Achieving 10 Gb/s using Safe and Transparent Network Interface Virtualization

Kaushik Kumar Ram[§] [*] Jose Renato Santos[‡] Yoshio Turner[‡] Alan L. Cox[§] Scott Rixner[§]

[‡]HP Labs [§]Rice University

{kaushik@rice.edu, joserenato.santos@hp.com, yoshio.turner@hp.com, alc@rice.edu, rixner@rice.edu}

Abstract

This paper presents mechanisms and optimizations to reduce the overhead of network interface virtualization when using the driver domain I/O virtualization model. The driver domain model provides benefits such as support for legacy device drivers and fault isolation. However, the processing overheads incurred in the driver domain to achieve these benefits limit overall I/O performance. This paper demonstrates the effectiveness of two approaches to reduce driver domain overheads. First, Xen is modified to support multi-queue network interfaces to eliminate the software overheads of packet demultiplexing and copying. Second, a grant reuse mechanism is developed to reduce memory protection overheads. These mechanisms shift the bottleneck from the driver domain to the guest domains, improving scalability and enabling significantly higher data rates. This paper also presents and evaluates a series of optimizations that substantially reduce the I/O virtualization overheads in the guest domain. In combination, these mechanisms and optimizations increase the maximum throughput achieved by guest domains from 2.9 Gb/s to full 10 Gigabit Ethernet link rates.

Categories and Subject Descriptors D.4.8 [*Operating Systems*]: Performance—Measurements

General Terms Design, Measurement, Performance

Keywords virtual machine, virtualization, performance analysis, I/O, networking, device drivers.

1. Introduction

Several virtual machine monitors—including Xen, L4, and Microsoft Hyper-V—use the *driver domain* model to virtualize I/O devices [8, 11, 16]. The driver domain is a virtual machine that runs a largely unmodified operating system. Consequently, it is able to use all of the device drivers that are available for that operating system. This greatly simplifies the complexity of providing support for a wide variety of devices in a virtualized environment. In addition, the driver domain model provides a safe execution environment for physical device drivers, enabling improved fault isolation over alternative models that locate device drivers in the hypervisor. However, while the driver domain provides several benefits, it also incurs significant performance overheads [14, 15]. For example, with a 10 Gigabit Ethernet (10 GbE) network interface, a guest domain running Linux on Xen can only achieve 2.9 Gb/s of throughput, whereas native Linux can achieve above 9.3 Gb/s (effectively line rate) on the same machine.

Direct I/O access has been proposed by many in order to eliminate the overheads of software-based I/O virtualization and close the gap with native I/O performance [12, 17, 18, 19, 20, 24]. Direct I/O access allows virtual machines to directly communicate with specially-designed I/O devices. These devices perform the appropriate packet multiplexing/demultiplexing among virtual machines. This allows virtual machines to achieve near native I/O performance. However, direct I/O solutions sacrifice fault isolation and device transparency. In particular, direct I/O requires device-specific code in the guest domain which has several negative consequences. It increases guest image complexity, reduces guest portability, and complicates live guest migration between systems with different devices. Therefore, I/O virtualization solutions that preserve the benefits of the driver domain model while minimizing its performance overheads are still needed.

The leading sources of overhead under the driver domain model in Xen are packet copying, packet demultiplexing and memory protection for I/O operations [21]. In Xen, received packets must be copied between the driver domain and the guest domain. The network interface card (NIC) places incoming packets into driver domain buffers. The driver domain must then demultiplex the received packet, determining to which guest it is destined. It then copies the packet into a buffer owned by that guest. In order for that packet copy to take place, the driver domain must have access to memory that belongs to the target guest. In Xen, guests must *grant* the driver domain access to such a buffer. This grant mechanism maintains protection among virtual machines, but is expensive, as the hypervisor must validate ownership of the memory buffer involved in the grant.

Previous work has argued that these copying, demultiplexing and memory protection overheads can be eliminated, in part through the use of a new generation of commodity NICs that support multiple transmit and receive queues for packets [21]. This paper fully realizes that vision, contributing a complete design and implementation of multi-queue NIC support for the driver domain model in Xen. This multi-queue NIC support eliminates packet copying between the driver domain and guest domains by allocating each guest domain its own queue on the multi-queue NIC. The driver domain then posts to each queue buffers that are owned by the associated guest domain. This allows the NIC to demultiplex packets and to transfer them directly to the guest domain's memory.

[*] This work was performed while Kaushik Kumar Ram was an intern at HP Labs.

VEE'09, March 11–13, 2009, Washington, DC, USA.
Copyright © 2009 ACM 978-1-60558-375-4/09/03…$5.00

VMware has also implemented support for multi-queue NICs in VMware ESX server [3]. As with multi-queue support in Xen, this allows packet multiplexing and demultiplexing to be performed on the NIC and eliminates a copy to the guest domain. However, ESX server hosts device drivers directly in the hypervisor, so the driver has access to all of the guest's memory. In contrast, a driver in the driver domain must be explicitly granted access to memory buffers in the guest domains.

To address this grant overhead in the driver domain model, this paper also introduces a novel grant reuse mechanism based on a software I/O address translation table. This mechanism takes advantage of the temporal locality in the use of I/O buffers to nearly eliminate the overhead of Xen's grant mechanism.

The combination of multi-queue NIC support and a grant reuse mechanism maintains the advantages of the driver domain model while significantly reducing its cost. The efficient use of a multi-queue NIC to eliminate packet demultiplexing and copying overheads leads to a 69% reduction in CPU cycles in the driver domain. The use of a software I/O translation table with grant reuse leads to a 53% reduction in the remaining CPU cycles in the driver domain. These optimizations shift the networking bottleneck from the driver domain to the guest domain. Moving the bottleneck to the guest enables the system to take advantage of multiple cores available in modern systems more effectively. Multiple guests can thereby achieve significantly higher aggregate data rates.

Finally, this paper describes several optimizations that substantially reduce the I/O virtualization overheads in the guest domain. These optimizations lead to a 30% reduction in CPU cycles in the guest domain. With all optimizations, the throughput of a single CPU guest is increased from 2.9 Gb/s to 8.2 Gb/s. When running two or more guests, the total throughput reaches the full 10 GbE line rate.

The rest of the paper is organized as follows. Section 2 reviews necessary background information about Xen's I/O architecture. Section 3 describes multi-queue NICs and how they can be used to eliminate packet copying and demultiplexing overheads. Section 4 presents the design of our software I/O translation table and grant reuse mechanisms. Section 5 presents experimental results that demonstrate reduced packet processing costs in the driver domain as a consequence of the proposed mechanisms. Section 6 analyzes the costs in the guest domain, and based on this analysis proposes and evaluates a series of additional optimizations to improve guest domain efficiency. Section 7 analyzes the impact on the achievable throughput of all of the optimizations proposed in the paper. Finally, Section 8 discusses related work, and Section 9 summarizes our conclusions.

