Optimizing Network Virtualization in Xen

Aravind Menon
EPFL, Switzerland

Alan L. Cox
Rice university, Houston

Willy Zwaenepoel
EPFL, Switzerland

Abstract

In this paper, we propose and evaluate three techniques for optimizing network performance in the Xen virtualized environment. Our techniques retain the basic Xen architecture of locating device drivers in a privileged 'driver' domain with access to I/O devices, and providing network access to unprivileged 'guest' domains through virtualized network interfaces.

First, we redefine the virtual network interfaces of guest domains to incorporate high-level network offfload features available in most modern network cards. We demonstrate the performance benefits of high-level offload functionality in the virtual interface, even when such functionality is not supported in the underlying physical interface. Second, we optimize the implementation of the data transfer path between guest and driver domains. The optimization avoids expensive data remapping operations on the transmit path, and replaces page remapping by data copying on the receive path. Finally, we provide support for guest operating systems to effectively utilize advanced virtual memory features such as superpages and global page mappings.

The overall impact of these optimizations is an improvement in transmit performance of guest domains by a factor of 4.4. The receive performance of the driver domain is improved by 35% and reaches within 7% of native Linux performance. The receive performance in guest domains improves by 18%, but still trails the native Linux performance by 61%. We analyse the performance improvements in detail, and quantify the contribution of each optimization to the overall performance.

1 Introduction

In recent years, there has been a trend towards running network intensive applications, such as Internet servers, in virtual machine (VM) environments, where multiple VMs running on the same machine share the machine's network resources. In such an environment, the virtual machine monitor (VMM) virtualizes the machine's network I/O devices to allow multiple operating systems running in different VMs to access the network concurrently.

Despite the advances in virtualization technology [15, 4], the overhead of network I/O virtualization can still significantly affect the performance of network-intensive applications. For instance, Sugerman et al. [15] report that the CPU utilization required to saturate a 100 Mbps network under Linux 2.2.17 running on VMware Workstation 2.0 was 5 to 6 times higher compared to the utilization under native Linux 2.2.17. Even in the paravirtualized Xen 2.0 VMM [4], Menon et al. [10] report significantly lower network performance under a Linux 2.6.10 guest domain, compared to native Linux performance. They report performance degradation by a factor of 2 to 3x for receive workloads, and a factor of 5x degradation for transmit workloads. The latter study, unlike previous reported results on Xen [4, 5], reports network performance under configurations leading to full CPU saturation.

In this paper, we propose and evaluate a number of optimizations for improving the networking performance under the Xen 2.0 VMM. These optimizations address many of the performance limitations identified by Menon et al. [10]. Starting with version 2.0, the Xen VMM adopted a network I/O architecture that is similar to the hosted virtual machine model [5]. In this architecture, a physical network interface is owned by a special, privileged VM called a *driver domain* that executes the native Linux network interface driver. In contrast, an ordinary VM called a *guest domain* is given access to a virtualized network interface. The virtualized network interface has a front-end device in the guest domain and a back-end device in the corresponding driver domain. The front-end and back-end devices transfer network packets between their domains over an *I/O channel* that is provided by the Xen VMM. Within the driver domain, either Eth-

ernet bridging or IP routing is used to demultiplex incoming network packets and to multiplex outgoing network packets between the physical network interface and the guest domain through its corresponding back-end device.

Our optimizations fall into the following three categories:

1. We add three capabilities to the virtualized network interface: scatter/gather I/O, TCP/IP checksum offload, and TCP segmentation offload (TSO). Scatter/gather I/O and checksum offload improve performance in the guest domain. Scatter/gather I/O eliminates the need for data copying by the Linux implementation of `sendfile()`. TSO improves performance throughout the system. In addition to its well-known effects on TCP performance [11, 9] benefiting the guest domain, it improves performance in the Xen VMM and driver domain by reducing the number of network packets that they must handle.

2. We introduce a faster I/O channel for transferring network packets between the guest and driver domains. The optimizations include transmit mechanisms that avoid a data remap or copy in the common case, and a receive mechanism optimized for small data receives.

3. We present VMM support mechanisms for allowing the use of efficient virtual memory primitives in guest operating systems. These mechanisms allow guest OSes to make use of superpage and global page mappings on the Intel x86 architecture, which significantly reduce the number of TLB misses by guest domains.

Overall, our optimizations improve the transmit performance in guest domains by a factor 4.4. Receive side network performance is improved by 35% for driver domains, and by 18% for guest domains. We also present a detailed breakdown of the performance benefits resulting from each individual optimization. Our evaluation demonstrates the performance benefits of TSO support in the virtual network interface even in the absence of TSO support in the physical network interface. In other words, emulating TSO in software in the driver domain results in higher network performance than performing TCP segmentation in the guest domain.

The outline for the rest of the paper is as follows. In section 2, we describe the Xen network I/O architecture and a summary of its performance overheads as described in previous research. In section 3, we describe the design of the new virtual interface architecture. In section 4, we describe our optimizations to the I/O channel. Section 5 describes the new virtual memory optimization mechanisms added to Xen. Section 6 presents an evaluation of the different optimizations described. In section 7, we discuss related work, and we conclude in section 8.

2 Background

The network I/O virtualization architecture in Xen can be a significant source of overhead for networking performance in guest domains [10]. In this section, we describe the overheads associated with different aspects of the Xen network I/O architecture, and their overall impact on guest network performance. To understand these overheads better, we first describe the network virtualization architecture used in Xen.

The Xen VMM uses an I/O architecture which is similar to the hosted VMM architecture [5]. Privileged domains, called 'driver' domains, use their native device drivers to access I/O devices directly, and perform I/O operations on behalf of other unprivileged domains, called guest domains. Guest domains use virtual I/O devices controlled by paravirtualized drivers to request the driver domain for device access.

The network architecture used in Xen is shown in figure 1. Xen provides each guest domain with a number of virtual network interfaces, which is used by the guest domain for all its network communications. Corresponding to each virtual interface in a guest domain, a 'backend' interface is created in the driver domain, which acts as the proxy for that virtual interface in the driver domain. The virtual and backend interfaces are 'connected' to each other over an 'I/O channel'. The I/O channel implements a zero-copy data transfer mechanism for exchanging packets between the virtual interface and backend interfaces by remapping the physical page containing the packet into the target domain.

