Bridging the Gap between Software and Hardware Techniques for I/O Virtualization

Jose Renato Santos, Yoshio Turner, G.(John) Janakiraman, Ian Pratt
HP Laboratories
HPL-2008-39

Keyword(s):
Virtualization, Networking, Operating System, I/O, Performance Analysis, Xen

Abstract:

The paravirtualized I/O driver domain model, used in Xen, provides several advantages including device driver isolation in a safe execution environment, support for guest VM transparent services including live migration, and hardware independence for guests. However, these advantages currently come at the cost of high CPU overhead which can lead to low throughput for high bandwidth links such as 10 gigabit Ethernet. Direct I/O has been proposed as the solution to this performance problem but at the cost of removing the benefits of the driver domain model. In this paper we show how to significantly narrow the performance gap by improving the performance of the driver domain model. In particular, we reduce execution costs for conventional NICs by 56% on the receive path, and we achieve close to direct I/O performance for network devices supporting multiple hardware receive queues. These results make the Xen driver domain model an attractive solution for I/O virtualization for a wider range of scenarios.

External Posting Date: June 21, 2008 [Fulltext] Approved for External Publication

Internal Posting Date: June 21, 2008 [Fulltext]

To be published and presented at 2008 Usenix Annual Technical Conference, June 22-27, 2008, Boston, Massachusetts.

Bridging the Gap between Software and Hardware Techniques for I/O Virtualization

Jose Renato Santos[‡] Yoshio Turner[‡] G.(John) Janakiraman[‡*] Ian Pratt[§]

[‡]*Hewlett Packard Laboratories, Palo Alto, CA*
[§]*University of Cambridge, Cambridge, UK*

Abstract

The paravirtualized I/O driver domain model, used in Xen, provides several advantages including device driver isolation in a safe execution environment, support for guest VM transparent services including live migration, and hardware independence for guests. However, these advantages currently come at the cost of high CPU overhead which can lead to low throughput for high bandwidth links such as 10 gigabit Ethernet. Direct I/O has been proposed as the solution to this performance problem but at the cost of removing the benefits of the driver domain model. In this paper we show how to significantly narrow the performance gap by improving the performance of the driver domain model. In particular, we reduce execution costs for conventional NICs by 56% on the receive path, and we achieve close to direct I/O performance for network devices supporting multiple hardware receive queues. These results make the Xen driver domain model an attractive solution for I/O virtualization for a wider range of scenarios.

1 Introduction

In virtual machine environments like VMware [12], Xen [7], and KVM [23], a major source of performance degradation is the cost of virtualizing I/O devices to allow multiple guest VMs to securely share a single device. While the techniques used for virtualizing CPU and memory resources have very low overhead leading to near native performance [28][7][4], it is challenging to efficiently virtualize most current I/O devices. Each interaction between a guest OS and an I/O device needs to undergo costly interception and validation by the virtualization layer for security isolation and for data multiplexing and demultiplexing [27]. This problem is particularly acute when virtualizing high-bandwidth network interface devices because frequent software interactions with the device are needed to handle the high rate of packet arrivals.

Paravirtualization [29] has been proposed and used (e.g., in Xen [7][13]) to significantly shrink the cost and complexity of I/O device virtualization compared to using full device emulation. In this approach, the guest OS executes a paravirtualized (PV) driver that operates on a simplified abstract device model exported to the guest. The real device driver that actually accesses the hardware can reside in the hypervisor, or in a separate device driver domain which has privileged access to the device hardware. Using device driver domains is attractive because they allow the use of legacy OS device drivers for portability, and because they provide a safe execution environment isolated from the hypervisor and other guest VMs [16][13]. Even with the use of PV drivers, there remains very high CPU overhead (e.g., factor of four) compared to running in non-virtualized environments [18][17], leading to throughput degradation for high bandwidth links (e.g., 10 gigabits/second Ethernet). While there has been significant recent progress making the transmit path more efficient for paravirtualized I/O [17], little has been done to streamline the receive path, the focus of this paper.

To avoid the high performance overheads of software-based I/O device virtualization, efforts in academia and industry are working on adding, to varying degrees, hardware support for virtualization into I/O devices and platforms [10][24][31][22][3][6]. These approaches present a tradeoff between efficiency and transparency of I/O device virtualization. In particular, using hardware support for "direct I/O" [31][24][22], in which a device presents multiple logical interfaces which can be securely accessed by guest VMs bypassing the virtualization layer, results in the best possible performance, with CPU cost close to native performance. However, direct I/O lacks key advantages of a dedicated driver domain model: device driver isolation in a safe execution environment avoiding guest domain corruption by buggy drivers, etc.,

[*] Currently at Skytap.

and full support for guest VM transparent services including live migration [26] [21] [11] and traffic monitoring. Restoring these services would require either additional support in the devices or breaking the transparency of the services to the guest VM. In addition, it is difficult to exploit direct I/O in emerging virtual appliance models of software distribution which rely on the ability to execute on arbitrary hardware platforms. To use direct I/O, virtual appliances would have to include device drivers for a large variety of devices increasing their complexity, size, and maintainability.

In this paper, we significantly bridge the performance gap between the driver domain model and direct I/O in the Xen virtual machine environment, making the driver domain model a competitive and attractive approach in a wider range of scenarios. We first present a detailed analysis of the CPU costs of I/O operations for devices without hardware support for virtualization. Based on this analysis we present implementation and configuration optimizations, and we propose changes to the software architecture to reduce the remaining large costs identified by the analysis: per-packet overheads for memory management and protection, and per-byte data copy overheads. Our experimental results show that the proposed modifications reduce the CPU cost by 56% for a streaming receive microbenchmark. In addition to improving the virtualization of conventional network devices, we propose extensions to the Xen driver domain model to take advantage of emerging network interface devices that provide multiple hardware receive queues with packet demultiplexing performed by the device based on packet MAC address and VLAN ID [10]. The combination of multi-queue devices with our software architecture extensions provides a solution that retains all the advantages of the driver domain model and preserves all the benefits of virtualization including guest-transparent migration and other services. Our results show that this approach has low overhead, incurring CPU cost close to that of direct I/O for a streaming receive microbenchmark.