2. Xen Networking Overview

In Xen, *driver domains* are privileged domains that have direct access to hardware devices and perform I/O operations on behalf of unprivileged guest domains [8]. A driver domain runs a Linux kernel with standard device drivers. Each hardware device can be assigned to a single driver domain, and multiple driver domains can control different hardware devices.

The driver domain model has two main advantages when compared to an alternative I/O model where device drivers are hosted in the hypervisor. First, they allow the use of legacy device drivers available for standard OSes such as Linux, minimizing the cost to develop and mantain new device drivers for the hypervisor. In addition, driver domains provide a safe execution environment isolating device drivers from the hypervisor and guest domains. This ensures that most device driver bugs are contained in the driver domain and cannot corrupt or crash other domains or the hypervisor. Although driver domain crashes can still affect guest domains due to unavailable I/O service, this is a more tolerable failure mode and usually only lasts a short period of time since I/O service can be rapidly restored by simply rebooting a faulty driver domain.

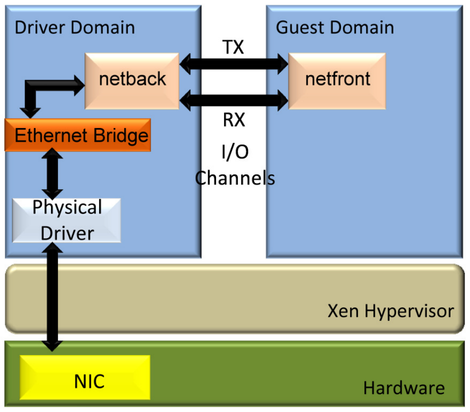

Figure 1. Xen's driver domain architecture

Figure 1 illustrates the operation of the driver domain to enable guest domains to send and receive network traffic over a NIC. The driver domain includes a standard Linux physical device driver for the NIC, an Ethernet bridge, and a *back-end driver* (netback) that interacts with a guest *front-end driver* (netfront). A different netback interface is created for each guest operating system that the driver domain supports.

When network packets are received by the NIC, the NIC will raise an interrupt. While the NIC's device driver in the driver domain may directly access the NIC, all interrupts in the system must first go through the hypervisor. The hypervisor receives the interrupt, determines that the NIC is owned by the driver domain, and then generates a virtual interrupt for the NIC's device driver in the driver domain. The device driver receives the virtual interrupt, processes the packets received by the NIC and sends them to the Ethernet bridge. The Ethernet bridge demultiplexes the packet based on its Ethernet address and delivers it to the appropriate netback interface. The netback driver then sends the packet to the netfront driver in the guest domain over an I/O channel. The netfront driver then delivers the packet to the guest operating system as if it had come directly from the NIC. The process is basically reversed to send packets from the guest domains.

The I/O channels allow communication between the front-end and back-end drivers using an event notification mechanism and a ring of requests and responses in memory that is shared between the domains. Netfront posts I/O requests pointing to I/O buffers in guest memory. Netback uses these requests either to send a transmitted packet to the NIC (or to another guest) or to copy a received packet into a guest I/O buffer. It finally sends a response on the corresponding I/O channel when the I/O operation is completed. The event notification mechanism enables netfront and netback to trigger a virtual interrupt in the other domain to indicate new requests or responses have been posted.

In order to enable driver domains to access guest I/O buffers in a safe manner without compromising the isolation properties of the driver domain model, Xen provides a page sharing mechanism known as the *grant mechanism*. The grant mechanism allows a guest to control which memory pages are allowed to be accessed by the driver domain. In addition, the grant mechanism allows the driver domain to validate that the I/O buffer belongs to the guest and that the page ownership does not change while the I/O is in progress (page pinning). Thus the grant mechanism serves two purposes. It ensures that the I/O operation does not access memory not owned by the guest and limits the set of guest pages that the driver domain can access. Currently, guest domains limit the pages available to the driver domain to only those containing buffers currently being used for I/O. This limits the amount of guest memory that is exposed to the driver domain to the minimum necessary, preventing bugs in device drivers from corrupting guest memory containing either code or data structures.

The guest uses a grant table shared with the hypervisor to indicate to which pages the driver domain has been granted access. Before sending a request on the I/O channel the front-end driver finds a free entry in the grant table and fills it with 1) the page address of the I/O buffer, 2) the driver domain id, and 3) the access permission (*read-only* or *read-write*) depending on whether the buffer is used for transmission or reception. The guest writes the grant reference (i.e. the index in the grant table) associated with the I/O buffer in the request posted on the I/O channel. Using this grant reference the back-end driver issues a *hypercall* that requests the hypervisor to map the guest page in its virtual address space. The hypervisor reads the guest grant table to validate the grant and also checks if the specified page belongs to the guest. It then pins the page and finally maps it in the driver domain address space. The driver domain can then access the guest buffer to perform the I/O operation. When the I/O is completed the page is unmapped using another hypercall before sending a response back to the guest. When netfront receives the response the grant is revoked by removing it from the grant table. Consequently the page can no longer be accessed by the driver domain. The guest operating system can then safely reallocate the memory to a different part of the kernel for non-I/O purposes.

For the receive path Xen provides a *grant copy* operation which maps the page, copies the packet and unmaps the page in a single hypercall reducing the cost of the grant mechanism.

In Xen, there are three major sources of overhead that limit network performance [21]. First, there are substantial packet demultiplexing overheads. Each received packet must first traverse the Ethernet bridge to arrive at the appropriate back-end driver. Second, there is an additional data copy required to copy a received packet from driver domain to guest memory. And third, there are substantial overheads associated with the grant mechanism. The repeated issue and revocation of page grants to the driver domain from the guest domains leads to prohibitive overheads. In addition to the hypercall overhead, pinning and mapping the page are also expensive operations associated with each grant.

3. Multi-queue NICs

Packet demultiplexing can be offloaded from the driver domain to the NIC. This can dramatically reduce the per-packet overhead in the driver domain.

Typically, modern NICs include a queue of data buffers to be transmitted and a queue of available buffers to use for received data. A NIC's device driver communicates with the NIC almost exclusively through these queues. NICs with multiple sets of these queues ("multi-queue NICs") have emerged to improve networking performance in multicore systems. These NICs can demultiplex incoming traffic into the multiple queues based on a hash value of packet headers fields. These devices support Microsoft's Receive Side Scaling architecture and Linux's Scalable I/O architecture. The basic idea is to allow each core exclusive access to one of the sets of queues on the NIC. This will allow each core to run the NIC's device driver concurrently without the need for additional synchronization. The use of multi-queue NICs in this manner increases the achievable parallelism within the network stack of the operating system, enabling more effective use of multiple cores for networking.

