furthermore, with 24 virtual machines, CDNA can still transmit and receive at a rate of over 1860 Mb/s, but with no idle time. In contrast, Xen is only able to transmit at a rate of 891 Mb/s and receive at a rate of 558 Mb/s with 24 virtual machines.

The CDNA network virtualization architecture achieves this dramatic increase in network efficiency by dividing the tasks of traffic multiplexing, interrupt delivery, and memory protection among hardware and software in a novel way. Traffic multiplexing is performed directly on the network interface, whereas interrupt delivery and memory protection are performed by the VMM with support from the network interface. This division of tasks into hardware and software components simplifies the overall software architecture, minimizes the hardware additions to the network interface, and addresses the network performance bottlenecks of Xen.

The remainder of this paper proceeds as follows. The next section discusses networking in the Xen VMM in more detail. Section 3 describes how CDNA manages traffic multiplexing, interrupt delivery, and memory protection in software and hardware to provide concurrent access to the NIC. Section 4 then describes the custom hardware NIC that facilitates concurrent direct network access on a single device. Section 5 presents the experimental methodology and results. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 Networking in Xen

2.1 Hypervisor and Driver Domain Operation

A VMM allows multiple guest operating systems, each running in a virtual machine, to share a single physical machine safely and fairly. It provides isolation between these guest operating systems and manages their access to hardware resources. Xen is an open source VMM that supports paravirtualization, which requires modifications to the guest operating system [4]. By modifying the guest operating systems to interact with the VMM, the complexity of the VMM can be reduced and overall system performance improved.

Xen performs three key functions in order to provide virtual machine environments. First, Xen allocates the physical resources of the machine to the guest operating systems and isolates them from each other. Second, Xen receives all

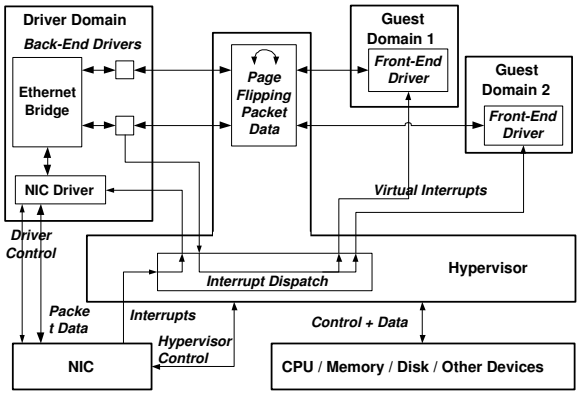

Figure 1. Xen virtual machine environment.

interrupts in the system and passes them on to the guest operating systems, as appropriate. Finally, all I/O operations go through Xen in order to ensure fair and non-overlapping access to I/O devices by the guests.

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, called guest domains, and the actual hardware, enabling each guest operating system to execute as if it were the only operating system on the machine. However, the guest operating systems cannot directly communicate with the physical I/O devices. Exclusive access to the physical devices is given by the hypervisor to the driver domain, a privileged virtual machine. Each guest operating system is then given a *virtual I/O device* that is controlled by a paravirtualized driver, called a front-end driver. In order to access a physical device, such as the network interface card (NIC), the guest's front-end driver communicates with the corresponding back-end driver in the driver domain. The driver domain then multiplexes the data streams for each guest onto the physical device. The driver domain runs a modified version of Linux that uses native Linux device drivers to manage I/O devices.

As the figure shows, in order to provide network access to the guest domains, the driver domain includes a software Ethernet bridge that interconnects the physical NIC and all of the virtual network interfaces. When a packet is transmitted by a guest, it is first transferred to the back-end driver in the driver domain using a page remapping operation. Within the driver domain, the packet is then routed through the Ethernet bridge to the physical device driver. The device driver 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 network interface's device driver in the driver domain as a virtual interrupt. The network interface'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 transfers the packet to the front-end driver in the guest domain using a page remapping operation. Once the packet is transferred, the back-end driver requests that the hypervisor send a virtual interrupt to the guest notifying it of the new packet. Upon receiving the virtual interrupt, the front-end driver delivers the packet to the guest operating system's network stack, as if it had come directly from the physical device.

2.2 Device Driver Operation

The driver domain in Xen is able to use unmodified Linux device drivers to access the network interface. Thus, all interactions between the device driver and the NIC are as they would be in an unvirtualized system. These interactions include programmed I/O (PIO) operations from the driver to the NIC, direct memory access (DMA) transfers by the NIC to read or write host memory, and physical interrupts from the NIC to invoke the device driver.

The device driver directs the NIC to send packets from buffers in host memory and to place received packets into preallocated buffers in host memory. The NIC accesses these buffers using DMA read and write operations. In order for the NIC to know where to store or retrieve data from the host, the device driver within the host operating system generates DMA descriptors for use by the NIC. These descriptors indicate the buffer's length and physical address on the host. The device driver notifies the NIC via PIO that new descriptors are available, which causes the NIC to retrieve them via DMA transfers. Once the NIC reads a DMA descriptor, it can either read from or write to the associated buffer, depending on whether the descriptor is being used by the driver to transmit or receive packets.

Device drivers organize DMA descriptors in a series of rings that are managed using a producer/consumer protocol. As they are updated, the producer and consumer pointers wrap around the rings to create a continuous circular buffer. There are separate rings of DMA descriptors for transmit and receive operations. Transmit DMA descriptors point to host buffers that will be transmitted by the NIC, whereas receive DMA descriptors point to host buffers that the OS wants the NIC to use as it receives packets. When the host driver wants to notify the NIC of the availability of a new DMA descriptor (and hence a new packet to be transmitted or a new buffer to be posted for packet reception), the driver first creates the new DMA descriptor in the next-available slot in the driver's descriptor ring and then increments the producer index on the NIC to reflect that a new descriptor is available. The driver updates the NIC's producer index by writing the value via PIO into a specific location, called a *mailbox*, within the device's PCI memory-mapped region.