The rest of the paper is organized as follows. Section 2 reviews the Xen driver domain model. Section 3 presents a detailed analysis and breakdown of the costs of I/O virtualization. Section 4 presents our implementation and architectural changes that significantly improve performance for the driver domain model. Section 5 discusses how configuration settings affect I/O virtualization performance, and Section 6 presents our conclusions.

2 Xen Network I/O Architecture

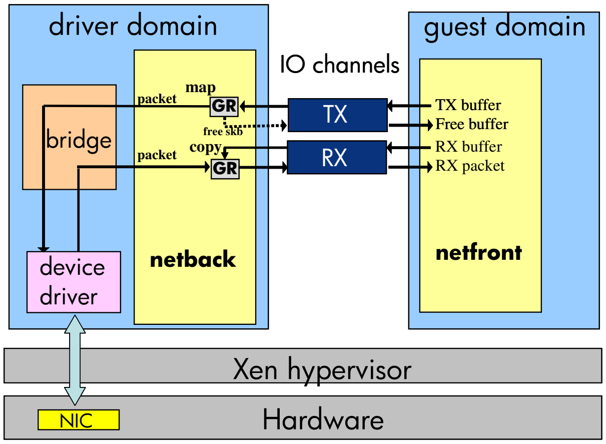

Figure 1: Xen PV driver architecture

Figure 1 shows the architecture of Xen paravirtualized (PV) networking. Guest domains (i.e., running virtual machines) host a paravirtualized device driver, netfront, which interacts with the device indirectly through a separate device driver domain, which has privileged access to the hardware. Driver domains directly access hardware devices that they own; however, interrupts from these devices are first handled by the hypervisor which then notifies the corresponding driver domain through virtual interrupts. Netfront communicates with a counterpart backend driver called netback in the driver domain, using shared memory I/O channels. The driver domain uses a software bridge to route packets among the physical device and multiple guests though their netback interfaces.

Each I/O channel comprises of an event notification mechanism and a bidirection ring of asynchronous requests carrying I/O buffer descriptors from netfront to netback and the corresponding responses. The event notification mechanism enables netfront and netback to trigger a virtual interrupt on the other domain to indicate new requests or responses have been posted. To enable driver domains to access I/O buffers in guest memory Xen provides a page grant mechanism. A guest creates a grant reference providing access to a page and forwards the reference as part of the I/O buffer descriptor. By invoking a hypercall, the driver domain uses the grant reference to access the guest page. For transmit (TX) requests the driver domain uses a hypercall to map the guest page into its address space before sending the request through the bridge. When the physical device driver frees the page a callback function is automatically invoked to return a response to netfront which then revokes the grant. For RX requests netfront posts I/O buffer page grants to the RX I/O channel. When netback receives a packet from the bridge it retrieves a posted grant from the I/O channel and issues a grant copy hypercall to copy the packet to the guest page. Finally, netback sends a response to the guest via the RX channel indicating a packet is available.

Table 1: Classes grouping Linux functions

Class	Description
driver	network device driver and netfront
network	general network functions
bridge	network bridge
netfilter	network filter
netback	netback
mem	memory management
interrupt	interrupt, softirq, & Xen events
schedule	process scheduling & idle loop
syscall	system call
time	time functions
dma	dma interface
hypercall	call into Xen
grant	issuing & revoking grant

Table 2: Classes grouping Xen Function

Class	Description
grant	grant map unmap or copy operation
schedule	domain scheduling
hypercall	hypercall handling
time	time functions
event	Xen events
mem	memory
interrupt	interrupt
entry	enter/exit Xen (hypercall, interrupt, fault),
traps	fault handling (also system call intercept)

3 Xen Networking Performance Analysis

This section presents a detailed performance analysis of Xen network I/O virtualization. Our analysis focuses on the receive path, which has higher virtualization overhead than the transmit path and has received less attention in the literature. We quantify the cost of processing network packets in Xen and the distribution of cost among the various components of the system software. Our analysis compares the Xen driver domain model, the Xen direct I/O model, and native Linux. The analysis provides insight into the main sources of I/O virtualization overhead and guides the design changes and optimizations we present in Section 4.

3.1 Experimental Setup

We run our experiments on two HP c-class blade servers BL460c connected through a gigabit Cisco Catalyst Blade Switch 3020. Each server has two 3GHz Intel Xeon 5160 CPUs (two dual-core CPUs) with 4MB of L2 cache each, 8GB of memory, and two Broadcom NetXtreme II BCM57085 gigabit Ethernet Network Interface Cards (NICs). Although each server had two NICs, only one was used in each experiment presented in this paper.

Table 3: Global function grouping

Class	Description
xen0	Xen functions in domain 0 context
kernel0	kernel functions in domain 0
grantcopy	data copy in grant code
xen	Xen functions in guest context
kernel	kernel funnctions in guest
usercopy	copy from kernel to user buffer

To generate network traffic we used the netperf[1] UDP_STREAM microbenchmark. It would be difficult to separate the CPU costs on the transmit and receive paths using TCP, which generates ACK packets in the opposite direction of data packets. Therefore, we used unidirectional UDP instead of TCP traffic. Although all results are based on UDP traffic, we expect TCP to have similar behavior at the I/O virtualization level.

