Optimizing Network Virtualization in Xen

Aravind Menon
EPFL, Lausanne
aravind.menon@epfl.ch

Alan L. Cox
Rice University, Houston
alc@cs.rice.edu

Willy Zwaenepoel
EPFL, Lausanne
willy.zwaenepoel@epfl.ch

ABSTRACT

This paper reports on improvements to Xen's networking performance resulting from the re-design and the re-implementation of certain aspects of its network virtualization and better support for advanced, hardware memory management features. Our design remains compatible with the basic Xen 2.0 architecture of locating device drivers in a privileged driver domain that has direct access to the devices, and providing ordinary, unprivileged guest operating systems in guest domains access to the network through virtualized interfaces.

We investigate three techniques. First, we define a new virtual network interface that incorporates many of the optimizations common in hardware network interfaces. We demonstrate that the use of such a virtual interface leads to much better transmit performance, even if the hardware interface does not support the optimizations. Second, we optimize the implementation of the I/O channel between the driver domain and the guest domains. On the transmit side we avoid any remapping of the data into the driver domain, and on the receive side we copy the data, instead of the current page remapping approach. Finally, we provide support for the use of superpages and global page table bits in guest operating systems.

We show that, with these optimizations, transmit performance from a guest domain improves by a factor of 4.4, approaching the performance of native Linux within 12 %. Furthermore, receive performance in the driver domain improves by 35%, coming within 7 % of native Linux performance. Receive performance in a guest domain improves by 18%, but stays well behind the native Linux performance by 61%. We provide a detailed analysis of these results, quantifying the benefits of the individual optimizations.

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 [14, 4], the overhead of network I/O virtualization can still have a significant, negative impact on the performance of network-intensive applications. For instance, Sugerman et al. [14] reported that the CPU utilization required to saturate a 100 Mbps network running Linux 2.2.17 under VMware Workstation 2.0 was 5 to 6 times higher than that when running Linux 2.2.17 natively. Even under the paravirtualized Xen 2.0 VMM [4], Menon et al. [9] reported that the network performance achieved by a VM running Linux 2.6.10 was significantly lower than that achieved running natively. Specifically, they reported a degradation in network throughput by a factor of 2 to 3 for receive traffic, and by a factor of more than 5 for transmit traffic. The latter study, unlike previous reported results on Xen [4, 5], reported network performance for configurations leading to full CPU saturation.

In this paper, we present a number of optimizations to improve the networking performance under the Xen 2.0 VMM. These optimizations address many of the performance limitations identified by Menon et al. [9]. 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 Ethernet 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 of-

fload, 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 [10, 8] 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, approaching the performance of native Linux within 12%. 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 [9]. 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. To virtualize network access, Xen provides each guest domain with a number of 'virtual' network interfaces, which the guest uses for all its network communication. Each virtual interface in the guest domain is 'connected' to a corresponding backend network interface in the driver domain, which in turn is connected to the physical network interfaces through bridging [1]. Data transfer between the virtual and backend network interface is achieved over an 'I/O channel', which uses a zero-copy page remap mechanism to implement the data transfer.

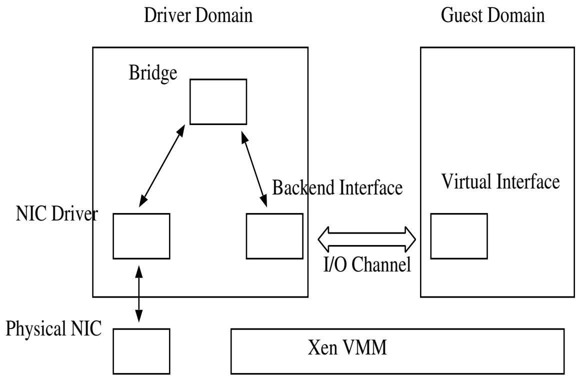

Figure 1: Xen Network I/O Architecture

The combination of page remapping over the I/O channel and packet transfer over the bridge device provides a communication path for multiplexing and demultiplexing packets between the physical interface and the guest's virtual interfaces.

For each packet transmission and reception in the guest domain's virtual interface, the packet needs to be transferred between the guest and the driver domain over the I/O channel, and between the backend and physical network interface through the bridge device.