More recently multi-queue NICs have been extended to allow demultiplexing into different queues using the packet destination MAC address or VLAN tags (e.g. Intel's 82598 10 GbE controller [6]). This capability enables multi-queue NICs to be used for virtualization where each network interface queue can be dedicated to a specific guest. The NIC is able to identify the target guest domain for all incoming network traffic by associating a unique Ethernet address with each receive queue corresponding to the appropriate guest domain. The NIC can then demultiplex incoming network traffic based on the destination Ethernet address in the packet to place the packet in the appropriate receive queue, eliminating the need for processing these packets in the software bridge. To multiplex transmit network traffic, the NIC simply services all of the transmit queues fairly and interleaves the network traffic for each guest domain. While these NICs are currently available in the market, the drivers to support them are still being developed.

One benefit of using multi-queue NICs in this fashion is the elimination of the traffic multiplexing overheads within the driver domain. Another benefit is the elimination of copying between the driver domain and the guest domain. As each queue handles network traffic for a single guest, it is possible for the driver domain to direct the NIC to transfer data directly to/from memory owned by that guest domain. The guest domain need only grant the driver domain the right to use that physical memory. By appropriately posting buffers into receive queues, the driver domain can then direct the NIC to transfer packets from each queue directly to the corresponding guest domain's memory.

Since the number of RX queues on the multi-queue NIC is limited, only a bounded number of guest domains can take advantage of the dedicated RX queues. Additional guest domains fall back on a shared RX queue which is created by default when the multi-queue NIC is brought up in the driver domain.

4. Grant Reuse

As discussed in Section 2, guest domains in Xen grant the driver domain access to network buffers for each network I/O operation. Once the I/O operation completes, the grant is revoked. This approach provides memory protection and allows the guest to repurpose network buffers to other parts of the kernel after network I/O operations have completed. However, the need to map and unmap guest domain buffers into the driver domain address space on each I/O operation has high processing overhead that limits I/O performance.

When using a multi-queue NIC, packets are transferred directly into guest memory by the NIC itself. Therefore, in this case the driver domain does not have to access guest memory and does not need to map guest buffers into its address space. The driver domain still must verify that the buffers actually belong to the guest and that they have been pinned by the hypervisor. This is necessary to ensure that the backend driver only performs I/O operations using memory that is owned by the guest and that the page does not change ownership until the operation is complete. Section 4.1 presents a new *I/O translation table* mechanism that satisfies these requirements for memory protection and avoids mapping and unmapping guest buffers in the driver domain.

To avoid the high processing cost in the guest domain of issuing and revoking a grant for each I/O operation, Section 4.2 presents a new mechanism for *reusing grants*. This mechanism operates in conjunction with the I/O translation table to provide a pool of reusable I/O buffers in the guest domain. Memory protection overhead is greatly reduced by allowing grants and I/O translation table entries for buffers in the pool to be kept active across multiple I/O operations.

4.1 I/O Translation Table

We propose an *I/O translation table* mechanism that allows the driver domain to ensure that I/O pages belong to the corresponding guest and have been pinned by the hypervisor without having

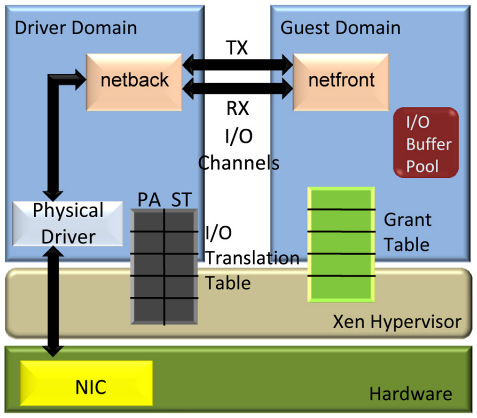

Figure 2. Grant Reuse Architecture

to map them. An I/O translation table is shared with the hypervisor and can be directly accessed by the driver domain to allow efficient validation of packet buffers. It is indexed by the same grant references used to index the grant table in the guest domain. When the guest domain grants the driver domain access to a network buffer, in addition to updating its grant table, the guest issues a grant hypercall. The hypervisor then updates the associated I/O translation table entry with the corresponding page address after validating and pinning the page. The guest domain then passes a grant reference to the driver domain as usual. In the original Xen grant mechanism the driver domain would make a hypercall with the grant reference to map and pin the page. In contrast, with use of the I/O translation table the page pinning has already been performed from the guest domain and the page does not need to be mapped into the driver domain's address space. Therefore, the driver domain avoids the hypercall and instead simply consults the I/O translation table to determine the machine address of the buffer. This address can be given directly to the NIC for packet transfer. Since the only way the guest domain can update the I/O translation table is via the hypervisor, the driver domain can trust that all entries belong to the corresponding guest and are pinned.

There is one I/O translation table for every guest-driver domain pair in the system. As shown in Figure 2, the table actually consists of two subtables: a buffer address subtable and a status subtable. The buffer address subtable contains physical addresses (PA) and the status subtable indicates the status (ST) of the buffer (i.e., whether or not the address is currently being used by the NIC). The subtables are separated because the driver domain only has read access to the buffer address subtable, while it has write access to the status subtable. This prevents the driver domain from inadvertently corrupting the buffer address subtable.

When the driver domain initiates a network operation using a grant reference, it increments the appropriate entry in the status subtable. When the operation completes, it decrements the entry in the status subtable. The hypervisor consults the status subtable when a guest tries to revoke a grant. If the status subtable indicates the granted memory is currently being used by the NIC, the hypervisor will not allow the grant to be revoked. This replicates the behavior that was previously accomplished by pinning and unpinning the memory buffer in the driver domain via a hypercall.

With the I/O translation table in place, the driver domain can directly read the buffer addresses from the table and safely use them for I/O operations without incurring the overhead of grant hypercalls. On the other hand the guest domains have additional overheads due to grant hypercalls to pin and unpin pages. But this overhead can be minimized by reusing grants over multiple I/O operations.

4.2 Reusing Grants

Given the desire to protect guests' memory from accidental corruption by the driver domain, it is important to only grant access to memory used as network buffers. However, immediately revoking the grant to a network buffer only provides minimal additional protection. Instead, it would be far more efficient to keep the grant active as long as the memory remains a network buffer in the guest operating system. In this way, grants can be reused for future network transfers and the cost of pinning and validating the grant can be amortized over multiple I/O operations. Once the guest decides to repurpose the memory, the grant can then be revoked and removed from the I/O translation table.