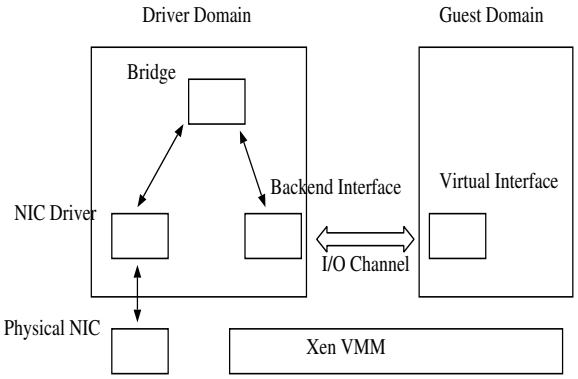

Figure 1: Xen Network I/O Architecture

All the backend interfaces in the driver domain (corresponding to the virtual interfaces) are connected to the physical NIC and to each other through a virtual network

bridge. [1] The combination of the I/O channel and network bridge thus establishes a communication path between the guest's virtual interfaces and the physical interface.

Table 1 compares the transmit and receive bandwidth achieved in a Linux guest domain using this virtual interface architecture, with the performance achieved with the benchmark running in the driver domain and in native Linux. These results are obtained using a netperf-like [1] benchmark which used the zero-copy `sendfile()` for transmit.

Configuration	Receive (Mb/s)	Transmit (Mb/s)
Linux	2508 (100%)	3760 (100%)
Xen driver	1738 (69.3%)	3760 (100%)
Xen guest	820 (32.7%)	750 (19.9%)

Table 1: Network performance under Xen

The driver domain configuration shows performance comparable to native Linux for the transmit case and a degradation of 30% for the receive case. However, this configuration uses native device drivers to directly access the network device, and thus the virtualization overhead is limited to some low-level functions such as interrupts.

In contrast, in the guest domain configuration, which uses virtualized network interfaces, the impact of network I/O virtualization is much more pronounced. The receive performance in guest domains suffers from a performance degradation of 67% relative to native Linux, and the transmit performance achieves only 20% of the throughput achievable under native Linux.

Menon et al. [10] report similar results. Moreover, they introduce Xenoprof, a variant of Oprofile [2] for Xen, and apply it to break down the performance of a similar benchmark. Their study made the following observations about the overheads associated with different aspects of the Xen network virtualization architecture.

Since each packet transmitted or received on a guest domain's virtual interface had to pass through the I/O channel and the network bridge, a significant fraction of the network processing time was spent in the Xen VMM and the driver domain respectively. For instance, 70% of the execution time for receiving a packet in the guest domain was spent in transferring the packet through the driver domain and the Xen VMM from the physical interface. Similarly, 60% of the processing time for a transmit operation was spent in transferring the packet from the guest's virtual interface to the physical interface. The breakdown of this processing overhead was roughly 40% Xen, 30% driver domain for receive traffic, and 30% Xen, 30% driver domain for transmit traffic.

In general, both the guest domains and the driver domain were seen to suffer from a significantly higher TLB miss rate compared to execution in native Linux. Additionally, guest domains were seen to suffer from much higher L2 cache misses compared to native Linux.

3 Virtual Interface Optimizations

Network I/O is supported in guest domains by providing each guest domain with a set of virtual network interfaces, which are multiplexed onto the physical interfaces using the mechanisms described in section 2. The virtual network interface provides the abstraction of a simple, low-level network interface to the guest domain, which uses paravirtualized drivers to perform I/O on this interface. The network I/O operations supported on the virtual interface consist of simple network transmit and receive operations, which are easily mappable onto corresponding operations on the physical NIC.

Choosing a simple, low-level interface for virtual network interfaces allows the virtualized interface to be easily supported across a large number of physical interfaces available, each with different network processing capabilities. However, this also prevents the virtual interface from taking advantage of different network offload capabilities of the physical NIC, such as checksum offload, scatter/gather DMA support, TCP segmentation offload (TSO).

3.1 Virtual Interface Architecture

We propose a new virtual interface architecture in which the virtual network interface always supports a fixed set of high level network offload features, irrespective of whether these features are supported in the physical network interfaces. The architecture makes use of offload features of the physical NIC itself if they match the offload requirements of the virtual network interface. If the required features are not supported by the physical interface, the new interface architecture provides support for these features in software.

Figure 2 shows the top-level design of the virtual interface architecture. The virtual interface in the guest domain supports a set of high-level offload features, which are reported to the guest OS by the front-end driver controlling the interface. In our implementation, the features supported by the virtual interface are checksum offloading, scatter/gather I/O and TCP segmentation offloading.

Supporting high-level features like scatter/gather I/O and TSO allows the guest domain to transmit network packets in sizes much bigger than the network MTU, and which can consist of multiple fragments. These large, fragmented packets are transferred from the virtual to the physical interface over a modified I/O channel and network bridge (The I/O channel is modified to allow transfer of packets consisting of multiple fragments, and

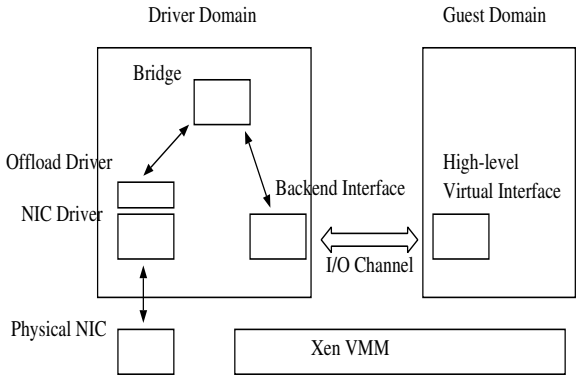

Figure 2: New I/O Architecture

the network bridge is modified to support forwarding of packets larger than the MTU).

The architecture introduces a new component, a 'software offload' driver, which sits above the network device driver in the driver domain, and intercepts all packets to be transmitted on the network interface. When the guest domain's packet arrives at the physical interface, the offload driver determines whether the offload requirements of the packet are compatible with the capabilities of the NIC, and takes different actions accordingly.

If the NIC supports the offload features required by the packet, the offload driver simply forwards the packet to the network device driver for transmission.

In the absence of support from the physical NIC, the offload driver performs the necessary offload actions in software. Thus, if the NIC does not support TSO, the offload driver segments the large packet into appropriate MTU sized packets before sending them for transmission. Similarly, it takes care of the other offload requirements, scatter/gather I/O and checksum offloading if these are not supported by the NIC.