Table 1 compares the transmit and receive bandwidth achieved by Linux running in Xen guest and driver domains to that achieved natively. 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	3760
Xen driver	1738	3760
Xen guest	820	750

Table 1: Network performance under Xen

The driver domain configuration shows performance comparable to native Linux for the transmit case and a degradation of 24% for the receive case. However, this configuration uses

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

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 65% relative to native Linux, and the transmit performance achieves only 19% of the throughput achievable under native Linux.

Menon et al. [9] 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.

The Xen VMM and driver domain consumed a significant fraction of the network processing time for network processing in the guest domain. For instance, for the network processing to receive a packet in the guest domain, 70% of the execution time was spent in the driver domain and the VMM for transferring the packet from the physical interface to the guest's virtual interface. Similarly, the driver domain and the VMM consumed 60% of the network processing time for a transmit operation to transfer the packet from the virtual interface to the physical interface.

The contribution of the Xen VMM to packet multiplexing/demultiplexing overheads was roughly 40% for receive traffic and 30% for transmit traffic. The remaining 30% of the overhead, in the driver domain, was incurred in bridging network packets between the physical and backend network interfaces, and in the packet processing overheads at the backend and physical interfaces themselves.

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

In this section, we describe optimizations to the virtual network interface used for network I/O in guest domains.

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).

Unfortunately, supporting network offload capabilities in the guest domain's virtual interface is not entirely straightforward. Specifically, it is not sufficient to just provide support for a set of features in the virtual network interface which is identical to the features supported in the physical interface. There may be scenarios in which the presence of multiple physical interfaces can prevent the virtual interface from supporting the offload features of any NIC.

For example, consider a system with two physical network interfaces connected to two different LANs, with each NIC having different offload capabilities. The virtual network interface in the guest domain may be bridged to both the physical interfaces through its backend interface, and the network packets of the guest domain may be forwarded over either physical interface depending on the destination host. Since both physical interfaces have different offload features, and the guest domain's virtual interface cannot determine which physical interface its packet will actually get transmitted on, the only safe option for the virtual interface is to support neither of the offload capabilities of the NICs.

Another complication with supporting offload features identical to the physical interface is that changes to the offload capabilities of the physical NIC (for example, using ethtool) will need to be propagated back to the virtual interfaces supported by this NIC.

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

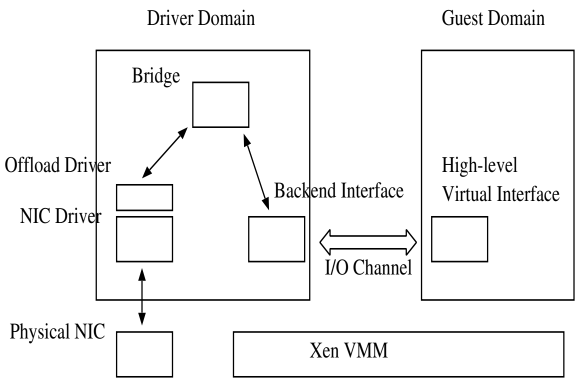

Figure 2: New I/O Architecture

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 [9] 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 [2]. (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 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 trans-

[2] More recent versions of Xen use a new grant-table mechanism. We have not evaluated this mechanism

mit operation. On the receive path, for each network packet (physical page) transferred from the driver to the guest domain, the guest domain releases 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 acquisition or release by a domain requires the previous virtual address mapping to be removed and a new virtual address mapping to be created. For the receive operation, the I/O channel amortizes the required number of operations to two memory allocation/deallocations and three address remap operations per packet. For the transmit path, there is no ownership transfer of the physical page, the page containing the transmit packet is directly mapped into the driver domain, and unmapped on completion of transmit. This incurs two remaps per packet transmitted.

In this section, we describe alternate mechanisms to implement packet transfer, which are expected to be less expensive in the common case scenario. We describe different mechanisms for packet transfer on the transmit path and the receive path.

4.1 Transmit Path

We observe that, for performing network transmits from a guest domain to either another guest domain or to a host on the external network, the driver domain does not need to remap the entire network packet. To forward the packet from the guest domain's backend interface to the target interface, the network bridge in the driver domain only needs to examine the MAC header of the packet. In this case, 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 does need to be mapped in 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

The Xen I/O channel uses page remapping on the receive path to avoid the cost of an extra data copy. Previous research [12] 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 incurrs some additional overheads. Firstly, each network packet has to be allocated on a separate page, so that it can be remapped. For each network packet (page) transferred from the driver to the guest domain, the driver domain has to ensure that no (potentially sensitive) information is leaked through this transferred page. The driver domain ensures this by zeroing out initially, all pages which can be potentially transferred out to guest domains. (The zeroing is done whenever new pages are added to the memory allocator 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. The guest domain avoids a second data copy for the received packet by constructing a network packet with data pointers set to the appropriate memory region in the shared area.