A pool of dedicated network I/O buffers was created in order to facilitate the effective reuse of grants. The network buffers were allocated from the pool when needed for I/O operations and returned to it afterwards. The socket buffer allocator in the guest operating system was modified in order to accomplish this. The I/O buffer pool was implemented as a new slab cache which was created and managed using the existing slab allocator in the Linux kernel. The slab cache guarantees that the memory associated with the network buffers will not be used for any other purpose until the kernel explicitly shrinks the cache to free memory and the pages are removed from the slab cache.

The guest OS also had to be modified to keep track of the grants associated with a network buffer page and to issue hypercalls to populate the I/O translation table. When an I/O buffer is allocated from the pool, netfront checks to see if there is already a grant reference associated with that buffer. If there is, then the existing grant reference is reused, eliminating the need for a hypercall to pin the granted page. If there is not a grant reference associated with the buffer, then netfront creates a new grant and makes a hypercall to pin the page. The hypervisor then simply validates the buffer and inserts an entry in the I/O translation table with the machine address of the buffer. In either case, the driver domain need not make any hypercalls to validate the buffer.

When the guest removes a buffer from the I/O buffer pool, it makes a hypercall to unpin the page. The hypervisor then removes the entry from the I/O translation table. The guest then revokes the grant and can safely use the buffer elsewhere in the kernel as the driver domain will no longer have access to it. This typically happens when the slab cache is shrunk to reclaim memory. The hypervisor ensures that a page in use by the NIC is not unpinned, preventing the page to change ownership if the guest is not well behaved.

Note that in the common case, I/O buffers will frequently be reused. In this case, the grant overhead is negligible, since only a single hypercall is needed the first time the guest domain grants the driver domain access to the buffer.

While grant reuse can also be done without an I/O translation table [21], this approach suffers from significant overheads and delays when the guest domain needs to revoke the grant. Before the guest domain can remove a buffer from the I/O buffer pool, it must interact with the driver domain over the I/O channel to ensure that it unmaps the buffer. However this might be unacceptable in certain situations of severe memory pressure in the guest which cannot wait for a handshake with another domain that may not even be running at the moment. The I/O translation table allows the guest to revoke grants without any handshake with the driver domain, as long as the buffer is not in use by an I/O operation.

Although in the current implementation the driver domain cannot perform any packet filtering or monitoring operation since guest buffers are not mapped in its address space, this is not a fundamen-

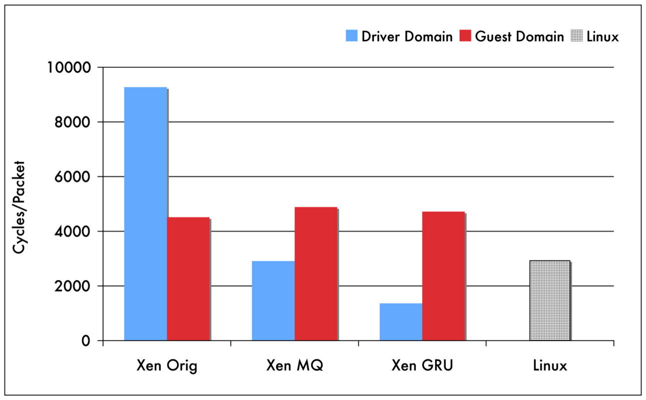

Figure 3. Packet Processing Cost: Multi-queue and Grant Reuse

Class	Description	associated source code
driver	native device driver and netfront in guest	drivers/net/ixgbe/* drivers/xen/netfront/*
netback	Xen backend driver	drivers/xen/netback/*
network	Linux network general	net/*
bridge	Linux network bridge	net/bridge/*
mm	Linux memory management	mm/*
mem*	memory copy/set in Linux	arch/x86_64/lib/memcpy.S arch/x86_64/lib/memset.S
Linux-grant	Linux grant table functions	drivers/xen/core/gnttab.c
User copy	Packet copy to user buffer	arch/x86_64/lib/copy_user.S
Linux other	other functions in Linux	numerous
Xen grant	Xen grant table functions	xen/common/grant_table.c
Xen	all other Xen functions not in grant table	numerous

Table 1. Classes grouping Linux and Xen functions profiles

tal limitation of the design. Modern multi-queue NICs also support *packet header splitting* which allows packet headers and payload to be placed in different buffers. Using this capability, the driver domain can use its own local buffers to receive packet headers and then perform packet filtering or monitoring operations, while still placing the payload directly into guest memory. Packet headers would need to be copied from driver domain to guest memory, but this should not add significant overhead since headers are small (e.g. 66 bytes for typical TCP packets).

5. Multi-queue and Grant Reuse Evaluation

This section presents experimental results evaluating the performance impact of multi-queue NICs and the grant reuse mechanism.

5.1 Experimental Setup

Our experiments are run on two HP Proliant DL380-G5 servers connected directly by a 10 gigabit multimode fiber cable. Each server has two 2.33 GHz Intel Xeon E5345 CPUs (two quad-core CPUs) with 20 GB of memory, and a 10 GbE Intel Oplin 82598EB NIC.

The netperf TCP stream microbenchmark (`www.netperf.org`) is used in the experiments to generate network traffic. The performance evaluation focuses on the network receive path which has significantly higher overhead than the transmit path in the current Xen implementation. The *sendfile* option to netperf is enabled to ensure that the transmit side of the connection is not the bottleneck.

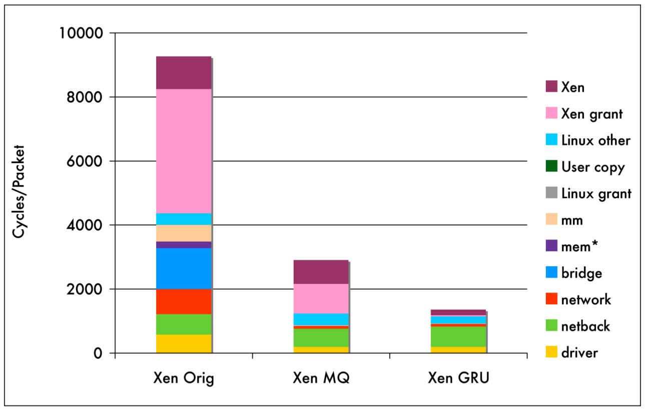

Figure 4. Packet Processing Cost in Driver Domain: Multi-queue and Grant Reuse

We use a recent Xen unstable[1] distribution along with paravirtualized Linux domains using linux-2.6.18-xen[2], and both use x86 64-bit mode. The system is configured with one guest domain, and one dedicated driver domain in addition to the privileged domain 0. The driver domain and the guest domain are each configured with 2 GB of memory and a single virtual CPU (to eliminate potential multi-CPU guest scheduling issues). Except when noted, the virtual CPUs of the guest and the driver domain are pinned to cores of different CPU sockets ensuring they do not share the L2 cache.