3.2 Advantage of a high level interface

A high level virtual network interface can reduce the network processing overhead in the guest domain by allowing it to offload some network processing to the physical NIC. Support for scatter/gather I/O is especially useful for doing zero-copy network transmits, such as sendfile in Linux. Gather I/O allows the OS to construct network packets consisting of multiple fragments directly from the file system buffers, without having to first copy them out to a contiguous location in memory. Support for TSO allows the OS to do transmit processing on network packets of size much larger than the MTU, thus reducing the per-byte processing overhead by requiring fewer packets to transmit the same amount of data.

Apart from its benefits for the guest domain, a significant advantage of high level interfaces comes from its impact on improving the efficiency of the network virtualization architecture in Xen.

As described in section 2, roughly 60% of the execution time for a transmit operation from a guest domain is spent in the VMM and driver domain for multiplexing the packet from the virtual to the physical network interface. Most of this overhead is a per-packet overhead incurred in transferring the packet over the I/O channel and the network bridge, and in packet processing overheads at the backend and physical network interfaces.

Using a high level virtual interface improves the efficiency of the virtualization architecture by reducing the number of packets transfers required over the I/O channel and network bridge for transmitting the same amount of data (by using TSO), and thus reducing the per-byte virtualization overhead incurred by the driver domain and the VMM.

For instance, in the absence of support for TSO in the virtual interface, each 1500 (MTU) byte packet transmitted by the guest domain requires one page remap operation over the I/O channel and one forwarding operation over the network bridge. In contrast, if the virtual interface supports TSO, the OS uses much bigger sized packets comprising of multiple pages, and in this case, each remap operation over the I/O channel can potentially transfer 4096 bytes of data. Similarly, with larger packets, much fewer packet forwarding operations are required over the network bridge.

Supporting larger sized packets in the virtual network interface (using TSO) can thus significantly reduce the overheads incurred in network virtualization along the transmit path. It is for this reason that the software offload driver is situated in the driver domain at a point where the transmit packet would have already covered much of the multiplexing path, and is ready for transmission on the physical interface.

Thus, even in the case when the offload driver may have to perform network offloading for the packet in software, we expect the benefits of reduced virtualization overhead along the transmit path to show up in overall performance benefits. Our evaluation in section 6 shows that this is indeed the case.

4 I/O Channel Optimizations

Previous research [10] noted that 30-40% of execution time for a network transmit or receive operation was spent in the Xen VMM, which included the time for page remapping and ownership transfers over the I/O channel and switching overheads between the guest and driver domain.

The I/O channel implements a zero-copy page remapping mechanism for transferring packets between the

guest and driver domain. The physical page containing the packet is remapped into the address space of the target domain. In the case of a receive operation, the ownership of the physical page itself which contains the packet is transferred from the driver to the guest domain (Both these operations require each network packet to be allocated on a separate physical page).

A study of the implementation of the I/O channel in Xen reveals that three address remaps and two memory allocation/deallocation operations are required for each packet receive operation, and two address remaps are required for each packet transmit operation. On the receive path, for each network packet (physical page) transferred from the driver to the guest domain, the guest domain releases ownership of a page to the hypervisor, and the driver domain acquires a replacement page from the hypervisor, to keep the overall memory allocation constant. Further, each page acquire or release operation requires a change in the virtual address mapping. Thus overall, for each receive operation on a virtual interface, two page allocation/freeing operations and three address remapping operations are required. For the transmit path, ownership transfer operation is avoided as the packet can be directly mapped into the privileged driver domain. Each transmit operation thus incurs two address remapping operations.

We now describe alternate mechanisms for packet transfer on the transmit path and the receive path.

4.1 Transmit Path Optimization

We observe that for network transmit operations, the driver domain does not need to map in the entire packet from the guest domain, if the destination of the packet is any host other than the driver domain itself. In order to forward the packet over the network bridge, (from the guest domain's backend interface to its target interface), the driver domain only needs to examine the MAC header of the packet. Thus, if the packet header can be supplied to the driver domain separately, the rest of the network packet does not need to be mapped in.

The network packet needs to be mapped into the driver domain only when the destination of the packet is the driver domain itself, or when it is a broadcast packet.

We use this observation to avoid the page remapping operation over the I/O channel in the common transmit case. We augment the I/O channel with an out-of-band 'header' channel, which the guest domain uses to supply the header of the packet to be transmitted to the backend driver. The backend driver reads the header of the packet from this channel to determine if it needs to map in the entire packet (i.e., if the destination of the packet is the driver domain itself or the broadcast address). It then constructs a network packet from the packet header and the (possibly unmapped) packet fragments, and forwards this over the network bridge.

To ensure correct execution of the network driver (or the backend driver) for this packet, the backend ensures that the pseudo-physical to physical mappings of the packet fragments are set correctly (When a foreign page frame is mapped into the driver domain, the pseudo-physical to physical mappings for the page have to be updated). Since the network driver uses only the physical addresses of the packet fragments, and not their virtual address, it is safe to pass an unmapped page fragment to the driver.

The 'header' channel for transferring the packet header is implemented using a separate set of shared pages between the guest and driver domain, for each virtual interface of the guest domain. With this mechanism, the cost of two page remaps per transmit operation is replaced in the common case, by the cost of copying a small header.

We note that this mechanism requires the physical network interface to support gather DMA, since the transmitted packet consists of a packet header and unmapped fragments. If the NIC does not support gather DMA, the entire packet needs to be mapped in before it can be copied out to a contiguous memory location. This check is performed by the software offload driver (described in the previous section), which performs the remap operation if necessary.

4.2 Receive Path Optimization

The Xen I/O channel uses page remapping on the receive path to avoid the cost of an extra data copy. Previous research [13] has also shown that avoiding data copy in network operations significantly improves system performance.

However, we note that the I/O channel mechanism can incur significant overhead if the size of the packet transferred is very small, for example a few hundred bytes. In this case, it may not be worthwhile to avoid the data copy.

Further, data transfer by page transfer from the driver domain to the guest domain incurs some additional overheads. Firstly, each network packet has to be allocated on a separate page, so that it can be remapped. Additionally, the driver domain has to ensure that there is no potential leakage of information by the remapping of a page from the driver to a guest domain. The driver domain ensures this by zeroing out at initialization, all pages which can be potentially transferred out to guest domains. (The zeroing is done whenever new pages are added to the memory allocator in the driver domain used for allocating network socket buffers).

We investigate the alternate mechanism, in which packets are transferred to the guest domain by data copy.

As our evaluation shows, packet transfer by data copy instead of page remapping in fact results in a small improvement in the receiver performance in the guest domain. In addition, using copying instead of remapping allows us to use regular MTU sized buffers for network packets, which avoids the overheads incurred from the need to zero out the pages in the driver domain.