Note that 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 [9], 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.

The use of superpages and global page mappings allows guest OSes to efficiently utilize the TLB, and can have a significant impact on overall system performance.

Unfortunately, supporting these virtual memory primitives in a virtual machine environment is complicated by several factors, such as the fragmented physical memory allocation given to domains and the need to multiplex multiple domains. We describe the issues involved in supporting such primitives in a virtualized environment, and describe the mechanisms we implement to support these in the Xen VMM.

5.1 Virtual Memory primitives

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 [11].

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 Virtual Machine Issues

Unfortunately, supporting the above virtual memory primitives for guest operating systems running in a virtual machine environment is not straightforward.

5.2.1 Superpage Mappings

A superpage mappings maps a contiguous virtual address range to a contiguous set of physical pages. Thus, all the physical pages inside a superpage translation range should belong to the same guest OS, if the guest OS is to use a superpage to map these pages. Unfortunately, in a VM environment, the VMM is often able to allocate only discontiguous, fragmented physical memory to each domain.

More importantly, in some cases, the guest OS may not be even aware that its physical memory allocation is discontiguous.

For instance, in fully virtualized VMMs, such as VMware, the guest OS is given a set of pages which are contiguous in ‘pseudo-physical’ memory [15], which need not be contiguous in physical memory. The VMM uses a layer of translation from pseudo-physical to physical memory pages, and the page table mappings of the guest OS are modified using this translation before they are installed in the MMU.

In such a system, it is difficult for the VMM to allow the guest OS to use superpages, since pseudo-physically contiguous pages need not be physically contiguous.

A second issue with the use of superpages in guest OSes arises due to the granularity of memory protection provided by superpages. In a superpages, all the set of physical pages covered within the superpage have identical virtual memory protection permissions (such as read-only or read-write etc). This can be problematic for situations where the VMM needs to set special protection permissions on certain pages of that guest OS.

As an example, to prevent the guest OS from accessing arbitrary memory, the VMM typically sets read-only permissions on the pages used by the guest OS for page tables. In Xen, the guest OS needs to voluntarily set read-only permissions before installing its page tables, and in VMware, the VMM uses shadow page tables which have read-only permissions for the guest OS’s page table pages. There are other special pages, such as the LDT and GDT pages on x86, which need to be protected with read-only permissions.

Using restricted permissions on certain pages interferes with the use of superpages. Since a superpage uses a single set of permissions, either all pages inside its range would be marked read-only, or all of them would be marked read-write. Since neither of these is acceptable, a superpage mapping cannot be used in the guest OS if any of the pages contained in its range requires restricted permissions.

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

We now describe the mechanisms we implemented in Xen and guest operating systems for supporting the above virtual memory primitives.

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 the memory pages within a superpage address range 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 reenabling 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 [15]. 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 [15] 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

We now present an evaluation of the different optimizations described in the previous sections. These optimizations have been implemented in Xen version 2.0.6, running Linux guest operating systems version 2.6.11.

For the guest domain, we evaluate the impact of all the three optimizations, viz. high level virtual interface, I/O channel optimizations and virtual memory optimizations, on the networking performance of the guest domain. For the driver domain, only the virtual memory optimizations are applicable. We evaluate their impact on driver domain networking performance.

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, 2393 Mhz 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 [9].

6.2 Overall Results

We first measure the overall impact of all the optimizations on the networking performance in the driver and guest domains. For the driver domain, this includes the superpage and global page mapping optimizations. For the guest domain, this includes the high level interface, the I/O channel optimizations, and the superpage optimization.

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.