We used a recent Xen unstable[1] distribution with paravirtualized Linux domains (i.e. a modified Linux kernel that does not require CPU virtualization support) using linux-2.6.18-xen[2]. The system was configured with one guest domain and one privileged domain 0 which was also used as driver domain. For direct I/O evaluation the application was executed directly in domain 0. Both domain 0 and the guest domain were configured with 512MB of memory and a single virtual CPU each. The virtual CPUs of the guest and domain 0 were pinned to different cores of different CPU sockets[3].

We use OProfile [2][18] to determine the number of CPU cycles used in each Linux and Xen function when processing network packets. Given the large number of kernel and hypervisor functions we group them into a small number of classes based on their high level purpose as described in Tables 1 and 2. In addition, when presenting overall results we group the functions in global classes as described in Table 3.

To provide safe direct I/O access to guests an IOMMU [8] is required to prevent device DMA operations from accessing other guests' memory. However, we were not able to obtain a server with IOMMU support. Thus, our results for direct I/O are optimistic since they do not include IOMMU overheads. Evaluations of IOMMU overheads are provided in [9][30].

3.2 Overall Cost of I/O Virtualization

Figure 2 compares the CPU cycles consumed for processing received UDP packets in Linux, and in Xen with direct I/O access from the guest and with Xen paravirtualized (PV) driver.

The graph shows results for three different sizes of UDP messages[4]: 52, 1500 and 48000 bytes. A message with 52 bytes corresponds to a typical TCP ACK,

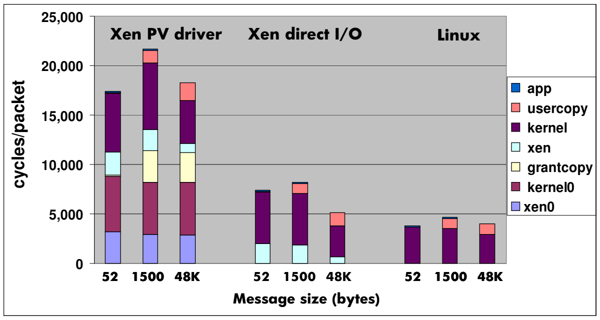

Figure 2: CPU usage to receive UDP packets

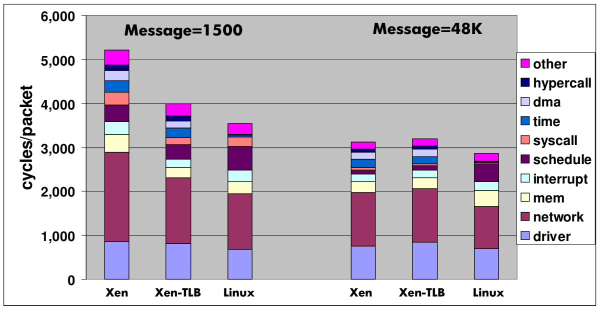

Figure 3: Kernel CPU cost for direct I/O

while a message with 1500 bytes corresponds to the maximum ethernet packet size. Using messages with 48000 bytes also generates maximum size packets but reduces the overhead of system calls at the kernel and application interface by delivering data in larger chunks. The experiments with maximum size packets in Figure 2 and in the remainder results presented in this paper were able to saturate the gigabit link. Experiments with 52-byte packets were not able to saturate the network link as the receive CPU becomes the bottleneck. To avoid having an overloaded CPU with a high number of dropped packets, we throttled the sender rate for small packets to have the same packet rate as with the large packet sizes.

The results show that in the current Xen implementation, the PV driver model consumes significantly more CPU cycles to process received packets than Linux. Direct I/O has much better performance than the PV driver, but it still has significant overhead when compared to Linux, especially for small message sizes. In the rest of this paper we analyze the sources of I/O virtualization overheads in detail, and based on this analysis we propose several changes in the design and implementation of network I/O virtualization in Xen.

3.3 System Call Overhead

We start by looking at the overheads when running guests with direct I/O access. It is surprising that for small message sizes Xen with direct I/O uses twice the number of CPU cycles to process received packets compared to non-virtualized Linux. This is a consequence of memory protection limitations of the 64-bit x86 architecture that are not present in the 32-bit X86 architecture.

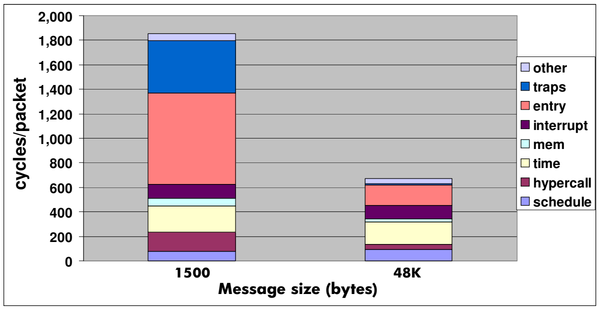

Figure 4: Hypervisor CPU cost for direct I/O

For paravirtualized guests, the entire hypervisor memory is mapped into a small region at the end of every guest virtual address space. In the 32-bit x86 architecture this memory is protected from guest accesses using segments. Segments containing hypervisor memory are only accessible in privilege level 0. The 3 additional privilege levels (rings) in the x86 architecture are available to the guest and enable the kernel to protect its memory from user level code. The 64-bit x86 architecture does not have support for segmentation, forcing Xen to rely on page table protection mechanisms to protect its memory from guest accesses. However, page table entries use only a single bit to distinguish supervisor and user mode, effectively limiting the processor to just two privilege levels. Since Xen uses one privilege level to protect its memory, kernel and user-level applications have to be executed at the same privilege level and therefore cannot share the same set of page tables. This requires Xen to intercept every system call to switch between user and kernel level page tables. In addition to the overhead of the interception, TLB flushes associated with the page table switches reduce performance of the kernel and application code. TLB flushes have negligible effect on hypervisor performance since Xen pages use global mappings and are not flushed from the TLB on system calls.