The data copy in the I/O channel is implemented using a set of shared pages between the guest and driver domains, which is established at setup time. A single set of pages is shared for all the virtual network interfaces in the guest domain (in order to restrict the copying overhead for a large working set size). Only one extra data copy is involved from the driver domain's network packets to the shared memory pool.

Sharing memory pages between the guest and the driver domain (for both the receive path, and for the header channel in the transmit path) does not introduce any new vulnerability for the driver domain. The guest domain cannot crash the driver domain by doing invalid writes to the shared memory pool since, on the receive path, the driver domain does not read from the shared pool, and on the transmit path, it would need to read the packet headers from the guest domain even in the original I/O channel implementation.

5 Virtual Memory Optimizations

It was noted in a previous study [10], that guest operating systems running on Xen (both driver and guest domains) incurred a significantly higher number of TLB misses for network workloads (more than an order of magnitude higher) relative to the TLB misses in native Linux execution. It was conjectured that this was due to the increase in working set size when running on Xen.

We show that it is the absence of support for certain virtual memory primitives, such as superpage mappings and global page table mappings, that leads to a marked increase in TLB miss rate for guest OSes running on Xen.

5.1 Virtual Memory features

Superpage and global page mappings are features introduced in the Intel x86 processor series starting with the Pentium and Pentium Pro processors respectively.

A superpage mapping allows the operating system to specify the virtual address translation for a large set of pages, instead of at the granularity of individual pages, as with regular paging. A superpage page table entry provides address translation from a set of contiguous virtual address pages to a set of contiguous physical address pages.

On the x86 platform, one superpage entry covers 1024 pages of physical memory, which greatly increases the virtual memory coverage in the TLB. Thus, this greatly reduces the capacity misses incurred in the TLB. Many operating systems use superpages to improve their overall TLB performance: Linux uses superpages to map the 'linear' (lowmem) part of the kernel address space; FreeBSD supports superpage mappings for both user-space and kernel space translations [12].

The support for global page table mappings in the processor allows certain page table entries to be marked 'global', which are then kept persistent in the TLB across TLB flushes (for example, on context switch). Linux uses global page mappings to map the kernel part of the address space of each process, since this part is common between all processes. By doing so, the kernel address mappings are not flushed from the TLB when the processor switches from one process to another.

5.2 Issues in supporting VM features

5.2.1 Superpage Mappings

A superpage mapping maps a contiguous virtual address range to a contiguous physical address range. Thus, in order to use a superpage mapping, the guest OS must be able to determine physical contiguity of its page frames within a superpage block. This is not be possible in a fully virtualized system like VMware ESX server [16], where the guest OS uses contiguous 'pseudo-physical' addresses, which are transparently mapped to discontiguous physical addresses.

A second issue with the use of superpages is the fact that all page frames within a superpage block must have identical memory protection permissions. This can be problematic in a VM environment because the VMM may want to set special protection bits for certain page frames. As an example, page frames containing page tables of the guest OS must be set read-only so that the guest OS cannot modify them without notifying the VMM. Similarly, the GDT and LDT pages on the x86 architecture must be set read-only.

These special permission pages interfere with the use of superpages. A superpage block containing such special pages must use regular granularity paging to support different permissions for different constituent pages.

5.2.2 Global Mappings

Xen does not allow guest OSes to use global mappings since it needs to fully flush the TLB when switching between domains. Xen itself uses global page mappings to map its address range.

5.3 Support for Virtual Memory primitives in Xen

5.3.1 Superpage Mappings

In the Xen VMM, supporting superpages for guest OSes is simplified because of the use of the paravirtualization approach. In the Xen approach, the guest OS is already aware of the physical layout of its memory pages. The Xen VMM provides the guest OS with a pseudo-physical to physical page translation table, which can be used by the guest OS to determine the physical contiguity of pages.

To allow the use of superpage mappings in the guest OS, we modify the Xen VMM and the guest OS to cooperate with each other over the allocation of physical memory and the use of superpage mappings.

The VMM is modified so that for each guest OS, it tries to give the OS an initial memory allocation such that page frames within a superpage block are also physically contiguous (Basically, the VMM tries to allocate memory to the guest OS in chunks of superpage size, i.e., 4 MB). Since this is not always possible, the VMM does not guarantee that page allocation for each superpage range is physically contiguous. The guest OS is modified so that it uses superpage mappings for a virtual address range only if it determines that the underlying set of physical pages is also contiguous.

As noted above in section 5.2.1, the use of pages with restricted permissions, such as pagetable (PT) pages, prevents the guest OS from using a superpage to cover the physical pages in that address range.

As new processes are created in the system, the OS frequently needs to allocate (read-only) pages for the process's page tables. Each such allocation of a read-only page potentially forces the OS to convert the superpage mapping covering that page to a two-level regular page mapping. With the proliferation of such read-only pages over time, the OS would end up using regular paging to address much of its address space.

The basic problem here is that the PT page frames for new processes are allocated randomly from all over memory, without any locality, thus breaking multiple superpage mappings. We solve this problem by using a special memory allocator in the guest OS for allocating page frames with restricted permissions. This allocator tries to group together all memory pages with restricted permissions into a contiguous range within a superpage.

When the guest OS allocates a PT frame from this allocator, the allocator reserves the entire superpage containing this PT frame for future use. It then marks the entire superpage as read-only, and reserves the pages of this superpage for read-only use. On subsequent requests for PT frames from the guest OS, the allocator returns pages from the reserved set of pages. PT page frames freed by the guest OS are returned to this reserved pool. Thus, this mechanism collects all pages with the same permission into a different superpage, and avoids the breakup of superpages into regular pages.

Certain read-only pages in the Linux guest OS, such as the boot time GDT, LDT, initial page table (init_mm), are currently allocated at static locations in the OS binary. In order to allow the use of superpages over the entire kernel address range, the guest OS is modified to relocate these read-only pages to within a read-only superpage allocated by the special allocator.

5.3.2 Global Page Mappings

Supporting global page mappings for guest domains running in a VM environment is quite simple on the x86 architecture. The x86 architecture allows some mechanisms by which global page mappings can be invalidated from the TLB (This can be done, for example, by disabling and re-enabling the global paging flag in the processor control registers, specifically the PGE flag in the CR4 register).

We modify the Xen VMM to allow guest OSes to use global page mappings in their address space. On each domain switch, the VMM is modified to flush all TLB entries, using the mechanism described above. This has the additional side effect that the VMM's global page mappings are also invalidated on a domain switch.