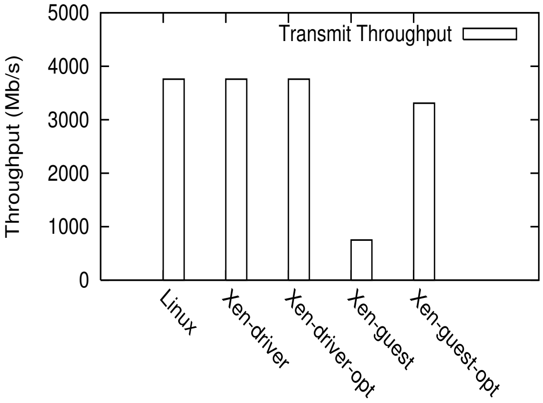

Figure 3: Transmit throughput in different configurations

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 optimized version of the guest domain, Xen-guest-opt achieves a throughput of 3310 Mb/s, compared to 750 Mb/s in the unoptimized guest. This is an improvement of 341% and is within 12% of the native Linux performance. However, we note that Xen-guest-opt gets CPU saturated at this

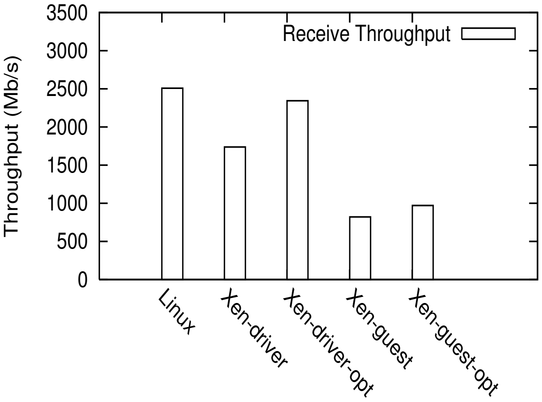

Figure 4: Receive throughput in different configurations

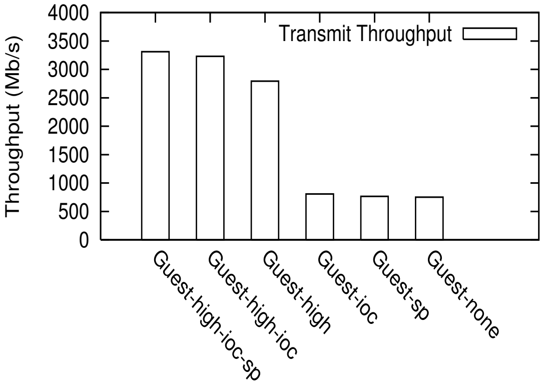

Figure 5: Contribution of individual transmit optimizations

throughput, whereas the Linux configuration reaches a CPU utilization of less than 50% 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 receive performance of the optimized guest configuration, Xen-guest-opt, is only slightly better than that of unoptimized Xen guest. It achieves a throughput of 970 Mb/s vs. the 820 Mb/s in the unoptimized version.

6.3 Transmit Benchmark

We now present the contribution of each optimization in isolation, and in conjunction with other optimizations, to the overall improvement in transmit performance for the guest domain. Note that the driver domain already saturates the NICs for the transmit workload, so we do not evaluate the optimizations for this domain.

We evaluate the following configurations: '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.

Figure 5 shows the transmit performance of the guest domain in the different configurations.

The high-level interface optimization, i.e. configuration Guest-high contributes the most towards improving the guest domain performance. This single optimization improves guest performance from 750 Mb/s to 2794 Mb/s, an increase of 272%.

The superpage and I/O channel optimizations, by themselves, contribute very little to improving the guest domain performance. The increase guest performance by 16 Mb/s and 56 Mb/s, respectively. However, when used in conjunction with the high level virtual interface, they yield non-trivial improvements in performance. The I/O channel optimization improves performance of the high level interface configuration by 439 Mb/s, to yield 3230 Mb/s (configuration Guest-high-ioc), while the addition of the superpage optimization improves this by a further 80 Mb/s to reach 3310 Mb/s (configuration Guest-high-ioc-sp).

This indicates that the high level interface optimization removes a first order bottleneck, in the absence of which the I/O channel and superpage optimizations have no noticeable effect. We now examine in more detail the benefits of each of the optimizations.

6.3.1 High level Interface

Figure 6 compares the execution profiles of the optimized Guest-high configuration with the unoptimized Guest-none configurations for a transmit workload using a single network interface. The figure shows the distribution of the CPU execution cycles (in millions of cycles per second) spent in the Xen VMM, the driver domain and the guest domain. The optimized Guest-high configuration requires much lower CPU utilization to run on a single NIC compared to the unoptimized Guest-none configuration.

The figure shows that the use of a high level virtual interface reduces the execution cost of the guest domain by almost a factor of 4 compared to the execution cost when using a low-level interface. Further, a high-level interface also reduces the time spent in the Xen VMM by a factor of 1.9, and in the driver domain by a factor of 2.1.