Figure 3 shows the cost of TLB flushes on the guest kernel. The graph for *Xen-TLB* shows the performance improvement when the TLB flush on system call interception is disabled. This is not a safe optimization as it exposes guest kernel memory to user-level processes but it is used here for performance evaluation purposes. The results in Figure 3 show that the cost of TLB flushes is approximately 30% of the Linux cost when we have one system call per packet (message = 1500 bytes). In contrast, with large message sizes of 48000 bytes the overhead due to TLB flushes is negligible. Figure 4 shows the cost of Xen code used to intercept system calls. The cost difference between the two message sizes corresponds approximately to the system call interception cost. Note that the cost of intercepting system calls and TLB flushes affects guests using PV driver in the same way as they affect guests using direct I/O.

We note that system call interception is an artifact of current hardware limitations which will be eliminated with time. CPU hardware support for virtualization[20][5] provides different partitions for hypervisor and guests and eliminates the need for system call interception. Over time, as the performance of hardware virtualization mechanisms improves, it is expected that paravirtualized guests will also use this mechanism (currently used only by fully virtualized guests) and eliminate the need for system call interception. Given that the overhead for system call interception is temporary and not specific to I/O virtualization we will avoid its effect in the remaining results presented in this paper. In the following graphs we only present results for large message sizes (48000 bytes) where the overhead due to system call interception is negligible.

3.4 Analysis of Direct I/O Performance

Ignoring system call interception we observe that the kernel CPU cost for Xen with direct I/O is similar to Linux, as illustrated in the large message results in Figure 3. Xen with direct I/O consumes approximately 900 more CPU cycles per packet than native Linux (i.e. 31% overhead) for the receive path.

Of these 900 cycles, 250 cycles are due to paravirtualization changes in the kernel code. In particular, direct I/O in Xen has more CPU cycles compared to Linux in DMA functions. This is due to the use of a different implementation of the Linux DMA interface. The DMA interface is a kernel service used by device drivers to translate a virtual address to a bus address used in device DMA operations. Native Linux uses the default (*pci-nommu.c*) implementation, which simply returns the physical memory address associated with the virtual address. Para-virtualized Linux uses the software emulated I/O TLB code ($swioltb.c$), which implements the Linux bounce buffer mechanism. This is needed in Xen because guest I/O buffers spanning multiple pages may not be contiguous in physical memory. The I/O bounce buffer mechanism uses an intermediary contiguous buffer and copies the data to/from its original memory location after/before the DMA operation, in case the original buffer is not contiguous in physical memory. However, the I/O buffers used for regular ethernet packets do not span across multiple pages and thus are not copied into a bounce buffer. The different number of observed CPU cycles is due to the different logic and extra checks needed to verify if the buffer is contiguous, and not due to an extra data copy.

Of the total 900 cycles/packet overhead for direct I/O, the hypervisor accounts for 650 cycles/packet as shown in Figure 7. Most of these cycles are in timer related functions, interrupt processing, and entering and exiting (*entry*) the hypervisor due to interrupts and hypercalls.

3.5 Analysis of PV Driver Performance

Xen PV driver consumes significantly more CPU cycles than direct I/O as shown in Figure 2. This is expected since Xen PV driver runs an additional domain to host the device driver. Extra CPU cycles are consumed both by the driver domain kernel and by the hypervisor which is now executed in the context of two domains compared to one domain with direct I/O. Additional CPU cycles are consumed to copy I/O data across domains using the Xen grant mechanism. The end result is that the CPU cost to process received network packets for Xen PV driver is approximately 18,200 cycles per packet which corresponds to 4.5 times the cost of native Linux. However, as we show in this paper, implementation and design optimizations can significantly reduce this CPU cost.

Of the total 18,200 CPU cycles consumed, approximately 1700 cycles are for copying data from kernel to user buffer (*usercopy*), 4300 for guest kernel functions (*kernel*), 900 for Xen functions in guest context (*xen*), 3000 for copying data from driver domain to guest domain (*grantcopy*), 5400 for driver domain kernel functions (*kernel0*), and 2900 for Xen functions in driver domain context (*xen0*). In contrast native Linux consumes approximately 4000 CPU cycles for each packets where 1100 cycles are used to copy data from kernel to user space and 2900 cycles are consumed in other kernel functions. In the following subsections we examine each component of the CPU cost for Xen PV driver to identify the specific causes of high overhead.

3.5.1 Copy overhead

We note in Figure 2 that both data copies (*usercopy* and *grantcopy*) in Xen PV driver consume a significantly higher number of CPU cycles than the single data copy in native Linux (*usercopy*). This is a consequence of using different memory address alignments for source and destination buffers.