The use of global page mappings potentially improves the TLB performance in the absence of domain switches. However, in the case when multiple domains have to be switched frequently, the benefits of global mappings may be much reduced. In this case, use of global mappings in the guest OS forces both the guest's and the VMM's TLB mappings to be invalidated on each switch. Our evaluation in section 6 shows that global mappings are beneficial only when there is a single driver domain, and not in the presence of guest domains.

We make one additional optimization to the domain switching code in the VMM. Currently, the VMM flushes the TLB whenever it switches to a non-idle domain. We notice that this incurs an unnecessary TLB flush when the VMM switches from a domain `d` to the idle domain and then back to the domain `d`. We make a modification to avoid the unnecessary flush in this case.

5.4 Outstanding issues

There remain a few aspects of virtualization which are difficult to reconcile with the use of superpages in the guest OS. We briefly mention them here.

5.4.1 Transparent page sharing

Transparent page sharing between virtual machines is an effective mechanism to reduce the memory usage in the system when there are a large number of VMs [16]. Page sharing uses address translation from pseudo-physical to physical addresses to transparently share pseudo-physical pages which have the same content.

Page sharing potentially breaks the contiguity of physical page frames. With the use of superpages, either the entire superpage must be shared between the VMs, or no page within the superpage can be shared. This can significantly reduce the scope for memory sharing between VMs.

Although Xen does not make use of page sharing currently, this is a potentially important issue for superpages.

5.4.2 Ballooning driver

The ballooning driver [16] is a mechanism by which the VMM can efficiently vary the memory allocation of guest OSes. Since the use of a ballooning driver potentially breaks the contiguity of physical pages allocated to a guest, this invalidates the use of superpages for that address range.

A possible solution is to force memory allocation/deallocation to be in units of superpage size for coarse grained ballooning operations, and to invalidate superpage mappings only for fine grained ballooning operations. This functionality is not implemented in the current prototype.

6 Evaluation

The optimizations described in the previous sections have been implemented in Xen version 2.0.6, running Linux guest operating systems version 2.6.11.

6.1 Experimental Setup

We use two micro-benchmarks, a transmit and a receive benchmark, to evaluate the networking performance of guest and driver domains. These benchmarks are similar to the netperf [1] TCP streaming benchmark, which measures the maximum TCP streaming throughput over a single TCP connection. Our benchmark is modified to use the zero-copy sendfile system call for transmit operations.

The 'server' system for running the benchmark is a Dell PowerEdge 1600 SC, 2.4 GHz Intel Xeon machine. This machine has four Intel Pro-1000 Gigabit NICs. The 'clients' for running an experiment consist of Intel Xeon machines with a similar CPU configuration, and having one Intel Pro-1000 Gigabit NIC per machine. All the NICs have support for TSO, scatter/gather I/O and checksum offload. The clients and server machines are connected over a Gigabit switch.

The experiments measure the maximum throughput achievable with the benchmark (either transmitter or receiver) running on the server machine. The server is connected to each client machine over a different network interface, and uses one TCP connection per client. We use as many clients as required to saturate the server CPU, and measure the throughput under different Xen and Linux configurations. All the profiling results presented in this section are obtained using the Xenoprof system wide profiler in Xen [10].

6.2 Overall Results

We evaluate the following configurations: 'Linux' refers to the baseline unmodified Linux version 2.6.11 running native mode. 'Xen-driver' refers to the unoptimized XenoLinux driver domain. 'Xen-driver-opt' refers to the Xen driver domain with our optimizations. 'Xen-guest' refers to the unoptimized, existing Xen guest domain. 'Xen-guest-opt' refers to the optimized version of the guest domain.

Figure 3 compares the transmit throughput achieved under the above 5 configurations. Figure 4 shows receive performance in the different configurations.

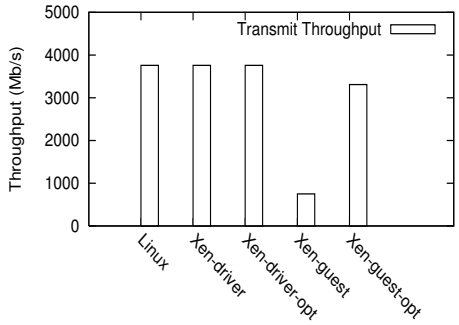

Figure 3: Transmit throughput in different configurations

For the transmit benchmark, the performance of the Linux, Xen-driver and Xen-driver-opt configurations is limited by the network interface bandwidth, and does not fully saturate the CPU. All three configurations achieve an aggregate link speed throughput of 3760 Mb/s. The CPU utilization values for saturating the network in the three configurations are, respectively, 40%, 46% and 43%.

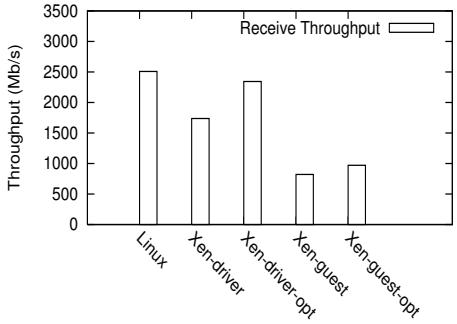

Figure 4: Receive throughput in different configurations

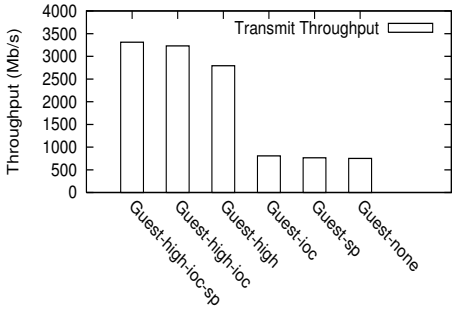

Figure 5: Contribution of individual transmit optimizations

The optimized guest domain configuration, Xen-guest-opt improves on the performance of the unoptimized Xen guest by a factor of 4.4, increasing transmit throughput from 750 Mb/s to 3310 Mb/s. However, Xen-guest-opt gets CPU saturated at this throughput, whereas the Linux configuration reaches a CPU utilization of roughly 40% to saturate 4 NICs.

For the receive benchmark, the unoptimized Xen driver domain configuration, Xen-driver, achieves a throughput of 1738 Mb/s, which is only 69% of the native Linux throughput, 2508 Mb/s. The optimized Xen-driver version, Xen-driver-opt, improves upon this performance by 35%, and achieves a throughput of 2343 Mb/s. The optimized guest configuration Xen-guest-opt improves the guest domain receive performance only slightly, from 820 Mb/s to 970 Mb/s.