For native Linux experiments (hereafter referred to simply as "Linux") we use the same kernel version and similar config options as used in the paravirtualized Linux kernel. We also limit the kernel to use a single CPU and restrict the amount of memory to 2 GB to make the comparison between Xen and Linux as similar as possible.

We use OProfile [14] to determine the number of CPU cycles used in each Linux and Xen function when processing network packets. Given the large number of kernel and hypervisor functions we group them into a small number of classes based on their high level purpose as described in Table 1.

5.2 Multi-queue and Grant Reuse Results

Figure 3 compares the number of CPU cycles per packet consumed in the guest domain and in the driver domain for different configurations. The first bar shows the CPU cost for original Xen (*Xen Orig*) to process network packets on the receive side of a TCP connection; while the last bar shows the CPU cost for *Linux*. Original Xen has significantly higher cost than Linux, which is consistent with previous results. Moreover the driver domain has significantly higher cost than the guest itself. Most of this overhead is due to packet demultiplexing cost, extra data copies needed to move the packet from driver domain memory to guest memory, and the cost of the Xen grant mechanism [21].

The second bar (*Xen MQ*) in Figure 3 shows the improvement in processing cost that is achieved when guests are assigned a dedicated receive queue in a multi-queue NIC. As expected, multi-queue significantly improves performance. In particular, multi-queue reduces the driver domain cost by 69%. To understand the detailed impact on driver domain processing, Figure 4 shows the breakdown of cost across all the kernel and hypervisor functions that are executed on behalf of the driver domain.

[1] xen-unstable.hg changeset 17823:cc4e471bbc08, June 10, 2008

[2] linux-2.6.18.8-xen.hg changeset 572:5db911a71eac, June 10, 2008

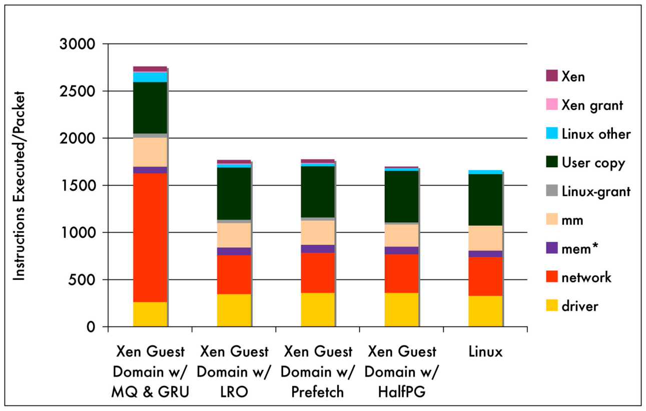

Figure 5. Guest domain cost breakdown in instructions executed/packet

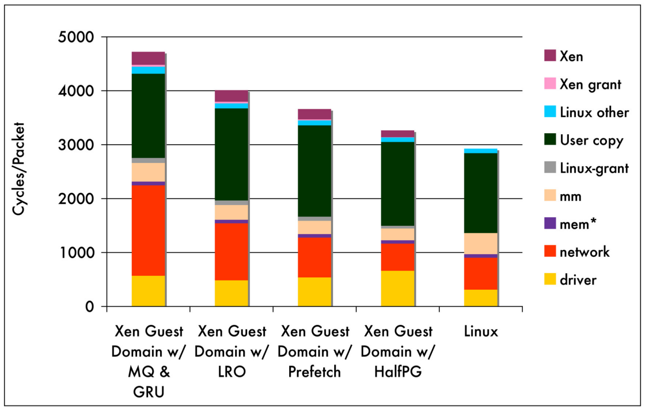

Figure 6. Guest domain cost breakdown in cycles/packet

Figure 4 shows that multi-queue significantly reduces the cost of the Xen grant table mechanism. This is because multi-queue avoids the cost of the data copy performed by the *grant copy* hypercall. This copy is responsible for most of the grant overhead in original Xen. Some grant overhead remains, however, since grants are still used to map and unmap the guest pages into the driver domain's address space.

In addition to avoiding a data copy, multi-queue avoids the need to route packets in software, because the multi-queue NIC performs routing in hardware. This avoids the overheads associated with the Linux bridge and with network processing as illustrated in Figure 4.

Finally, multi-queue removes the cost associated with memory management functions in the driver domain kernel as shown in Figure 4. This is a consequence of using a mechanism that we added to netback to recycle socket buffers when using multi-queue. Netback uses a regular Linux socket buffer data structure to represent a packet. When the native device driver for the multi-queue NIC requests an empty receive buffer to be posted on the device receive queue, netback gets a guest buffer previously posted on the I/O channel, attaches it as a fragment to the socket buffer and returns the socket buffer to the native device driver. When a packet is received in that buffer, the device driver returns the associated socket buffer to netback which then removes the fragment from the socket buffer and notifies the guest that a packet has been received. However, instead of freeing the local socket buffer, netback keeps it in a free list to be reused when a new guest buffer needs to be posted in the receive queue of the NIC. This socket buffer recycling avoids the Linux memory management costs associated with allocating and deallocating socket buffers.

The third bar (*GRU*) in Figure 3 shows the improvement in processing cost due to the I/O translation table and grant reuse mechanisms described in Section 4. Similar to multi-queue, grant reuse also reduces the overhead in the driver domain. In particular, grant reuse reduces the driver domain cost by 53% compared to the multi-queue results. Figure 4 confirms that grant reuse eliminates the cost associated with grant table code in Xen. In addition, most of the cost associated with other Xen functions is significantly reduced. These extra savings are due to overheads in other Xen functions which are indirectly executed when using grants. Most of these extra costs are due to hypercall code to enter and exit the hypervisor, and Xen memory management functions used to pin pages and to map and unmap granted pages into the driver domain address space.

In summary, support for multi-queue NICs, combined with grant reuse, reduces the per-packet processing cost in the driver domain by more than 80% compared to original Xen. By reducing the driver domain overhead the bottleneck shifts to the guest domain CPU which becomes the main throughput limiting factor. The processing cost in the guest domain is unaffected by multi-queue and grant reuse, and is 61% higher than the cost of Linux.