System	Transmit (Mb/s)	Receive (Mb/s)
Native Linux	5126	3629
Xen Guest	1602	1112

Table 1. Transmit and receive performance for native Linux 2.6.16.29 and paravirtualized Linux 2.6.16.29 as a guest OS within Xen 3.

The network interface monitors these mailboxes for such writes from the host. When a mailbox update is detected, the NIC reads the new producer value from the mailbox, performs a DMA read of the descriptor indicated by the index, and then is ready to use the DMA descriptor. After the NIC consumes a descriptor from a ring, the NIC updates its consumer index, transfers this consumer index to a location in host memory via DMA, and raises a physical interrupt to notify the host that state has changed.

In an unvirtualized operating system, the network interface trusts that the device driver gives it valid DMA descriptors. Similarly, the device driver trusts that the NIC will use the DMA descriptors correctly. If either entity violates this trust, physical memory can be corrupted. Xen also requires this trust relationship between the device driver in the driver domain and the NIC.

2.3 Performance

Despite the optimizations within the paravirtualized drivers to support communication between the guest and driver domains (such as using page remapping rather than copying to transfer packets), Xen introduces significant processing and communication overheads into the network transmit and receive paths. Table 1 shows the networking performance of both native Linux 2.6.16.29 and paravirtualized Linux 2.6.16.29 as a guest operating system within Xen 3 Unstable[1] on a modern Opteron-based system with six Intel Gigabit Ethernet NICs. In both configurations, checksum offloading, scatter/gather I/O, and TCP Segmentation Offloading (TSO) were enabled. Support for TSO was recently added to the unstable development branch of Xen and is not currently available in the Xen 3 release. As the table shows, a guest domain within Xen is only able to achieve about 30% of the performance of native Linux. This performance gap strongly motivates the need for networking performance improvements within Xen.

3 Concurrent Direct Network Access

With CDNA, the network interface and the hypervisor collaborate to provide the abstraction that each guest operating system is connected directly to its own network in-

[1] Changeset 12053:874cc0ff214d from 11/1/2006.

terface. This eliminates many of the overheads of network virtualization in Xen. Figure 2 shows the CDNA architecture. The network interface must support multiple *contexts* in hardware. Each context acts as if it is an independent physical network interface and can be controlled by a separate device driver instance. Instead of assigning ownership of the entire network interface to the driver domain, the hypervisor treats each context as if it were a physical NIC and assigns ownership of contexts to guest operating systems. Notice the absence of the driver domain from the figure: each guest can transmit and receive network traffic using its own private context without any interaction with other guest operating systems or the driver domain. The driver domain, however, is still present to perform control functions and allow access to other I/O devices. Furthermore, the hypervisor is still involved in networking, as it must guarantee memory protection and deliver virtual interrupts to the guest operating systems.

With CDNA, the communication overheads between the guest and driver domains and the software multiplexing overheads within the driver domain are eliminated entirely. However, the network interface now must multiplex the traffic across all of its active contexts, and the hypervisor must provide protection across the contexts. The following sections describe how CDNA performs traffic multiplexing, interrupt delivery, and DMA memory protection.

3.1 Multiplexing Network Traffic

CDNA eliminates the software multiplexing overheads within the driver domain by multiplexing network traffic on the NIC. The network interface must be able to identify the source or target guest operating system for all network traffic. The network interface accomplishes this by providing independent hardware contexts and associating a unique Ethernet MAC address with each context. The hypervisor assigns a unique hardware context on the NIC to each guest operating system. The device driver within the guest operating system then interacts with its context exactly as if the context were an independent physical network interface. As described in Section 2.2, these interactions consist of creating DMA descriptors and updating a mailbox on the NIC via PIO.

Each context on the network interface therefore must include a unique set of mailboxes. This isolates the activity of each guest operating system, so that the NIC can distinguish between the different guests. The hypervisor assigns a context to a guest simply by mapping the I/O locations for that context's mailboxes into the guest's address space. The hypervisor also notifies the NIC that the context has been allocated and is active. As the hypervisor only maps each context into a single guest's address space, a guest cannot accidentally or intentionally access any context on the NIC

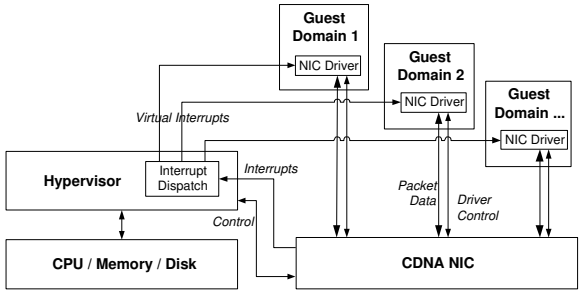

Figure 2. CDNA architecture in Xen.

other than its own. When necessary, the hypervisor can also revoke a context at any time by notifying the NIC, which will shut down all pending operations associated with the indicated context.

To multiplex transmit network traffic, the NIC simply services all of the hardware contexts fairly and interleaves the network traffic for each guest. When network packets are received by the NIC, it uses the Ethernet MAC address to demultiplex the traffic, and transfers each packet to the appropriate guest using available DMA descriptors from that guest's context.