Intel procesor manuals [15] indicate that the processor is more efficient when copying data between memory locations that have the same 64-bit word alignment and even more efficient when they have the same cache line alignment. But packets received from the network are non aligned in order to align IP headers following the typical 14-byte Ethernet header. Netback copies the

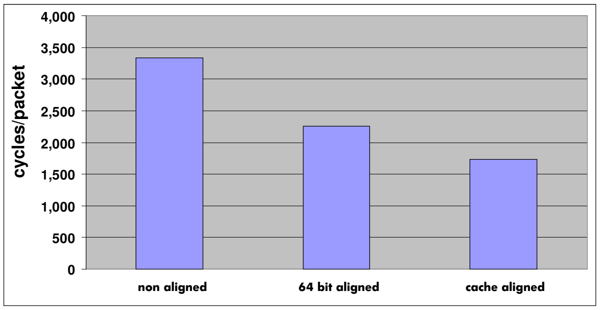

Figure 5: Alignment effect on data copy cost

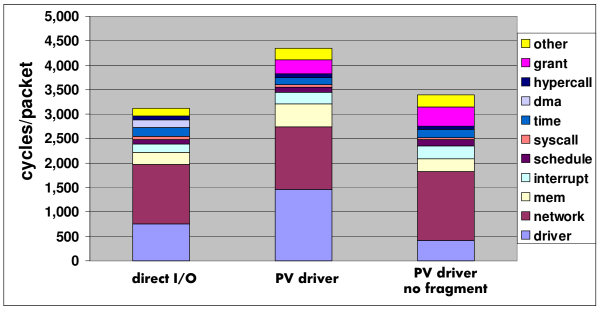

Figure 6: Kernel CPU cost

non aligned packets to the beginning of the granted page which by definition is aligned. In addition, since now the packet starts at a 64-bit word boundary in the guest, the packet payload will start at a non word boundary in the destination buffer, due to the Ethernet header. This causes the second copy from the kernel to the user buffer in the guest to also be misaligned.

The two unaligned data copies consume significantly more CPU cycles than aligned copies. To evaluate this overhead, we modified netback to copy the packet into the guest buffer with an offset that makes the destination of the copy have the same alignment as the source. This is possible because we use a Linux guest which permits changing the socket buffer boundaries after it is allocated. Figure 5 shows the CPU cost of the grant copy for different copy alignments. The first bar shows the number of CPU cycles used to copy a 1500 byte packet when the source and destination have different word alignments. The second bar shows the number of CPU cycles when source and destination have the same 64-bit word alignment but have different cache line alignment (128 bytes), while the third bar shows the number of CPU cycles when source and destination have the same cache line alignment. These results show that proper alignment can reduce the cost of the copy by a factor of two.

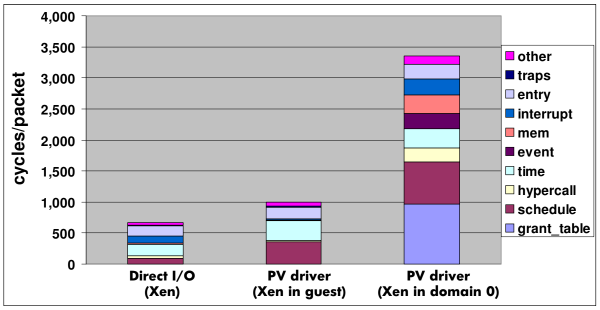

Figure 7: Hypervisor CPU cost

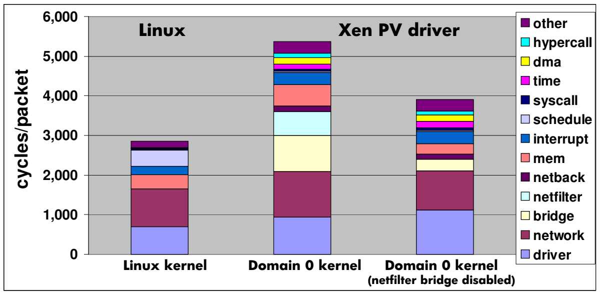

Figure 8: Driver domain CPU cost (kernel)

3.5.2 Kernel overhead

Figure 6 compares the kernel cost for the Xen PV driver model and for the direct I/O model. Xen PV driver uses significantly more CPU cycles than direct I/O, especially in device driver functions. While the device drivers are different (i.e. physical device driver for direct I/O and netfront for PV driver), we would expect that the physical device driver in direct I/O would be more expensive as it has to interact with a real device, while netfront in the PV driver model only accesses the I/O channels hosted in main memory. After careful investigation we determined that the main source of increased CPU cycles in Xen PV driver is the use of fragments in guest socket buffers.

Xen supports TCP Segmentation Offloading (TSO)[17] and Large Receive Offloading (LRO)[14][19] in its virtual interfaces enabling the use of large packets spanning multiple pages for efficient packet processing. For this reason netfront posts full page buffers which are used as fragments in socket buffers, even for normal size ethernet frames. Netfront copies the first 200 bytes of the first fragment into the main linear socket buffer area, since the network stack requires the packet headers in this area. Any remaining bytes past the 200 bytes are kept in the fragment. Use of fragments thereby introduces copy overheads and socket buffer memory allocation overheads.

To measure the cost of using fragments we modified netfront to avoid using fragments. This was accomplished by pre-allocating socket buffers and posting those buffers directly into the I/O channel instead of posting fragment pages. This modification assumes packets will not use multiple pages and thus will not require fragments, since the posted buffers are regular socket buffers and not full page fragments. Since our NIC does not support LRO [14] and it is not configured with jumbo packets, all packets received from the external network use single-page buffers in our experiments. Of course this modification cannot be used for guest to guest communication since a guest can always send large fragmented packets. The third bar in Figure 6 shows the performance results using the modified netfront. The result shows that using socket buffers with fragments is responsible for most of the additional kernel cycles for the PV driver case when compared to direct I/O.

Without the overhead of fragments the kernel CPU cost for PV driver is very close to that of direct I/O, except for small variations in the distribution of cycles among the different kernel functions. In particular, netfront in Xen PV driver now has lower cost than the physical device driver in direct I/O, as expected. Also since netfront does not access a physical device it does not need to use the software I/O TLB implementation of the DMA interface used by direct I/O. On the other hand PV drivers have the cost of issuing and revoking grants, which are not used with direct I/O.