As explained in section 3, the reduction in cost of the guest domain can be explained by two factors: because of the support for scatter/gather I/O, the guest domain does not have to perform a data copy for its sendfile operations, and because of the support for TSO in the virtual interface, the

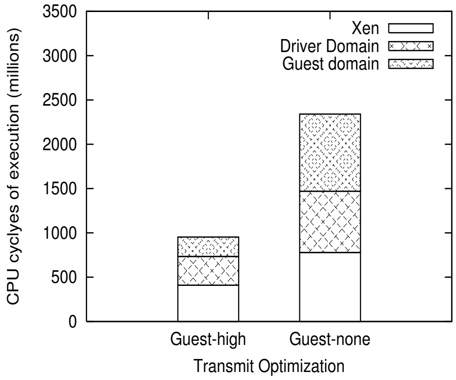

Figure 6: Breakdown of execution cost in high-level and unoptimized virtual interface

guest domain can transmit the same data using fewer packets. The support for TSO also explains the reduced costs of the Xen VMM and the driver domains: fewer packets need to be processed through the I/O channel and the network bridge.

We now show that supporting a high level virtual interface gives performance benefits for the guest domain even when the corresponding offload operations 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. On the Y axis, the throughput of guest domain is shown for the different NIC capabilities. (Note that this configuration does not include the I/O channel optimization and superpage optimization, so the highest throughput is less than the optimal shown in the previous section).

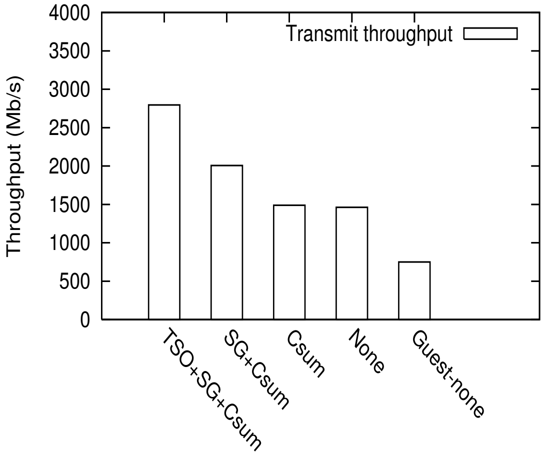

Figure 7: Advantage of higher-level interface

The figure shows that 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), namely 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. Support for TSO in the guest's virtual interface allows it to transmit much bigger packets, which reduces the number of transmit operations required over the transmit virtualizatio 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, without support for scatter/gather I/O, the transmit data needs to be copied to a contiguous location, and checksum computation can be inlined in this phase.

The support for scatter/gather I/O in the NIC further increases throughput to 2007 Mb/s, which shows the benefits of supporting scatter/gather for true zero-copy transmits.

6.3.2 I/O Channel Optimizations

In figure 5, we saw that using the I/O channel optimizations in conjunction with the high level interface (i.e. configuration Guest-high-ioc) improves the transmit performance from 2794 Mb/s to 3239 Mb/s, an improvement of 439 Mb/s. The main cause for this improvement is the decrease in the time spent in the Xen VMM, by avoiding the need to remap the packet fragments from the guest to the driver domain.

Figure 8 compares the execution profile of the two configurations, Guest-high (high level interface) and Guest-high-ioc (high level interface with I/O channel optimizations) for the execution of the transmit benchmark running on a single network interface.

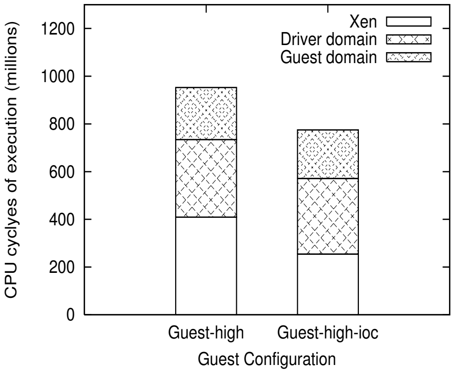

Figure 8: I/O channel optimization benefit

The figure shows that by avoiding the page remap operations in the Guest-high-ioc configuration, the execution time spent in the Xen VMM is reduced by 38%. This accounts for the corresponding improvement in transmit throughput.

The time spent in the guest and driver domains remains nearly the same in both configurations.