3.2 Interrupt Delivery

In addition to isolating the guest operating systems and multiplexing network traffic, the hardware contexts on the NIC must also be able to interrupt their respective guests. As the NIC carries out network requests on behalf of any particular context, the CDNA NIC updates that context's consumer pointers for the DMA descriptor rings, as described in Section 2.2. Normally, the NIC would then interrupt the guest to notify it that the context state has changed. However, in Xen all physical interrupts are handled by the hypervisor. Therefore, the NIC cannot physically interrupt the guest operating systems directly. Even if it were possible to interrupt the guests directly, that could create a much higher interrupt load on the system, which would decrease the performance benefits of CDNA.

Under CDNA, the NIC keeps track of which contexts have been updated since the last physical interrupt, encoding this set of contexts in an interrupt bit vector. The NIC transfers an interrupt bit vector into the hypervisor's memory space using DMA. The interrupt bit vectors are stored in a circular buffer using a producer/consumer protocol to ensure that they are processed by the host before being overwritten by the NIC. After an interrupt bit vector is transferred, the NIC raises a physical interrupt, which invokes the hypervisor's interrupt service routine. The hypervisor then decodes all of the pending interrupt bit vectors and schedules virtual interrupts to each of the guest operating systems that have pending updates from the NIC. When the

guest operating systems are next scheduled by the hypervisor, the CDNA network interface driver within the guest receives these virtual interrupts as if they were actual physical interrupts from the hardware. At that time, the driver examines the updates from the NIC and determines what further action, such as processing received packets, is required.

3.3 DMA Memory Protection

In the x86 architecture, network interfaces and other I/O devices use physical addresses when reading or writing host system memory. The device driver in the host operating system is responsible for doing virtual-to-physical address translation for the device. The physical addresses are provided to the network interface through read and write DMA descriptors as discussed in Section 2.2. By exposing physical addresses to the network interface, the DMA engine on the NIC can be co-opted into compromising system security by a buggy or malicious driver. There are two key I/O protection violations that are possible in the x86 architecture. First, the device driver could instruct the NIC to transmit packets containing a payload from physical memory that does not contain packets generated by the operating system, thereby creating a security hole. Second, the device driver could instruct the NIC to receive packets into physical memory that was not designated as an available receive buffer, possibly corrupting memory that is in use.

In the conventional Xen network architecture discussed in Section 2.2, Xen trusts the device driver in the driver domain to only use the physical addresses of network buffers in the driver domain's address space when passing DMA descriptors to the network interface. This ensures that all network traffic will be transferred to/from network buffers within the driver domain. Since guest domains do not interact with the NIC, they cannot initiate DMA operations, so they are prevented from causing either of the I/O protection violations in the x86 architecture.

Though the Xen I/O architecture guarantees that untrusted guest domains cannot induce memory protection violations, any domain that is granted access to an I/O device by the hypervisor can potentially direct the device to perform DMA operations that access memory belonging to other guests, or even the hypervisor. The Xen architecture does not fundamentally solve this security defect but instead limits the scope of the problem to a single, trusted driver domain [9]. Therefore, as the driver domain is trusted, it is unlikely to intentionally violate I/O memory protection, but a buggy driver within the driver domain could do so unintentionally.

This solution is insufficient for the CDNA architecture. In a CDNA system, device drivers in the guest domains have direct access to the network interface and are able to pass DMA descriptors with physical addresses to the device. Thus, the untrusted guests could read or write memory in any other domain through the NIC, unless additional security features are added. To maintain isolation between guests, the CDNA architecture validates and protects all DMA descriptors and ensures that a guest maintains ownership of physical pages that are sources or targets of outstanding DMA accesses. Although the hypervisor and the network interface share the responsibility for implementing these protection mechanisms, the more complex aspects are implemented in the hypervisor.

The most important protection provided by CDNA is that it does not allow guest domains to directly enqueue DMA descriptors into the network interface descriptor rings. Instead, the device driver in each guest must call into the hypervisor to perform the enqueue operation. This allows the hypervisor to validate that the physical addresses provided by the guest are, in fact, owned by that guest domain. This prevents a guest domain from arbitrarily transmitting from or receiving into another guest domain. The hypervisor prevents guest operating systems from independently enqueueing unauthorized DMA descriptors by establishing the hypervisor's exclusive write access to the host memory region containing the CDNA descriptor rings during driver initialization.

As discussed in Section 2.2, conventional I/O devices autonomously fetch and process DMA descriptors from host memory at runtime. Though hypervisor-managed validation and enqueuing of DMA descriptors ensures that DMA operations are valid when they are enqueued, the physical memory could still be reallocated before it is accessed by the network interface. There are two ways in which such a protection violation could be exploited by a buggy or malicious device driver. First, the guest could return the memory to the hypervisor to be reallocated shortly after enqueueing the DMA descriptor. Second, the guest could attempt to reuse an old DMA descriptor in the descriptor ring that is no longer valid.

When memory is freed by a guest operating system, it becomes available for reallocation to another guest by the hypervisor. Hence, ownership of the underlying physical memory can change dynamically at runtime. However, it is critical to prevent any possible reallocation of physical memory during a DMA operation. CDNA achieves this by delaying the reallocation of physical memory that is being used in a DMA transaction until after that pending DMA has completed. When the hypervisor enqueues a DMA descriptor, it first establishes that the requesting guest owns the physical memory associated with the requested DMA. The hypervisor then increments the reference count for each physical page associated with the requested DMA. This per-page reference counting system already exists within the Xen hypervisor; so long as the reference count is non-zero, a physical page cannot be reallocated. Later, the hypervisor

then observes which DMA operations have completed and decrements the associated reference counts. For efficiency, the reference counts are only decremented when additional DMA descriptors are enqueued, but there is no reason why they could not be decremented more aggressively, if necessary.