Most of this grant cost is due to the use of expensive atomic compare and swap instructions to revoke grant privileges in the guest grant table. This is necessary because Xen uses different bits of the same grant table field to store the status of the grant (grant in use or not by the driver domain) updated by Xen, and the grant access permission (enable or revoke grant access) updated by the issuing domain. The guest must revoke and check that the grant is no longer in use by the driver domain using an atomic operation, to ensure that a driver domain does not race with grant revoking and keep an undesired reference to the page after the grant is revoked. The use of atomic compare and swap instructions to revoke a grant adds a significant number of CPU cycles in the guest kernel cost. If however these two different grant bits are stored in different words, we can ensure atomicity using less expensive operations such as memory barriers. The result with implementation optimizations presented later in Section 4 include this and other grant optimizations discussed in the following Section 3.5.3.

3.5.3 Hypervisor overhead

I/O processing consumes CPU cycles executing hypervisor functions in addition to guest kernel code. Figure 7 shows the CPU cost in Xen hypervisor functions for receiving network packets with PV driver. The graph shows the CPU cost when executing Xen functions in the context of the guest and in the context of the driver domain (i.e. domain 0), and compares them with the CPU cost for direct I/O.

The graph shows that most of the CPU cost for the hypervisor is due to code executing in the context of the driver domain, and the larger cost components are due to grant operations and schedule functions. The hypervisor CPU cost in guest context for PV drivers is similar to the hypervisor cost for direct I/O except for a higher cost in schedule functions.

The higher cost in scheduling functions for PV driver is due to increased cache misses when accessing Xen data structures for domain scheduling purposes. With PV driver, the guest and the driver domain run on different CPUs, causing domain data structures used by scheduling functions to bounce between the two CPUs.

We identified the most expensive operations for executing grant code by selectively removing code from the grant functions. Basically they are the following: **1)** acquiring and releasing spin locks, **2)** pinning pages, and **3)** use of expensive atomic swap operations to update grant status as previously described.

All of these operations can be optimized. The number of spinlock operations can be significantly reduced by combining the operations of multiple grants in a single critical section. The use of atomic swap operations can be avoided by separating grant fields in different words as described in section 3.5.2. Note that this optimization reduces overhead in Xen and in the guest, since both of them have to access the same grant field atomically.

The cost of pinning pages can also be optimized when using grants for data copy. During a grant copy operation, the hypervisor creates temporary mappings into hypervisor address space for both source and destination of the copy. The hypervisor also pins (i.e. increment a reference counter) both pages to prevent the pages from being freed while the grant is active. However, usually one of the pages is already pinned and mapped in the address space of the current domain which issued the grant operation hypercall. Thus we can avoid mapping and pinning the domain local page and just pin the foreign page referred by the grant. It turns out that pinning pages for writing in Xen is significantly more expensive than pinning pages for read, as it requires to increment an additional reference counter using an expensive atomic instruction. Therefore, this optimization has higher performance impact when the grant is used to copy data from a granted page to a local page (as we propose below in Section 4.1) instead of the other way around.

3.5.4 Driver domain overhead

Figure 8 shows CPU cycles consumed by the driver domain kernel (2nd bar graph; domain 0) and compares it with the kernel cost in native Linux (1st bar graph). The results show that the kernel cost in the driver domain is almost twice the cost of the Linux kernel. This is somewhat surprising, since in the driver domain the packet is only processed by the lower level of the network stack (ethernet), although it is handled by two device drivers: native device driver and netback. We observe that a large number of the CPU cycles in driver domain kernel are due to bridge and netfilter functions. Note that although the kernel has netfilter support enabled, no netfilter rule or filter is used in our experiments. The cost shown in the graph is the cost of netfilter hooks in the bridge code that are always executed to test if a filter needs to be applied. The third bar graph in the figure shows the performance of a driver domain when the kernel is compiled with the bridge netfilter disabled. The results show that most of the bridge cost and all netfilter cost can be eliminated if the kernel is configured appropriately when netfilter rules are not needed.

4 Xen Network Design Changes

In the previous section we identified several sources of overhead for Xen PV network drivers. In this section we propose architectural changes to the Xen PV driver model that significantly improve performance. These changes modify the behavior of netfront and netback, and the I/O channel protocol.

Some inefficiencies identified in Section 3 can be reduced through implementation optimizations that do not constitute architectural or protocol changes. Although some of the implementation optimizations are easier to implement in the new architecture, we evaluated their performance impact in the current architecture in order to separate their performance benefits from that of the architectural changes. The implementation optimizations include: disabling the netfilter bridge, using aligned data copies, avoiding socket buffer fragments and the various grant optimizations discussed in Sections 3.5.2 and 3.5.3.

The second bar in Figure 9 shows the cumulative performance impact of all these implementation optimizations and is used as a reference point for the performance improvements of the architectural changes presented in this section. In summary, the implementation optimizations reduce the CPU cost of Xen PV driver by approximately 4950 CPU cycles per packet. Of these, 1550 cycles are due to disabling bridge netfilter, 1850 cycles due to using cache aligned data copies, 900 cycles due to avoiding socket buffer fragments and 650 cycles due to grant optimizations.

4.1 Move Data Copy to Guest

As described in Section 3 a primary source of overhead for Xen PV driver is the additional data copy between driver domain and guest. In native Linux there is only one data copy, as the received packet is placed directly into a kernel socket buffer by the NIC and later copied from there to the application buffer. In Xen PV driver model there are two data copies, as the received packet is first placed in kernel memory of the driver domain by the NIC, and then it is copied to kernel memory of the guest before it can be delivered to the application.