6.3.3 Superpage mappings

The use of superpage mappings in the guest domain configurations contributes to a small improvement in the transmit performance of the guest domain. (an increase of 80 Mb/s from 3230 Mb/b in Guest-high-ioc to 3310 Mb/s in Guest-high-ioc-sp). This improvement can be attributed primarily to the decrease in TLB misses with the use of superpages.

We noted in section 5.3.2 that the use of global page mappings could be potentially harmful in the guest domain configuration. This hypothesis is confirmed by the TLB miss measurements with and without global page mappings shown in figure 9.

Figure 9 shows data and instruction TLB misses incurred for the transmit benchmark for three sets of optimizations. 'Regular' refers to a configuration which uses the high-level interface and I/O channel optimizations, but with regular page sized virtual memory mappings. 'SP-GP' refers to the configuration which uses superpages and global page mappings in addition to the above two optimizations. 'SP' refers to the configuration which uses only superpages in addition.

We categorize the TLB misses in these configurations into three groups: 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).

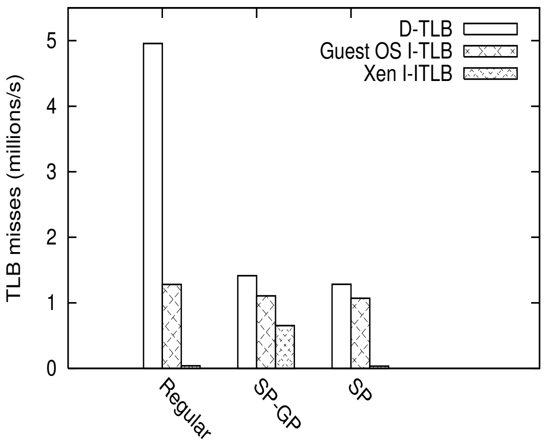

Figure 9: TLB misses for transmit benchmark

Figure 9 shows that the use of superpages alone is sufficient to bring down the data TLB misses by a factor of 3.8. This is also the factor primarily responsible for the improvement in transmit performance. We note that 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 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.

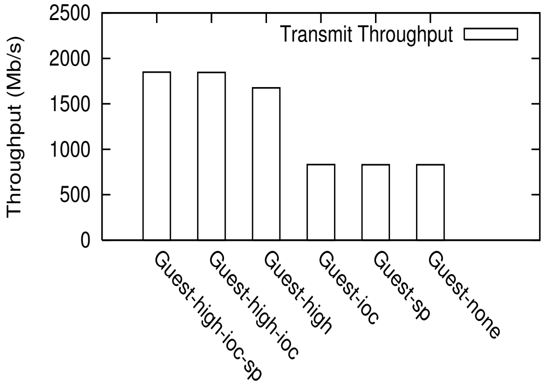

Figure 10: Transmit performance with writes

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

We now examine the performance of the receive benchmark for the guest and driver domain configurations. In figure 4, we saw 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 consider the impact of the following optimizations on the receive performance in the driver domain: 1. superpages, 2. global page mappings, and 3. I/O channel receive optimization. As noted in section 4.2, the use of data copying instead of page remapping allows the use of regular 1500 byte (MTU) sized socket buffers for network packets.

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.

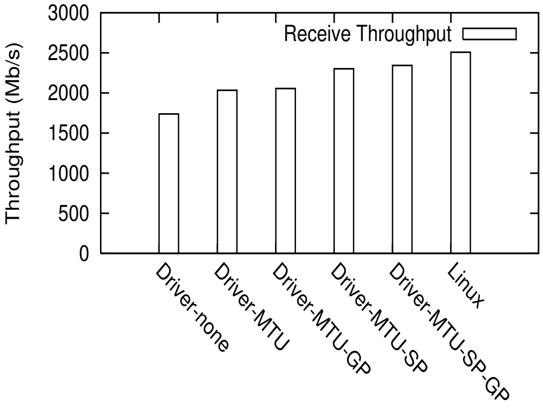

Figure 11: Receive performance in different driver configurations

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 of the Driver-MTU and Driver-none configurations reveals that when the driver domain does not use the I/O channel receive optimization (i.e., it uses 4 KB socket buffers), it spends a significant fraction of the time zeroing out socket buffer pages. We noted in section 4.2 that the driver domain needs to zero out socket buffer pages to prevent leakage of information to other domains. The high cost of this operation indicates memory pressure in the driver domain, under which scenario the socket buffer memory allocator has to constantly release its buffer pages and then zero out newly acquired pages.