After enqueuing DMA descriptors, the device driver notifies the NIC by writing a producer index into a mailbox location within that guest's context on the NIC. This producer index indicates the location of the last of the newly created DMA descriptors. The NIC then assumes that all DMA descriptors up to the location indicated by the producer index are valid. If the device driver in the guest increments the producer index past the last valid descriptor, the NIC will attempt to use a stale DMA descriptor that is in the descriptor ring. Since that descriptor was previously used in a DMA operation, the hypervisor may have decremented the reference count on the associated physical memory and reallocated the physical memory.

To prevent such stale DMA descriptors from being used, the hypervisor writes a strictly increasing sequence number into each DMA descriptor. The NIC then checks the sequence number before using any DMA descriptor. If the descriptor is valid, the sequence numbers will be continuous modulo the size of the maximum sequence number. If they are not, the NIC will refuse to use the descriptors and will report a guest-specific protection fault error to the hypervisor. Because each DMA descriptor in the ring buffer gets a new, increasing sequence number, a stale descriptor will have a sequence number exactly equal to the correct value minus the number of descriptor slots in the buffer. Making the maximum sequence number at least twice as large as the number of DMA descriptors in a ring buffer prevents aliasing and ensures that any stale sequence number will be detected.

3.4 Discussion

The CDNA interrupt delivery mechanism is neither device nor Xen specific. This mechanism only requires the device to transfer an interrupt bit vector to the hypervisor via DMA prior to raising a physical interrupt. This is a relatively simple mechanism from the perspective of the device and is therefore generalizable to a variety of virtualized I/O devices. Furthermore, it does not rely on any Xen-specific features.

The handling of the DMA descriptors within the hypervisor is linked to a particular network interface only because the format of the DMA descriptors and their rings is likely to be different for each device. As the hypervisor must validate that the host addresses referred to in each descriptor belong to the guest operating system that provided them, the hypervisor must be aware of the descriptor format. Fortunately, there are only three fields of interest in any DMA descriptor: an address, a length, and additional flags. This commonality should make it possible to generalize the mechanisms within the hypervisor by having the NIC notify the hypervisor of its preferred format. The NIC would only need to specify the size of the descriptor and the location of the address, length, and flags. The hypervisor would not need to interpret the flags, so they could just be copied into the appropriate location. A generic NIC would also need to support the use of sequence numbers within each DMA descriptor. Again, the NIC could notify the hypervisor of the size and location of the sequence number field within the descriptors.

CDNA's DMA memory protection is specific to Xen only insofar as Xen permits guest operating systems to use physical memory addresses. Consequently, the current implementation must validate the ownership of those physical addresses for every requested DMA operation. For VMMs that only permit the guest to use virtual addresses, the hypervisor could just as easily translate those virtual addresses and ensure physical contiguity. The current CDNA implementation does not rely on physical addresses in the guest at all; rather, a small library translates the driver's virtual addresses to physical addresses within the guest's driver before making a hypercall request to enqueue a DMA descriptor. For VMMs that use virtual addresses, this library would do nothing.

4 CDNA NIC Implementation

To evaluate the CDNA concept in a real system, RiceNIC, a programmable and reconfigurable FPGA-based Gigabit Ethernet network interface [17], was modified to provide virtualization support. RiceNIC contains a Virtex-II Pro FPGA with two embedded 300MHz PowerPC processors, hundreds of megabytes of on-board SRAM and DRAM memories, a Gigabit Ethernet PHY, and a 64-bit/66 MHz PCI interface [3]. Custom hardware assist units for accelerated DMA transfers and MAC packet handling are provided on the FPGA. The RiceNIC architecture is similar to the architecture of a conventional network interface. With basic firmware and the appropriate Linux or FreeBSD device driver, it acts as a standard Gigabit Ethernet network interface that is capable of fully saturating the Ethernet link while only using one of the two embedded processors.

To support CDNA, both the hardware and firmware of the RiceNIC were modified to provide multiple protected contexts and to multiplex network traffic. The network interface was also modified to interact with the hypervisor through a dedicated context to allow privileged management operations. The modified hardware and firmware components work together to implement the CDNA inter-

faces.

To support CDNA, the most significant addition to the network interface is the specialized use of the 2 MB SRAM on the NIC. This SRAM is accessible via PIO from the host. For CDNA, 128 KB of the SRAM is divided into 32 partitions of 4 KB each. Each of these partitions is an interface to a separate hardware *context* on the NIC. Only the SRAM can be memory mapped into the host's address space, so no other memory locations on the NIC are accessible via PIO. As a context's memory partition is the same size as a page on the host system and because the region is page-aligned, the hypervisor can trivially map each context into a different guest domain's address space. The device drivers in the guest domains may use these 4 KB partitions as general purpose shared memory between the corresponding guest operating system and the network interface.

Within each context's partition, the lowest 24 memory locations are mailboxes that can be used to communicate from the driver to the NIC. When any mailbox is written by PIO, a global mailbox event is automatically generated by the FPGA hardware. The NIC firmware can then process the event and efficiently determine which mailbox and corresponding context has been written by decoding a two-level hierarchy of bit vectors. All of the bit vectors are generated automatically by the hardware and stored in a data scratchpad for high speed access by the processor. The first bit vector in the hierarchy determines which of the 32 potential contexts have updated mailbox events to process, and the second vector in the hierarchy determines which mailbox(es) in a particular context have been updated. Once the specific mailbox has been identified, that off-chip SRAM location can be read by the firmware and the mailbox information processed.