This extra cost could be avoided if we could transfer the ownership of the page containing the received packet from the driver domain to the guest, instead of copying the data. In fact this was the original approach used in previous versions of Xen [13]. The first versions of Xen PV network driver used a page flipping mechanism which swapped the page containing the received packet with a free guest page, avoiding the data copy. The original page flip mechanism was replaced by the data copy mechanism in later versions of Xen for performance reasons. The cost of mapping and unmapping pages in both guest and driver domain was equivalent to the copy cost for large 1500 byte packets, which means that page flipping was less efficient than copy for small packet sizes [25]. In addition, the page flip mechanism increases memory fragmentation and prevents the use of super-page mappings with Xen. Menon et al. [17] have shown that super-pages provide superior performance in Xen, making page flipping unattractive.

One problem with the current data copy mechanism is that the two data copies per packet are usually performed by different CPUs leading to poor data cache behavior. On an SMP machine, it is expected that an I/O intensive guest and the driver domain will be executing on different CPUs, especially if there is high I/O demand. The overhead introduced by the extra data copy can be reduced if both copies are performed by the same CPU and benefit from cache locality. The CPU will bring the packet to its cache during the first data copy. If the data is still present in the CPU cache during the second copy, this copy will be significantly faster using fewer CPU cycles. Of course there is no guarantee that the data will not be evicted from the cache before the second copy, which can be delayed arbitrarily depending on the application and overall system workload behavior. At high I/O rates, however, it is expected that the data will be delivered to the application as soon as it is received and will benefit from L2 cache locality.

We modified the architecture of Xen PV driver and moved the grant copy operation from the driver domain to the guest domain, improving cache locality for the second data copy. In the new design, when a packet is received netback issues a grant for the packet page to the guest and notifies the guest of the packet arrival through the I/O channel. When netfront receives the I/O channel notification, it issues a grant copy operation to copy the packet from a driver domain page to a local socket buffer, and then delivers the packet to the kernel network stack.

Moving the grant copy to the guest has benefits beyond speeding up the second data copy. It avoids polluting the cache of the driver domain CPU with data that will not be used, and thus should improve cache behavior in the driver domain as well. It also provides better CPU usage accounting. The CPU cycles used to copy the packet will now be accounted to the guest instead of to the driver domain, increasing fairness when accounting for CPU usage in Xen scheduling. Another benefit is improved scalability for multiple guests. For high speed networks (e.g. 10GigE), the driver domain can become the bottleneck and reduce I/O throughput. Offloading some of the CPU cycles to multiple guests' CPUs avoids the driver domain from becoming a bottleneck improving I/O scalability. Finally, some implementation optimizations described earlier are easier to implement when the copy is done by the guest. For example, it is easier to avoid the extra socket buffer fragment discussed in Section 3.5.2. Moving the copy to the guest allows the guest to allocate the buffer *after* the packet is received from netback at netfront. Having knowledge of the received packet allows netfront to allocate the appropriate buffers with the right sizes and alignments. Netfront can allocate one socket buffer for the first page of each packet and additional fragment pages only if the packet spans multiple pages. For the same reason, it is also easier to make aligned data copies, as the right socket buffer alignment can be selected at buffer allocation time.

The third bar in Figure 9 shows the performance benefit of moving the grant copy from the driver domain to the guest. The cost of the data copy is reduced because of better use of the L2 cache. In addition, the number of cycles consumed by Xen in guest context (*xen*) increases while it decreases for Xen in driver domain context (*xen0*). This is because the cycles used by the grant operation are shifted from the driver domain to the guest. We observe that the cost decrease in *xen0* is higher than the cost increase in *xen* leading to an overall reduction in the CPU cost of Xen. This is because the grant optimizations described in Section 3.5.3 are more effective when the grant operations are performed by the guest, as previously discussed.

In summary, moving the copy from the driver domain to the guest reduces the CPU cost for Xen PV driver by approximately 2250 cycles per packet. Of these, 1400 cycles are due to better cache locality for guest copies (*usercopy*, *grantcopy*, and also *kernel* for faster accesses to packet headers in protocol processing), 600 cycles are due to grant optimizations being more effective (*xen* + *xen0*) and 250 cycles are due to less cache pollution in driver domain (*kernel0*).

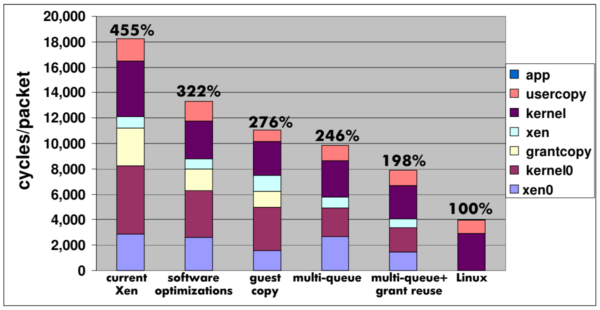

Figure 9: Xen PV driver Optimizations

4.2 Extending Grant Mechanism

Moving the grant copy to the guest requires a couple of extensions to the Xen grant mechanism. We have not yet implemented these extensions but we discuss them here.

To support copy in the receiving guest, the driver domain has to issue a grant to the memory page containing the received packet. This works fine for packets received on the physical device since they are placed in driver domain memory. In contrast, the memory buffers with packets received from other guests are not owned by the driver domain and cannot be granted to the receiving guest using the current Xen grant mechanism. Thus, this mechanism needs to be extended to provide grant transitivity allowing a domain that was granted access to another domain's page, to transfer this right to a third domain.