6. Guest Domain Optimizations

This section analyzes the loss of efficiency when running Linux as a guest in Xen compared to running Linux natively. This section also proposes additional optimizations to reduce packet processing cost in the guest domain. These optimizations are aimed at reducing the virtualization overheads in the guest and have all been implemented in netfront, the virtual driver in the guest domain.

6.1 LRO in Virtual Device Driver

In Figure 5, the first bar shows the distribution of the number of instructions executed per packet in the guest domain, when multi-queue support and the grant reuse mechanism are enabled. The instruction distribution for Linux is shown in the last bar in Figure 5. The guest domain has a much higher instruction count than Linux in the *network* component, caused by executing a larger number of instructions per packet for TCP/IP network stack processing.

This difference in the network stack processing is caused by the use of a software optimization called Large Receive Offload (LRO) [9, 13]. In Linux, the device driver (provided by Intel) uses LRO to aggregate arriving TCP/IP packets into a much smaller number of larger-sized packets (TCP segments), and then passes the large segments to the network stack. As a result, the network stack processes a group of packets as a single unit, at the same cost as for processing a single packet.

The second bar in Figure 5 shows that implementing LRO in netfront reduces the instruction execution count in the network stack close to that of Linux. Compared to the first bar, the second bar also shows that slightly more instructions are executed per packet in netfront to aggregate arriving packets into large TCP segments. Overall, the total instruction count is close to that of Linux (last bar).

The impact on processing cost (CPU cycles per packet) of using LRO in netfront is shown in the first two bars of Figure 6. The results show that adding LRO reduces guest domain processing cost by almost 15%, from 4721 to 4013 CPU cycles per packet. The reduction in CPU cycles in the network component offsets a smaller increase of CPU cycles in the netfront component. However, the total cost is still far higher than the Linux cost of 2927 CPU cycles per packet.

In comparing the guest domain with LRO to Linux, both have similar instruction count distributions (Figure 5), whereas the per-packet processing costs in CPU cycles are still far apart (Figure 6). Basically, the average number of cycles per instructions (CPI) in a Xen guest is higher than in native Linux. This higher instruction cost is caused by worse cache and TLB miss behavior. The two additional guest domain optimizations described in the next subsections address both of these factors.

6.2 Software Prefetching

After the NIC places packet data into guest buffers, the initial software access to the packet in netfront is certain to cause a cache miss (although future hardware optimizations may avoid this cache miss [10]). To reduce the cache miss penalty, we added instructions in netfront to prefetch packet data headers.

On each virtual interrupt, netfront executes a loop to process a batch of arriving packets (the batch size is determined by the interrupt rate of the NIC, which can be configured with the interrupt coalescing driver parameter). We added instructions causing each iteration of the loop to prefetch the header of the packet that will be processed in the next iteration. However, to issue this prefetch, the address of the packet data must be read from a field in the corresponding socket buffer. To minimize the cache miss penalty for accessing this field, in each iteration we also prefetch the socket buffer structure for the packet that will be processed two iterations later.

The third bar in Figure 6 shows the impact on processing cost of adding the software prefetch instructions to netfront. Also, the third bar in Figure 5 shows the corresponding impact on instruction execution count. In comparing these two figures, it is clear that the prefetch optimization reduces the guest domain processing cost by 8.7% (from 4013 to 3662 CPU cycles per packet), without adding significantly to the instruction execution count.

6.3 Half Page Buffers

For historical reasons, netfront in the guest kernel allocates full page buffers for each data packet. This was to support the original Xen I/O architecture in which page remapping and page flipping were used to transfer data between the driver domain and the guest domain. This approach has largely been supplanted by a data copy mechanism. In the case of multi-queue support, even the data copy is removed. There is no need to allocate full 4KB page buffers to accommodate standard sized Ethernet packets.

We modified netfront to allocate half page (2KB) buffers instead of full pages which are large enough to store standard sized Ethernet packet data. Compared to the previous optimizations (third bar in Figure 6), the half page allocation optimization (fourth bar in the figure) reduces the guest domain processing cost by 10%, from 3662 to 3266 CPU cycles per packet. There are two contributing causes for this cost reduction.

Figure 7 shows the TLB misses per packet before and after applying the half page allocation optimization. The result shows that using half page allocation reduces the number of TLB misses per packet. This is most likely a consequence of the fact that half page allocation reduces by a factor of two the contribution of packet data pages to the TLB working set. We conjecture that reducing the cost of handling TLB misses is responsible for most of the improvement in the overall cost when using half page allocation.

An additional effect that further reduces cost in the guest is that half page allocation turns out to improve the rate of grant reuse. Compared to the previous optimizations, the cost of grants (Xen-Grant and Linux-Grant categories) is reduced slightly. Since this cost is already small because of the effectiveness of grant reuse, the additional cost reduction is modest. Using half page allocation improves reuse since any page that is added to the grant table will be used for two I/O buffers instead of just one buffer in the case of full page allocation. The rate of grant reuse is around 95% using full page allocation and increases to 99.4% using half page allocation.

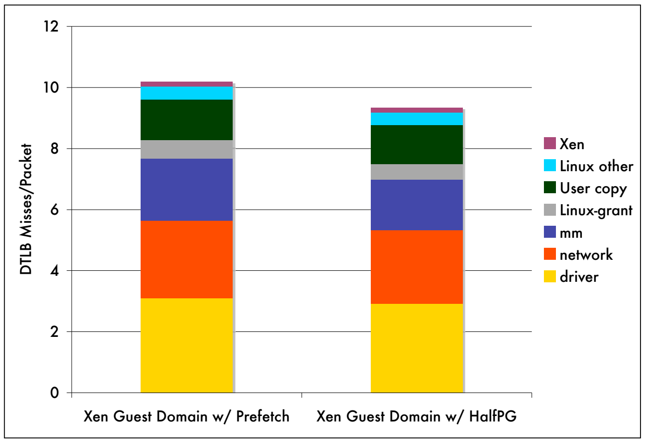

Figure 7. TLB Misses

The combined effect of the three guest domain optimizations described in this section is to reduce guest domain processing cost by 30%, from 4721 to 3266 CPU cycles per packet. This final cost is only 11% higher than the processing cost of Linux (2927 packets per cycle).

The remaining gap between Xen and native Linux is caused by two main factors: higher processing cost in the virtual device driver compared to the physical device driver, and additional CPU cycles spent in the hypervisor for event delivery between the guest and driver domains.