After factoring out the improvement from using MTU sized buffers, the second biggest source of improvement in the driver domain is the use of superpages (configuration Driver-MTU-SP). This configuration improves the Driver-MTU performance from 2033 Mb/s to 2301 Mb/s, and improvement of 13%.

The use of global page mapping optimization alone (Driver-MTU-GP) contributes to a only small improvement over the Driver-MTU configuration, from 2033 Mb/s to 2054 Mb/s. The combined use of global pages and superpages (Driver-MTU-SP-GP) yields the best performance, 2343 Mb/s, which reaches within 7% of the native Linux performance of 2508 Mb/s.

The improvements in performance from the virtual memory optimizations 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) incurred in the driver domain for the receive workload. For comparison, we also show the data TLB misses incurred in the native Linux configuration for the same workload.

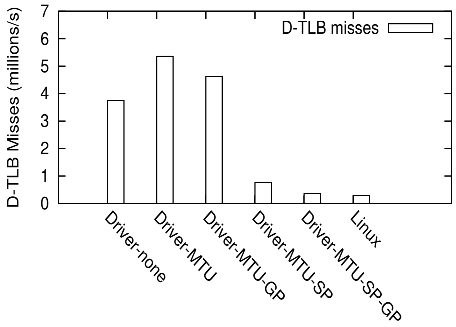

Figure 12: Data TLB misses in Driver domain configurations

The first result to observe from the figure is that 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. This higher TLB miss rate is the main factor responsible for the reduced throughput in the Driver domain configuration relative to Linux (after factoring out the effects of the 4 KB socket buffers). As an aside, the TLB misses in the Driver-none configuration are lower than in the Driver-MTU configuration because Driver-none deals with less data when it sustains lower throughput.

The use of superpages in the driver domain (Driver-MTU-SP) eliminates most of the TLB overhead in the driver domain (by 86%), and the throughput shows a corresponding improvement. 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

We evaluate the benefits of the I/O channel optimization (copying instead of remapping) and the superpage optimization for the receiver performance in the 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.

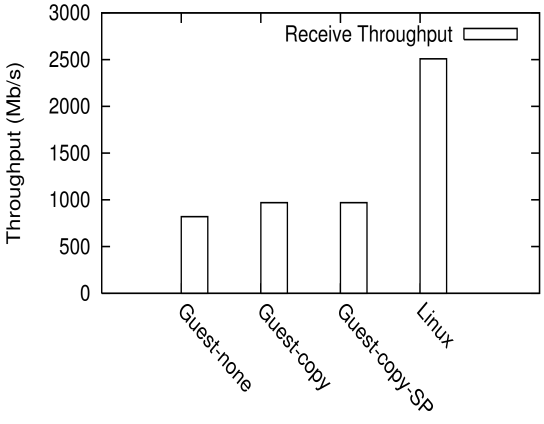

Figure 13: Receive performance in guest domains and Linux

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. Receive path virtualization may thus remain a fundamental bottleneck in network I/O virtualization.

7. RELATED WORK

Virtualization first became available with the IBM VM/370 [6, 13] to allow legacy code to run on new hardware platform. Currently, the most popular full virtualization system is VMWare [15]. 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 [16] 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 [14]. The downside of this approach is a performance penalty for device access, documented, among others, in Sugerman et al. [14] and in Menon et al. [9].

To the best of our knowledge we are the first to propose, implement, and evaluate specific optimizations to address these performance problems. These optimizations borrow from earlier work in different environments.

Moving functionality from the host to the network card is a well-known technique. Scatter-gather DMA, TCP checksum offloading, and TCP segmentation offloading [10, 8] 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 [12]). Our I/O channel optimizations avoid a remap for transmission, and replace a remap by a copy for reception. The latter is useful because of the small size of the MTU relative to the page size.

The advantages of superpages are also well documented. They are used in many operating systems, for instance in Linux and in FreeBSD [11]. 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 have 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. This is evidenced by the fact that the new virtual network interface yields performance benefits, even when the offload operations are not supported in the network device.

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. The copy turns out to be cheaper because the data size is much smaller than the page size, and because a number of expensive memory allocations and deallocations are required for the remap.

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.

Overall, our optimizations improved the transmit throughput of guest domains by more than 300%, and the receive throughput by 35%. The receive performance of guest domains remains a significant bottleneck which remains to be solved.

9. 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. 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

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