6.3 Transmit Workload

We now examine the contribution of individual optimizations for the transmit workload. Figure 5 shows the transmit performance under different combinations of optimizations. 'Guest-none' is the guest domain configuration with no optimizations. 'Guest-sp' is the guest domain configuration using only the superpage optimization. 'Guest-ioc' uses only the I/O channel optimization. 'Guest-high' uses only the high level virtual interface optimization. 'Guest-high-ioc' uses both high level interfaces and the optimized I/O channel. 'Guest-high-ioc-sp' uses all the optimizations: high level interface, I/O channel optimizations and superpages.

The single biggest contribution to guest transmit performance comes from the use of a high-level virtual interface (Guest-high configuration). This optimization improves guest performance by 272%, from 750 Mb/s to 2794 Mb/s.

The I/O channel optimization yields an incremental improvement of 439 Mb/s (15.7%) over the Guest-high configuration to yield 3230 Mb/s (configuration Guest-high-ioc). The superpage optimization improves this further to achieve 3310 Mb/s (configuration Guest-high-ioc-sp). The impact of these two optimizations by themselves in the absence of the high level interface optimization is insignificant.

6.3.1 High level Interface

We explain the improved performance resulting from the use of a high level interface in figure 6. The figure compares the execution overhead (in millions of CPU cycles per second) of the guest domain, driver domain and the Xen VMM, incurred for running the transmit workload on a single NIC, compared between the high-level Guest-high configuration and the Guest-none configuration.

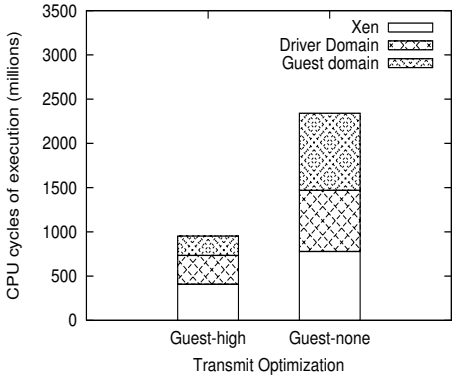

Figure 6: Breakdown of execution cost in high-level and unoptimized virtual interface

The use of the high level virtual interface reduces the execution cost of the guest domain by almost a factor of 4 compared to the execution cost with a low-level interface. Further, the high-level interface also reduces the Xen VMM execution overhead by a factor of 1.9, and

the driver domain overhead by a factor of 2.1. These reductions can be explained by the reasoning given in section 3, namely the absence of data copy because of the support for scatter/gather, and the reduced per-byte processing overhead because of the use of larger packets (TSO).

The use of a high level virtual interface gives performance improvements for the guest domain even when the offload features are not supported in the physical NIC. Figure 7 shows the performance of the guest domain using a high level interface with the physical network interface supporting varying capabilities. The capabilities of the physical NIC form a spectrum, at one end the NIC supports TSO, SG I/O and checksum offload, at the other end it supports no offload feature, with intermediate configurations supporting partial offload capabilities.

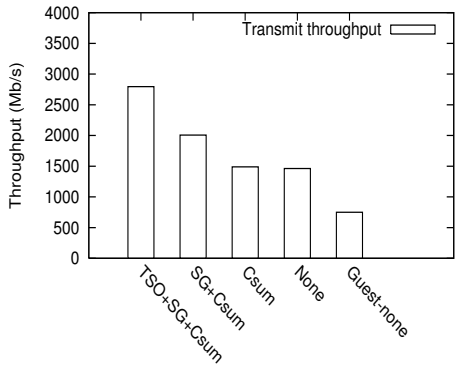

Figure 7: Advantage of higher-level interface

Even in the case when the physical NIC supports no offload feature (bar labeled 'None'), the guest domain with a high level interface performs nearly twice as well as the guest using the default interface (bar labeled Guest-none), viz. 1461 Mb/s vs. 750 Mb/s.

Thus, even in the absence of offload features in the NIC, by performing the offload computation just before transmitting it over the NIC (in the offload driver) instead of performing it in the guest domain, we significantly reduce the overheads incurred in the I/O channel and the network bridge on the packet's transmit path.

The performance of the guest domain with just checksum offload support in the NIC is comparable to the performance without any offload support. This is because, in the absence of support for scatter/gather I/O, data copying both with or without checksum computation incur effectively the same cost. With scatter/gather I/O support, transmit throughput increases to 2007 Mb/s.

6.3.2 I/O Channel Optimizations

Figure 5 shows that using the I/O channel optimizations in conjunction with the high level interface (configuration Guest-high-ioc) improves the transmit performance from 2794 Mb/s to 3239 Mb/s, an improvement of 15.7%.

This can be explained by comparing the execution profile of the two guest configurations, as shown in figure 8. The I/O channel optimization reduces the execution overhead incurred in the Xen VMM by 38% (Guest-high-ioc configuration), and this accounts for the corresponding improvement in transmit throughput.

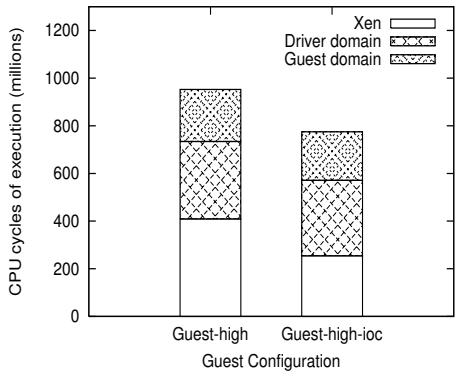

Figure 8: I/O channel optimization benefit

6.3.3 Superpage mappings

Figure 5 shows that superpage mappings improve guest domain performance slightly, by 2.5%. Figure 9 shows data and instruction TLB misses incurred for the transmit benchmark for three sets of virtual memory optimizations. 'Base' refers to a configuration which uses the high-level interface and I/O channel optimizations, but with regular 4K sized pages. 'SP-GP' is the Base configuration with the superpage and global page optimizations in addition. 'SP' is the Base configuration with superpage mappings.

The TLB misses are grouped into three categories: data TLB misses (D-TLB), instruction TLB misses incurred in the guest and driver domains (Guest OS I-TLB), and instruction TLB misses incurred in the Xen VMM (Xen I-TLB).

The use of superpages alone is sufficient to bring down the data TLB misses by a factor of 3.8. The use of global mappings does not have a significant impact on data TLB misses (configurations SP-GP and SP), since frequent switches between the guest and driver domain cancel out any benefits of using global pages.