7. Impact of Optimizations

This section evaluates the impact of our optimizations on the I/O throughput that is achieved by guest domains. The results demonstrate that the large cost savings provided by our optimizations actually enable guest domains to receive packets at the full line rate of the 10 GbE NIC. This high throughput is achieved while preserving device-transparent virtualization and the safe execution environment of the driver domain model.

Figure 8 shows the data receive throughput achieved for various cases, and Figure 9 shows the corresponding CPU utilization in the driver and guest domains. The first bar, Xen Orig, shows that Xen without our optimizations achieves only 2.9 Gb/s throughput to a single guest. The bottleneck resource in this case is the 100% saturated driver domain CPU. In comparison, non-virtualized Linux, shown in the last bar, Linux, achieves 9.31 Gb/s using 100% of a CPU. This is close to the full line rate of 9.41 Gb/s (calculated by considering the bandwidth used by packet headers in full sized 1514-byte Ethernet frames).

In Figures 8 and 9, the second bar, Xen Opt (1 Guest), corresponds to Xen with multi-queue support, grant reuse, and the guest domain optimizations. With these optimizations, the achieved throughput increases 180% reaching 8.2 Gb/s for a single CPU guest. In this case the guest domain CPU is saturated and is the bottleneck resource that prevents achieving full line rate throughput.

By removing the driver domain bottleneck we achieve better scalability and higher data rates when running multiple guests which can take advantage of multiple cores available in modern CPUs. The third bar in Figure 8, Xen Opt (2 Guests), shows that full 10 GbE line rates can be achieved when running two guests on different physical processors. In this scenario the bottleneck resource is the link bandwidth.

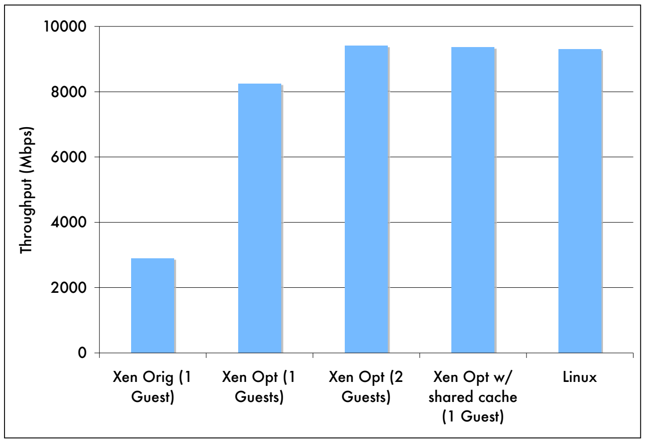

Figure 8. Impact of the optimizations on throughput

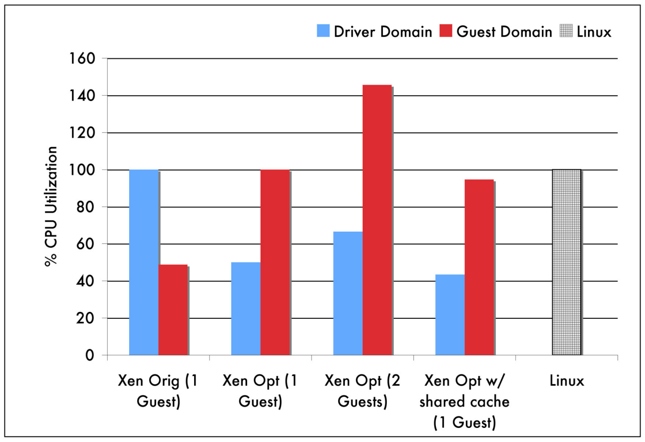

Figure 9. Impact of the optimizations on CPU utilization

The fourth bar in Figure 8 shows that a single CPU guest can achieve full 10GbE line rate if it executed on a CPU core that shares the L2 cache with the CPU core running the driver domain. Apparently, the cache footprints of the guest and driver domains fit in our 4 MB shared L2 cache, and this sharing provides some performance benefit when accessing shared data. These results indicate that a potentially fruitful topic for future research is to identify mechanisms and policies that can optimize cache sharing benefits for virtualized I/O. For example, in a large multi-core system it may be worthwhile to use driver domains with multiple virtual CPUs (VCPUs). Policies that assign VCPUs to physical CPU cores, and the assignment of device queues to driver domain VCPUs, can be coordinated with guest domain placement policies to maximize cache sharing between each guest domain and the driver domain VCPU that manages the device queue that is dedicated to the guest.

8. Related Work

Multiple virtual machine monitors have utilized driver domains, such as Xen, the L4 microkernel, and Microsoft's Hyper-V [8, 11, 16]. The driver domain model provides a safe execution environment for device drivers and protects guests and hypervisor from faults due to buggy device drivers. Chou *et al.* have shown that device driver bugs are the main cause of system crashes using an empirical study of Linux and OpenBSD Operating Systems [4]. To solve this problem, Swift *et al.* proposed to isolate device drivers from the rest of the kernel in standard Operating Systems [23]. The driver domain model provides the same capability by isolating device drivers in a separate virtual machine domain. However, as described earlier, previous I/O virtualization solutions based on driver domains incur significant CPU overheads and do not achieve high data rates.

In contrast to Xen, VMware's ESX server hosts device drivers directly in the hypervisor [7]. This sacrifices the fault isolation and range of driver support provided by the driver domain model, while enabling higher performance. Despite this difference, multi-queue NICs have also been used to reduce I/O virtualization overheads in ESX server [3]. The primary difference between the VMware solution and the Xen solution proposed here is the memory protection mechanisms. With device drivers directly in the hypervisor, all guest memory can implicitly be used for I/O operations. With the driver domain model of Xen, the grant mechanism is required, making the grant reuse mechanism critically important.

Direct I/O has been proposed as a very efficient I/O virtualization solution [12, 17, 18, 19, 20, 24]. With direct I/O, the device presents multiple logical interfaces which can be securely accessed directly by guest domains bypassing the virtualization layer, resulting in the best possible performance, with CPU cost close to native performance. However, direct I/O lacks the isolation property of driver domains since the device driver is executed inside the guest kernel. In addition, direct I/O breaks the device transparency provided to guest VMs when using software-based device virtualization. Device transparency has the benefit of avoiding the need to maintain device-specific code in guest VMs, thereby reducing the associated costs for maintaining and certifying guest images. In addition, device transparency simplifies live migration of guest VMs across physical machines that have different flavors of devices. Although techniques have been proposed to enable live migration [5, 22] with direct I/O [26], these techniques are not ideal since they are not transparent to the guest and require the guest to support complex mechanisms such as hot plug devices, device failover, etc. In addition, direct I/O increases the complexity of using virtual appliance models of software distribution which rely on the ability to execute on arbitrary hardware platforms. To use direct I/O, virtual appliances would have to include device drivers for a large variety of devices increasing their complexity, size, and maintainability. This paper shows that it is possible to efficiently virtualize network devices and achieve high data rates while avoiding the disadvantages of the direct I/O model and preserving the benefits of the driver domain model.