The mailbox event and associated hierarchy of bit vectors are managed by a small hardware core that snoops data on the SRAM bus and dispatches notification messages when a mailbox is updated. A small state machine decodes these messages and incrementally updates the data scratchpad with the modified bit vectors. This state machine also handles event-clear messages from the processor that can clear multiple events from a single context at once.

Each context requires 128 KB of storage on the NIC for metadata, such as the rings of transmit- and receive-DMA descriptors provided by the host operating systems. Furthermore, each context uses 128 KB of memory on the NIC for buffering transmit packet data and 128 KB for receive packet data. However, the NIC's transmit and receive packet buffers are each managed globally, and hence packet buffering is shared across all contexts.

The modifications to the RiceNIC to support CDNA were minimal. The major hardware change was the additional mailbox storage and handling logic. This could easily be added to an existing NIC without interfering with the normal operation of the network interface—unvirtualized device drivers would use a single context's mailboxes to interact with the base firmware. Furthermore, the computation and storage requirements of CDNA are minimal. Only one of the RiceNIC's two embedded processors is needed to saturate the network, and only 12 MB of memory on the NIC is needed to support 32 contexts. Therefore, with minor modifications, commodity network interfaces could easily provide sufficient computation and storage resources to support CDNA.

5 Evaluation

5.1 Experimental Setup

The performance of Xen and CDNA network virtualization was evaluated on an AMD Opteron-based system running Xen 3 Unstable[2]. This system used a Tyan S2882 motherboard with a single Opteron 250 processor and 4GB of DDR400 SDRAM. Xen 3 Unstable was used because it provides the latest support for high-performance networking, including TCP segmentation offloading, and the most recent version of Xenoprof [13] for profiling the entire system.

In all experiments, the driver domain was configured with 256 MB of memory and each of 24 guest domains were configured with 128 MB of memory. Each guest domain ran a stripped-down Linux 2.6.16.29 kernel with minimal services for memory efficiency and performance. For the base Xen experiments, a single dual-port Intel Pro/1000 MT NIC was used in the system. In the CDNA experiments, two RiceNICs configured to support CDNA were used in the system. Linux TCP parameters and NIC coalescing options were tuned in the driver domain and guest domains for optimal performance. For all experiments, checksum offloading and scatter/gather I/O were enabled. TCP segmentation offloading was enabled for experiments using the Intel NICs, but disabled for those using the RiceNICs due to lack of support. The Xen system was setup to communicate with a similar Opteron system that was running a native Linux kernel. This system was tuned so that it could easily saturate two NICs both transmitting and receiving so that it would never be the bottleneck in any of the tests.

To validate the performance of the CDNA approach, multiple simultaneous connections across multiple NICs to multiple guests domains were needed. A multithreaded, event-driven, lightweight network benchmark program was developed to distribute traffic across a configurable number of connections. The benchmark program balances the bandwidth across all connections to ensure fairness and uses a single buffer per thread to send and receive data to minimize the memory footprint and improve cache performance.

[2]Changeset 12053:874cc0ff214d from 11/1/2006.

System	NIC	Mb/s	Domain Execution Profile						Interrupts/s	
			Hyp	Driver Domain		Guest OS		Idle	Driver Domain	Guest OS
				OS	User	OS	User			
Xen	Intel	1602	19.8%	35.7%	0.8%	39.7%	1.0%	3.0%	7,438	7,853
Xen	RiceNIC	1674	13.7%	41.5%	0.5%	39.5%	1.0%	3.8%	8,839	5,661
CDNA	RiceNIC	1867	10.2%	0.3%	0.2%	37.8%	0.7%	50.8%	0	13,659

Table 2. Transmit performance for a single guest with 2 NICs using Xen and CDNA.

System	NIC	Mb/s	Domain Execution Profile						Interrupts/s	
			Hyp	Driver Domain		Guest OS		Idle	Driver Domain	Guest OS
				OS	User	OS	User			
Xen	Intel	1112	25.7%	36.8%	0.5%	31.0%	1.0%	5.0%	11,138	5,193
Xen	RiceNIC	1075	30.6%	39.4%	0.6%	28.8%	0.6%	0%	10,946	5,163
CDNA	RiceNIC	1874	9.9%	0.3%	0.2%	48.0%	0.7%	40.9%	0	7,402

Table 3. Receive performance for a single guest with 2 NICs using Xen and CDNA.

5.2 Single Guest Performance

Tables 2 and 3 show the transmit and receive performance of a single guest operating system over two physical network interfaces using Xen and CDNA. The first two rows of each table show the performance of the Xen I/O virtualization architecture using both the Intel and RiceNIC network interfaces. The third row of each table shows the performance of the CDNA I/O virtualization architecture.

The Intel network interface can only be used with Xen through the use of software virtualization. However, the RiceNIC can be used with both CDNA and software virtualization. To use the RiceNIC interface with software virtualization, a context was assigned to the driver domain and no contexts were assigned to the guest operating system. Therefore, all network traffic from the guest operating system is routed via the driver domain as it normally would be, through the use of software virtualization. Within the driver domain, all of the mechanisms within the CDNA NIC are used identically to the way they would be used by a guest operating system when configured to use concurrent direct network access. As the tables show, the Intel network interface performs similarly to the RiceNIC network interface. Therefore, the benefits achieved with CDNA are the result of the CDNA I/O virtualization architecture, not the result of differences in network interface performance.

Note that in Xen the interrupt rate for the guest is not necessarily the same as it is for the driver. This is because the back-end driver within the driver domain attempts to interrupt the guest operating system whenever it generates new work for the front-end driver. This can happen at a higher or lower rate than the actual interrupt rate generated by the network interface depending on a variety of factors, including the number of packets that traverse the Ethernet bridge each time the driver domain is scheduled by the hypervisor.