The use of global mappings, however, does have a negative impact on the instruction TLB misses in the

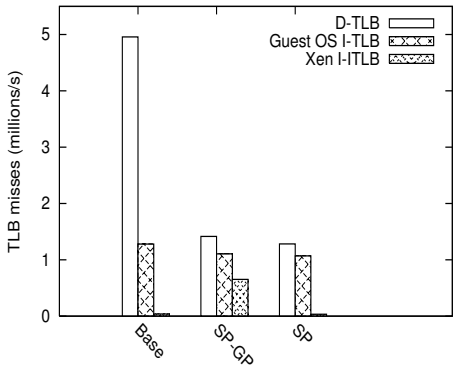

Figure 9: TLB misses for transmit benchmark

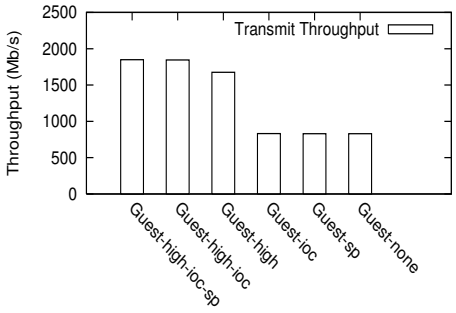

Figure 10: Transmit performance with writes

Xen VMM. As mentioned in section 5.3.2, the use of global page mappings forces the Xen VMM to flush out all TLB entries, including its own TLB entries, on a domain switch. This shows up as a significant increase in the number of Xen instruction TLB misses. (SP-GP vs. SP).

The overall impact of using global mappings on the transmit performance, however, is not very significant. (Throughput drops from 3310 Mb/s to 3302 Mb/s). The optimal guest domain performance shown in section 6.3 uses only the global page optimization.

6.3.4 Non Zero-copy Transmits

So far we have shown the performance of the guest domain when it uses a zero-copy transmit workload. This workload benefits from the scatter/gather I/O capability in the network interface, which accounts for a significant part of the improvement in performance when using a high level interface.

We now show the performance benefits of using a high level interface, and the other optimizations, when using a benchmark which uses copying writes instead of the zero-copy sendfile. Figure 10 shows the transmit performance of the guest domain for this benchmark under the different combinations of optimizations.

The breakup of the contributions of individual optimizations in this benchmark is similar to that for the sendfile benchmark. The best case transmit performance in this case is 1848 Mb/s, which is much less than the best sendfile throughput (3310 Mb/s), but still significantly better than the unoptimized guest throughput.

6.4 Receive Benchmark

Figure 4 shows that the optimized driver domain configuration, Xen-driver-opt, improves the receive performance from 1738 Mb/s to 2343 Mb/s. The Xen-guest-opt configuration shows a much smaller improvement in performance, from 820 Mb/s to 970 Mb/s.

6.4.1 Driver domain

We examine the individual contribution of the different optimizations for the receive workload. We evaluate the following configurations: 'Driver-none' is the driver domain configuration with no optimizations, 'Driver-MTU' is the driver configuration with the I/O channel optimization, which allows the use of MTU sized socket buffers. 'Driver-MTU-GP' uses both MTU sized buffers and global page mappings. 'Driver-MTU-SP' uses MTU sized buffers with superpages, and 'Driver-MTU-SP-GP' uses all three optimizations. 'Linux' is the baseline native Linux configuration.

Figure 11 shows the performance of the receive benchmark under the different configurations.

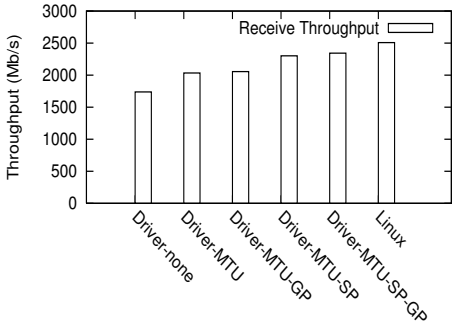

Figure 11: Receive performance in different driver configurations

It is interesting to note that the use of MTU sized socket buffers improves network throughput from 1738 Mb/s to 2033 Mb/s, an increase of 17%. Detailed profiling reveals that the Driver-none configuration, which

uses 4 KB socket buffers, spends a significant fraction of its time zeroing out socket buffer pages. The driver domain needs to zero out socket buffer pages to prevent leakage of information to other domains (section 4.2). The high cost of this operation in the profile indicates that the driver domain is under memory pressure, in which scenario the socket buffer memory allocator constantly frees its buffer pages and then zeroes out newly acquired pages.

The second biggest source of improvement is the use of superpages (configuration Driver-MTU-SP), which improves on the Driver-MTU performance by 13%, from 2033 Mb/s to 2301 Mb/s. The use of global page mappings improves this further to 2343 Mb/s, which is within 7% of the native Linux performance, 2508 Mb/s. The use of global page mappings alone (configuration Driver-MTU-GP) yields an insignificant improvement over the Driver-MTU configuration.

These improvements in performance can be shown to be in direct correspondence with the reduction in TLB misses in the driver domain. Figure 12 shows the effects of the different optimizations on the data TLB misses (millions per sec). For comparison, the data TLB misses incurred in the native Linux configuration is also shown.

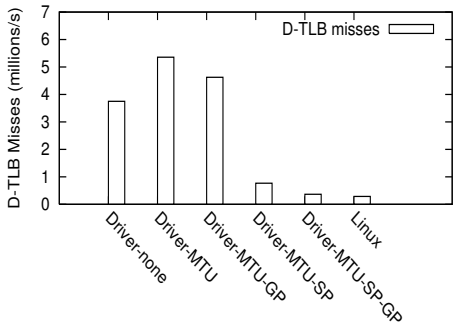

Figure 12: Data TLB misses in Driver domain configurations

The TLB misses incurred under the Xen driver domain configuration (Driver-MTU) are more than an order of magnitude (factor of 18) higher than under the native Linux configuration. The use of superpages in the driver domain (Driver-MTU-SP) eliminates most of the TLB overhead in the driver domain (by 86%). The use of global page mappings (Driver-MTU-GP), by itself, shows only a small improvement in the TLB performance. The combined use of superpages and global mappings brings down the TLB miss count to within 26% of the miss count in Linux.

6.4.2 Guest domain

Figure 13 shows the guest domain performance in three configurations: 'Guest-none' with no optimizations, 'Guest-copy' with the I/O channel optimization and 'Guest-copy-SP' with the I/O channel and superpage optimization. 'Linux' shows the native Linux performance.