Menon *et al.* presented several transmit-side optimizations for Xen and advocated copying instead of page flipping on the receive path [15]. Menon and Zwaenepoel then implemented LRO within the driver domain of Xen [13]. In contrast, this paper describes a different set of optimizations on the receive path, and implements LRO within the guest domain. LRO is performed in the guest because packets are not mapped into the driver domain's address space.

Santos *et al.* evaluated the potential benefits of multi-queue and grant reuse using a 1 GbE NIC [21]. There, multi-queue NICs were only emulated on a traditional single queue NIC. Moreover, a real grant reuse mechanism was not implemented and evaluated, but instead its potential benefits was estimated by circumventing the memory protection mechanisms. In contrast, this paper demonstrates safe and efficient virtualization of a real multi-queue 10 GbE NIC. In addition, this paper proposes and evaluates the implementation of a novel grant reuse mechanism based on a software I/O translation table. This mechanism allows guests to revoke grants unilaterally without requiring a handshake with the driver domain.

Finally, our I/O translation table is similar to an IOMMU table [1, 2, 25]. While an IOMMU provides address translation for hardware devices, our I/O translation table mechanism provides address translations to software running in the device driver. However, the driver domain does not need to access the guest page in software but only uses the guest page address to program the hardware device to perform a DMA operation. If the different queues in the multi-queue devices could be configured to use different IOMMU tables, the I/O translation table could be replaced with an IOMMU table mapping grant references to guest pages. This is possible if each queue is assigned to a different PCI address (BDF), since IOMMU tables are selected by the device PCI address. Modern NICs that support the PCI-IOV standard for direct I/O [18] have exactly this capability and could be used with a driver domain in a multi-queue mode where all virtual functions (VF) could be assigned to the driver domain. This paper shows that this can be a better model for PCI-IOV devices than exposing the device virtual functions directly to the guest.

9. Conclusions

Currently, network I/O virtualization in Xen has significant overhead. With Xen, 4.7 times the number of processing cycles are consumed compared to native Linux to process each received packet. The high cost of virtualizing I/O in the driver domain limits network throughput when running Xen to only 2.9 Gb/s on a modern server which is able to achieve 9.3 Gb/s when running native Linux.

This paper presented mechanisms that significantly reduce the processing cost of the driver domain. First, the efficient use of a multi-queue NIC eliminates packet demultiplexing and copying overheads leading to a 69% reduction in processing cycles executed on behalf of the driver domain (including kernel and hypervisor code). Second, the reuse of page grants from the guest domain leads to a 53% reduction in the remaining processing cycles executed on behalf of the driver domain. These optimizations reduce the per-packet processing cost of the driver domain from 310% to 44% of the processing cost in native Linux.

In addition, this paper presented guest optimizations that increase the throughput of a single processor guest. The use of software Large Receive Offload (LRO), software prefetching, and half-page network buffers leads to a 30% reduction in processing cycles executed on behalf of the guest. The processing overhead in the guest domain itself is reduced from 52% to 7% of the native Linux cost. The remaining 7% gap between the guest domain and native Linux is caused by higher processing cost in the virtual device driver compared to the physical device driver. In addition, the cost of executing hypervisor code on behalf of the guest is reduced from 8% to 4% of the processing cost of native Linux.

In summary, the aggregate processing cost of the guest and the driver domain is reduced from 4.7 to just 1.55 times the processing cycles consumed by native Linux. This cost reduction increases the throughput of a single processor guest from 2.9 Gb/s to 8.2 Gb/s. By shifting the bottleneck from the driver domain to the guests, the optimizations presented in this paper enable the system to take advantage of the multiple cores available in modern systems more effectively and achieve full 10 GbE line rates when running two or more guests.

Acknowledgments

We would like to thank Mitch Williams, John Ronciak and Intel for providing us samples of the Intel Oplin 82598 NIC and for extending the ixgbe linux device driver to support multi-queue for Xen. We would also like to thank the National Science Foundation for their partial support of this project under Grant Nos. CCF-0546140 and CNS-0720878.

References

[1] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3), August 2006.

[2] Advanced Micro Devices, Inc. IOMMU architectural specification. `www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`, Feb 2007. PID 34434 Rev 1.20.

[3] Shefali Chinni and Radhakrishna Hiremane. Virtual machine device queues. Intel Corp. White Paper, 2007.

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles (SOSP)*, pages 73–88, New York, NY, USA, 2001.

[5] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[6] Intel Corp. Intel 82598 10 GbE ethernet controller open source datasheet, 2008. Revision 2.5.

[7] Scott Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMware US Patent 6397242, Oct 1998.

[8] Keir Fraser, Steve Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williams. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004.

[9] Leonid Grossman. Large Receive Offload implementation in Neterion 10GbE Ethernet driver. In *Ottawa Linux Symposium (OLS)*, 2005.

[10] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high-bandwidth network I/O. In *International Symposium on Computer Architecture (ISCA)*, 2005.

[11] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2004.

[12] Kieran Mansley, Greg Law, David Riddoch, Guido Barzini, Neil Turton, and Steven Pope. Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains. In *Euro-Par 2007 Workshops: Parallel Processing*, 2007.

[13] Aravind Menon and Willy Zwaenepoel. Optimizing TCP receive performance. In *USENIX Annual Technical Conference*, June 2008.

[14] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Conference on Virtual Execution Environments (VEE)*, June 2005.

[15] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *USENIX Annual Technical Conference*, June 2006.

[16] Microsoft. Hyper-V architecture. http://msdn.microsoft.com/en-us/library/cc768520.aspx.

[17] Neterion. Product brief : Neterion X3100 series. `http://www.neterion.com/products/pdfs/X3100ProductBrief.pdf`, 2008.

[18] PCI SIG. I/O virtualization. `www.pcisig.com/specifications/iov/`.

[19] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *International Symposium on High Performance Distributed Computing (HPDC)*, 2007.

[20] Scott Rixner. Network virtualization: Breaking the performance barrier. *ACM Queue*, January/February 2008.

[21] Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference*, June 2008.

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

[23] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.

[24] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2007.

[25] Paul Willmann, Alan L. Cox, and Scott Rixner. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference*, June 2008.

[26] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. Live migration with pass-through device for linux vm. In *Ottawa Linux Symposium (OLS)*, 2008.