Table 2 shows that using all of the available processing resources, Xen's software virtualization is not able to transmit at line rate over two network interfaces with either the Intel hardware or the RiceNIC hardware. However, only 41% of the processor is used by the guest operating system. The remaining resources are consumed by Xen overheads—using the Intel hardware, approximately 20% in the hypervisor and 37% in the driver domain performing software multiplexing and other tasks.

As the table shows, CDNA is able to saturate two network interfaces, whereas traditional Xen networking cannot. Additionally, CDNA performs far more efficiently, with 51% processor idle time. The increase in idle time is primarily the result of two factors. First, nearly all of the time spent in the driver domain is eliminated. The remaining time spent in the driver domain is unrelated to networking tasks. Second, the time spent in the hypervisor is decreased. With Xen, the hypervisor spends the bulk of its time managing the interactions between the front-end and back-end virtual network interface drivers. CDNA eliminates these communication overheads with the driver domain, so the hypervisor instead spends the bulk of its time managing DMA memory protection.

Table 3 shows the receive performance of the same configurations. Receiving network traffic requires more processor resources, so Xen only achieves 1112 Mb/s with the Intel network interface, and slightly lower with the RiceNIC interface. Again, Xen overheads consume the bulk of the time, as the guest operating system only consumes about 32% of the processor resources when using the Intel hardware.

As the table shows, not only is CDNA able to saturate the two network interfaces, it does so with 41% idle time. Again, nearly all of the time spent in the driver domain is eliminated. As with the transmit case, the CDNA archi-

System	DMA Protection	Mb/s	Domain Execution Profile						Interrupts/s	
			Hyp	Driver Domain		Guest OS		Idle	Driver Domain	Guest OS
				OS	User	OS	User			
CDNA (Transmit)	Enabled	1867	10.2%	0.3%	0.2%	37.8%	0.7%	50.8%	0	13,659
CDNA (Transmit)	Disabled	1867	1.9%	0.2%	0.2%	37.0%	0.3%	60.4%	0	13,680
CDNA (Receive)	Enabled	1874	9.9%	0.3%	0.2%	48.0%	0.7%	40.9%	0	7,402
CDNA (Receive)	Disabled	1874	1.9%	0.2%	0.2%	47.2%	0.3%	50.2%	0	7,243

Table 4. CDNA 2-NIC transmit and receive performance with and without DMA memory protection.

tecture permits the hypervisor to spend its time performing DMA memory protection rather than managing higher-cost interdomain communications as is required using software virtualization.

In summary, the CDNA I/O virtualization architecture provides significant performance improvements over Xen for both transmit and receive. On the transmit side, CDNA requires half the processor resources to deliver about 200 Mb/s higher throughput. On the receive side, CDNA requires 60% of the processor resources to deliver about 750 Mb/s higher throughput.

5.3 Memory Protection

The software-based protection mechanisms in CDNA can potentially be replaced by a hardware IOMMU. For example, AMD has proposed an IOMMU architecture for virtualization that restricts the physical memory that can be accessed by each device [2]. AMD's proposed architecture provides memory protection as long as each device is only accessed by a single domain. For CDNA, such an IOMMU would have to be extended to work on a per-context basis, rather than a per-device basis. This would also require a mechanism to indicate a context for each DMA transfer. Since CDNA only distinguishes between guest operating systems and not traffic flows, there are a limited number of contexts, which may make a generic system-level context-aware IOMMU practical.

Table 4 shows the performance of the CDNA I/O virtualization architecture both with and without DMA memory protection. (The performance of CDNA with DMA memory protection enabled was replicated from Tables 2 and 3 for comparison purposes.) By disabling DMA memory protection, the performance of the modified CDNA system establishes an upper bound on achievable performance in a system with an appropriate IOMMU. However, there would be additional hypervisor overhead to manage the IOMMU that is not accounted for by this experiment. Since CDNA can already saturate two network interfaces for both transmit and receive traffic, the effect of removing DMA protection is to increase the idle time by about 9%. As the table shows, this increase in idle time is the direct result of reducing the number of hypercalls from the guests and the time spent in the hypervisor performing protection operations.

Even as systems begin to provide IOMMU support for techniques such as CDNA, older systems will continue to lack such features. In order to generalize the design of CDNA for systems with and without an appropriate IOMMU, wrapper functions could be used around the hypercalls within the guest device drivers. The hypervisor must notify the guest whether or not there is an IOMMU. When no IOMMU is present, the wrappers would simply call the hypervisor, as described here. When an IOMMU is present, the wrapper would instead create DMA descriptors without hypervisor intervention and only invoke the hypervisor to set up the IOMMU. Such wrappers already exist in modern operating systems to deal with such IOMMU issues.

5.4 Scalability

Figures 3 and 4 show the aggregate transmit and receive throughput, respectively, of Xen and CDNA with two network interfaces as the number of guest operating systems varies. The percentage of CPU idle time is also plotted above each data point. CDNA outperforms Xen for both transmit and receive both for a single guest, as previously shown in Tables 2 and 3, and as the number of guest operating systems is increased.

As the figures show, the performance of both CDNA and software virtualization degrades as the number of guests increases. For Xen, this results in declining bandwidth, but the marginal reduction in bandwidth decreases with each increase in the number of guests. For CDNA, while the bandwidth remains constant, the idle time decreases to zero. Despite the fact that there is no idle time for 8 or more guests, CDNA is still able to maintain constant bandwidth. This is consistent with the leveling of the bandwidth achieved by software virtualization. Therefore, it is likely that with more CDNA NICs, the throughput curve would have a similar shape to that of software virtualization, but with a much higher peak throughput when using 1–4 guests.