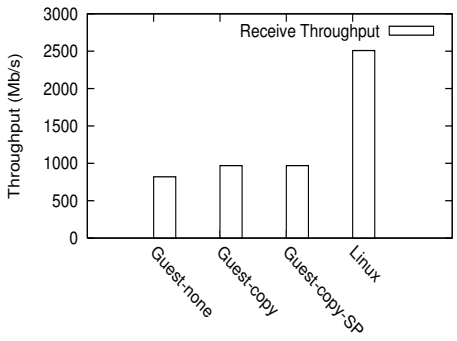

Figure 13: Receive performance in guest domains and Linux

The interesting result from figure 13 is that using copying to implement the I/O channel data transfer between the driver and the guest domain actually performs better than the zero-copy page remapping approach currently used. Copying improves receive performance from 820 Mb/s to 970 Mb/s. As noted in previous sections, the current I/O channel implementation requires the use of two memory allocation/deallocation operations and three page remap operations per packet transfer. The combined cost of these operations is significant enough to outweigh the overhead of data copy for the receive path.

The superpage optimization has no noticeable impact on the receive performance. This is possibly because its benefits are overshadowed by bigger bottlenecks along the receive path. In general, the receive performance in guest domains remains significantly lower than the performance in native Linux. As mentioned in section 2, 70% of the execution time for the receive workload is spent in network virtualization routines in the driver domain and the Xen VMM. Unlike the transmit path optimizations, for the receive path there are no corresponding 'offload' optimizations, which can amortize this cost.

7 Related Work

Virtualization first became available with the IBM VM/370 [6, 14] to allow legacy code to run on new hardware platform. Currently, the most popular full virtu-

alization system is VMware [16]. Several studies have documented the cost of full virtualization, especially on architectures such as the Intel x86.

To address these performance problems, paravirtualization has been introduced, with Xen [4, 5] as its most popular representative. Denali [17] similarly attempts to support a large number of virtual machines.

The use of driver domains to host device drivers has become popular for reasons of reliability and extensibility. Examples include the Xen 2.0 architecture and VMware's hosted workstation [15]. The downside of this approach is a performance penalty for device access, documented, among others, in Sugerman et al. [15] and in Menon et al. [10].

Sugerman et al. describe the VMware hosted virtual machine monitor in [15], and describe the major sources of network virtualization overhead in this architecture. In a fully virtualized system, 'world switches' incurred for emulating I/O access to virtual devices are the biggest source of overhead. The paper describes mechanisms to reduce the number of world switches to improve network performance.

King et al. [8] describe optimizations to improve overall system performance for Type-II virtual machines. Their optimizations include reducing context switches to the VMM process by moving the VM support into the host kernel, reducing memory protection overhead by using segment bounds, and reducing context switch overhead by supporting multiple addresses spaces within a single process.

Our work introduces and evaluates mechanisms for improving network performance in the paravirtualized Xen VMM. Some of these mechanisms, such as the high level virtual interface optimization, are general enough to be applicable to other classes of virtual machines as well.

Moving functionality from the host to the network card is a well-known technique. Scatter-gather DMA, TCP checksum offloading, and TCP segmentation offloading [11, 9] are present on high-end network devices. We instead add these optimizations to the virtual interface definition for use by guest domains, and demonstrate the advantages of doing so.

The cost of copying and frequent remapping is well known to the operating system's community, and much work has been done on avoiding costly copies or remap operations (e.g., in IOLite [13]). Our I/O channel optimizations avoid a remap for transmission, and replace a remap by a copy for reception.

The advantages of superpages are also well documented. They are used in many operating systems, for instance in Linux and in FreeBSD [12]. We provide primitives in the VMM to allow these operating systems to use superpages when they run as guest operating systems on top of the VMM.

Upcoming processor support for virtualization [3, 7] can address the problems associated with flushing global page mappings. Using Xen on a processor that has a tagged TLB can improve performance. A tagged TLB enables attaching address space identifier (ASID) to the TLB entries. With this feature, there is no need to flush the TLB when the processor switches between the hypervisor and the guest OSes, and this reduces the cost of memory operations.

8 Conclusions

In this paper, we presented a number of optimizations to the Xen network virtualization architecture to address network performance problems identified in guest domains.

We add three new capabilities to virtualized network interfaces, TCP segmentation offloading, scatter-gather I/O and TCP checksum offloading, which allow guest domains to take advantage of the corresponding offload features in physical NICs. Equally important, these capabilities also improve the efficiency of the virtualization path connecting the virtual and physical network interfaces.

Our second optimization streamlines the data transfer mechanism between guest and driver domains. We avoid a remap operation in the transmit path, and we replace a remap by a copy in the receive path. Finally, we provide a new memory allocator in the VMM which tries to allocate physically contiguous memory to the guest OS, and thereby allows the guest OS to take advantage of superpages.

Of the three optimizations, the high-level virtual interface contributes the most towards improving transmit performance. The optimized I/O channel and superpage optimizations provide additional incremental benefits. Receive performance in the driver domain benefits from the use of superpages.

Overall, the optimizations improve the transmit throughput of guest domains by a factor of 4.4, and the receive throughput in the driver domain by 35%. The receive performance of guest domains remains a significant bottleneck which remains to be solved.

References

[1] The netperf benchmark. `http://www.netperf.org/netperf/NetperfPage.html`.

[2] Oprofile. `http://oprofile.sourceforge.net`.

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

[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 *19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. 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)*, Oct 2004.

[6] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, Nov. 1983.

[7] Intel. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, April 2005.

[8] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *USENIX Annual Technical Conference*, Jun 2003.

[9] S. Makineni and R. Iyer. Architectural characterization of TCP/IP packet processing on the Pentium M microprocessor. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[10] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, June 2005.

[11] D. Minturn, G. Regnier, J. Krueger, R. Iyer, and S. Makineni. Addressing TCP/IP Processing Challenges Using the IA and IXP Processors. *Intel Technology Journal*, Nov. 2003.

[12] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[13] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, Feb. 2000.

[14] L. Seawright and R. MacKinnon. Vm/370 - a study of multiplicity and usefulness. *IBM Systems Journal*, pages 44–55, 1979.

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

[16] C. Waldspurger. Memory resource management in VMware ESX server. In *Operating Systems Design and Implementation (OSDI)*, Dec 2002.

[17] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali isolation kernel. In *Operating Systems Design and Implementation (OSDI)*, Dec 2002.

Notes

[1] Xen also allows IP routing based or NAT based solutions. However, bridging is the most widely used architecture.