These results clearly show that not only does CDNA deliver better network performance for a single guest operating system within Xen, but it also maintains significantly higher bandwidth as the number of guest operating systems

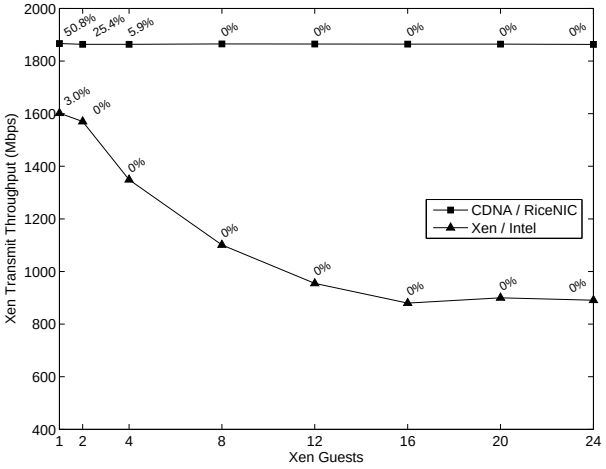

Figure 3. Transmit throughput for Xen and CDNA (with CDNA idle time).

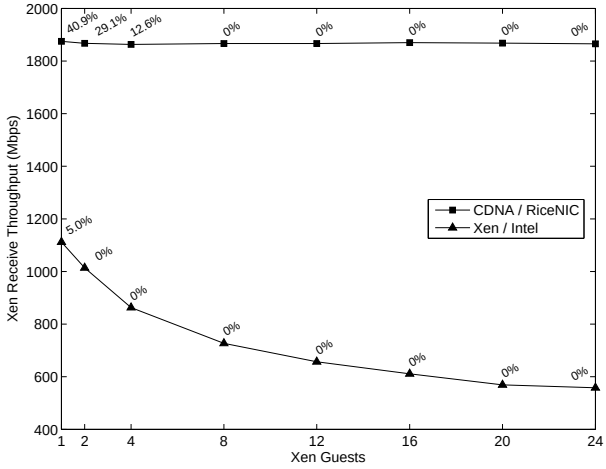

Figure 4. Receive throughput for Xen and CDNA (with CDNA idle time).

is increased. With 24 guest operating systems, CDNA's transmit bandwidth is a factor of 2.1 higher than Xen's and CDNA's receive bandwidth is a factor of 3.3 higher than Xen's.

6 Related Work

Previous studies have also found that network virtualization implemented entirely in software has high overhead. In 2001, Sugerman, *et al.* showed that in VMware, it could take up to six times the processor resources to saturate a 100 Mb/s network than in native Linux [19]. Similarly, in 2005, Menon, *et al.* showed that in Xen, network throughput degrades by up to a factor of 5 over native Linux for processor-bound networking workloads using Gigabit Ethernet links [13]. Section 2.3 shows that the I/O performance of Xen has improved, but there is still significant network virtualization overhead. Menon, *et al.* have also shown that it is possible to improve transmit performance with software-only mechanisms (mainly by leveraging TSO) [12]. However, there are no known software mechanisms to substantively improve receive performance.

Motivated by these performance issues, Raj and Schwan presented an Ethernet network interface targeted at VMMs that performs traffic multiplexing and interrupt delivery [16]. While their proposed architecture bears some similarity to CDNA, they did not present any mechanism for DMA memory protection.

As a result of the growing popularity of VMMs for commodity hardware, both AMD and Intel are introducing virtualization support to their microprocessors [2, 10]. This virtualization support should improve the performance of VMMs by providing mechanisms to simplify isolation among guest operating systems and to enable the hypervisor to occupy a new privilege level distinct from those normally used by the operating system. These improvements will reduce the duration and frequency of calls into the hypervisor, which should decrease the performance overhead of virtualization. However, none of the proposed innovations directly address the network performance issues discussed in this paper, such as the inherent overhead in multiplexing and copying/remapping data between the guest and driver domains. While the context switches between the two domains may be reduced in number or accelerated, the overhead of communication and multiplexing within the driver domain will remain. Therefore, concurrent direct network access will continue to be an important element of VMMs for networking workloads.

VMMs that utilize full virtualization, such as VMware ESX Server [7], support full binary compatibility with unmodified guest operating systems. This impacts the I/O virtualization architecture of such systems, as the guest operating system must be able to use its unmodified native device driver to access the virtual network interface. However, VMware also allows the use of paravirtualized network drivers (i.e., vmxnet), which enables the use of techniques such as CDNA.

The CDNA architecture is similar to that of user-level networking architectures that allow processes to bypass the operating system and access the NIC directly [5, 6, 8, 14, 15, 18, 20, 21]. Like CDNA, these architectures require DMA memory protection, an interrupt delivery mechanism, and network traffic multiplexing. Both user-level networking architectures and CDNA handle traffic multiplexing on the network interface. The only difference is that user-level NICs handle flows on a per-application basis, whereas

CDNA deals with flows on a per-OS basis. However, as the networking software in the operating system is quite different than that for user-level networking, CDNA relies on different mechanisms to implement DMA memory protection and interrupt delivery.

To provide DMA memory protection, user-level networking architectures rely on memory registration with both the operating system and the network interface hardware. The NIC will only perform DMA transfers to or from an application's buffers that have been registered with the NIC by the operating system. Because registration is a costly operation that involves communication with the NIC, applications typically register buffers during initialization, use them over the life of the application, and then deregister them during termination. However, this model of registration is impractical for modern operating systems that support zero-copy I/O. With zero-copy I/O, any part of physical memory may be used as a network buffer at any time. CDNA provides DMA memory protection without actively registering buffers on the NIC. Instead, CDNA relies on the hypervisor to enqueue validated buffers to the NIC by augmenting the hypervisor's existing memory-ownership functionality. This avoids costly runtime registration I/O and permits safe DMA operations to and from arbitrary physical addresses.

Because user-level networking applications typically employ polling at runtime rather than interrupts to determine when I/O operations have completed, interrupt delivery is relatively unimportant to the performance of such applications and may be implemented through a series of OS and application library layers. In contrast, interrupt delivery is an integral part of networking within the operating system. The interrupt delivery mechanism within CDNA efficiently delivers virtual interrupts to the appropriate guest operating systems.

Liu, *et al.* showed that user-level network interfaces can be used with VMMs to provide user-level access to the network from application processes running on a guest operating system within a virtual machine [11]. Their implementation replicates the existing memory registration and interrupt delivery interfaces of user-level NICs in the privileged driver domain, which forces such operations through that domain and further increases their costs. Conversely, CDNA simplifies these operations, enabling them to be efficiently implemented within the hypervisor.

7 Conclusion

Xen's software-based I/O virtualization architecture leads to significant network performance overheads. While this architecture supports a variety of hardware, the hypervisor and driver domain consume as much as 70% of the execution time during network transfers. A network interface that supports the CDNA I/O virtualization architecture eliminates much of this overhead, leading to dramatically improved single-guest performance and better scalability. With a single guest operating system using two Gigabit network interfaces, Xen consumes all available processing resources but falls well short of achieving the interfaces' line rate, sustaining 1602 Mb/s for transmit traffic and 1112 Mb/s for receive traffic. In contrast, CDNA saturates two interfaces for both transmit and receive traffic with 50.8% and 40.9% processor idle time, respectively. Furthermore, CDNA also maintains higher bandwidth as the number of guest operating systems increases. With 24 guest operating systems, CDNA improves aggregate transmit performance by a factor of 2.1 and aggregate receive performance by a factor of 3.3.

Concurrent direct network access is not specific to the Xen VMM. Any VMM that supports paravirtualized device drivers could utilize CDNA. Even VMware, a full virtualization environment, allows the use of paravirtualized device drivers. To support CDNA, a VMM would only need to add mechanisms to deliver interrupts as directed by the network interface and to perform DMA memory protection. The interrupt delivery mechanism of CDNA is suitable for a wide range of virtualized devices and would be relatively straightforward to implement in any VMM. However, the current implementation of CDNA's protection mechanism is specific to the Xen VMM and RiceNIC. In the future, the protection mechanism could be modified, as described in Section 3.4, to work with other devices and VMM environments.

This paper also shows that a commodity network interface needs only modest hardware modifications in order to support CDNA. As discussed in Section 4, three modifications would be required to enable a commodity NIC to support CDNA. First, the NIC must provide multiple contexts that can be accessed by programmed I/O, requiring 128 KB of memory in order to support 32 contexts. Second, the NIC must support several mailboxes within each context. Finally, the NIC must provide 12 MB of memory for use by the 32 contexts. A commodity network interface with these hardware modifications could support the CDNA I/O virtualization architecture with appropriate firmware modifications to service the multiple contexts, multiplex network traffic, and deliver interrupt bit vectors to the hypervisor.

In summary, the CDNA I/O virtualization architecture dramatically outperforms software-based I/O virtualization. Moreover, CDNA is compatible with modern virtual machine monitors for commodity hardware. Finally, commodity network interfaces only require minor modifications in order to support CDNA. Therefore, the CDNA concept is a cost-effective solution for I/O virtualization.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006.

[2] Advanced Micro Devices. *Secure Virtual Machine Architecture Reference Manual*, May 2005. Revision 3.01.

[3] Avnet Design Services. *Xilinx Virtex-II Pro Development Kit: User's Guide*, Nov. 2003. ADS-003704.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[5] P. Buonadonna and D. Culler. Queue pair IP: a hybrid architecture for system area networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, May 2002.

[6] Compaq Corporation, Intel Corporation, and Microsoft Corporation. Virtual interface architecture specification, version 1.0.

[7] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent #6,397,242*, Oct. 1998.

[8] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2), 1998.

[9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the On Demand IT InfraStructure (OASIS)*, Oct. 2004.

[10] Intel. *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*, Apr. 2005. Revision 2.0.

[11] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.

[12] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.

[13] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the ACM/USENIX Conference on Virtual Execution Environments*, June 2005.

[14] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The QUADRICS network: High-performance clustering technology. *IEEE MICRO*, Jan. 2002.

[15] I. Pratt and K. Fraser. Arsenic: a user-accessible Gigabit Ethernet interface. In *IEEE INFOCOM 2001*, pages 67–76, Apr. 2001.

[16] H. Raj and K. Schwan. Implementing a scalable self-virtualizing network interface on a multicore platform. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, Oct. 2005.

[17] J. Shafer and S. Rixner. *A Reconfigurable and Programmable Gigabit Ethernet Network Interface Card*. Rice University, Department of Electrical and Computer Engineering, Dec. 2006. Technical Report TREE0611.

[18] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *Proceedings of the Conference on Supercomputing (SC2001)*, Nov. 2001.

[19] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.

[20] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.

[21] T. von Eicken and W. Vogels. Evolution of the virtual interface architecture. *Computer*, 31(11), 1998.

[22] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.