We must also ensure memory isolation and prevent guests from accessing packets destined to other guests. The main problem is that the same I/O buffer can be reused to receive new packets destined to different guests. When a small sized packet is received on a buffer previously used to receive a large packet for another domain, the buffer may still contain data from the old packet. Scrubbing the I/O pages after or before every I/O to remove old data would have a high overhead wiping out the benefits of moving the copy to the guest. Instead we can just extend the Xen grant copy mechanism with offset and size fields to constrain the receiving guest to access only the region of the granted page containing the received packet data.

4.3 Support for Multi-queue Devices

Although moving the data copy to the guest reduces the CPU cost, eliminating the extra copy altogether should provide even better performance. The extra copy can be

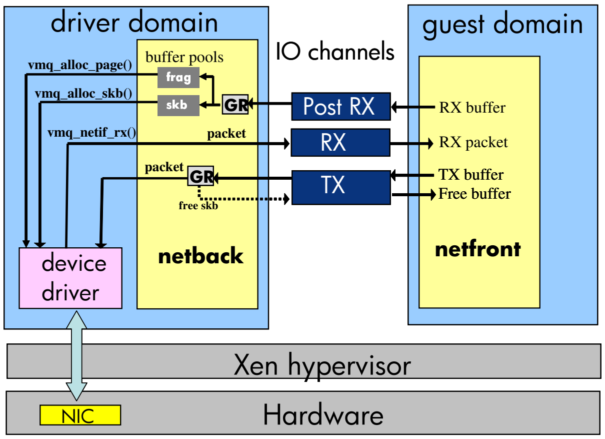

Figure 10: Multi-queue device support

avoided if the NIC can place received packets directly into the guest kernel buffer. This is only possible if the NIC can identify the destination guest for each packet and select a buffer from the respective guest's memory to place the packet. As discussed in Section 1, NICs are now becoming available that have multiple receive (RX) queues that can be used to directly place received packets in guest memory [10]. These NICs can demultiplex incoming traffic into the multiple RX queues based on the packet destination MAC address and VLAN IDs. Each individual RX queue can be dedicated to a particular guest and programmed with the guest MAC address. If the buffer descriptors posted at each RX queue point to kernel buffers of the respective guest, the device can place incoming packets directly into guest buffers, avoiding the extra data copy.

Figure 10 illustrates how the Xen PV network driver model can be modified to support multi-queue devices. Netfront posts grants to I/O buffers for use by the multi-queue device drivers using the I/O channel. For multi-queue devices the driver domain must validate if the page belongs to the (untrusted) guest and needs to pin the page for the I/O duration to prevent the page being reassigned to the hypervisor or other guests. The grant map and unmap operations accomplish these tasks in addition to mapping the page in the driver domain. Mapping the page is needed for guest to guest traffic which traverses the driver domain network stack (bridge). Experimental results not presented here due to space limitations show that the additional cost of mapping the page is small compared to the overall cost of the grant operation.

Netfront allocates two different types of I/O buffers which are posted to the I/O channel: regular socket buffers with the right alignments required by the network stack, and full pages for use as fragments in non linear socket buffers. Posting fragment pages is optional and just needed if the device can receive large packets spanning multiple pages, either because it is configured with jumbo frames or because it supports Large Receive Offload (LRO). Netback uses these grants to map the guest pages into driver domain address space buffers and keeps them in two different pools for each guest: one pool for each type of page. These buffers are provided to the physical device driver on demand when the driver needs to post RX descriptors to the RX queue associated with the guest. This requires that the device driver use new kernel functions to request I/O buffers from the guest memory. This can be accomplished by providing two new I/O buffer allocation functions in the driver domain kernel, $vmq\_alloc\_skb()$ and $vmq\_alloc\_page()$. These are equivalent to the traditional Linux functions $netdev\_alloc\_skb()$ and $alloc\_page()$ except that they take an additional parameter specifying the RX queue for which the buffer is being allocated. These functions return a buffer from the respective pool of guest I/O buffers. When a packet is received the NIC consumes one (or more in case of LRO or jumbo frame) of the posted RX buffers and places the data directly into the corresponding guest memory. The device driver is notified by the NIC that a packet arrived and then forwards the packet directly to netback. Since the NIC already demultiplexes packets, there is no need to use the bridge in the driver domain. The device driver can instead send the packet directly to netback using a new kernel function $vmq\_netif\_rx()$. This function is equivalent to the traditional Linux $netif\_rx()$ typically used by network drivers to deliver received messages, except that the new function takes an additional parameter that specifies which RX queue received the packet.

Note that these new kernel functions are not specific to Xen but enable use of multi-queue devices with any virtualization technology based on the Linux kernel, such as for example KVM[23]. These new functions only need to be used in new device drivers for modern devices that have multi-queue support. These new functions extend the current interface between Linux and network devices enabling the use of multi-queue devices for virtualization. This means that the same device driver for a multi-queue device can be used with Xen, KVM or any other Linux based virtualization technology.

We observe that the optimization that moves the copy to the guest is still useful even when using multi-queue devices. One reason is that the number of guests may exceed the number of RX queues causing some guests to share the same RX queue. In this case guests that share the same queue should still use the grant copy mechanism to copy packets from the driver domain memory to the guest. Also, grant copy is still needed to deliver local guest to guest traffic. When a guest sends a packet to another local guest the data needs to be copied from one guest memory to another, instead of being sent to the physical device. In these cases moving the copy to the guest still provides performance benefits. To support receiving packets from both the multi-queue device and other guests, netfront receives both types of packets on the RX I/O channel shown in Figure 10. Packets from other guests arrive with copy grants that are used by netfront, whereas packets from the device use pre-posted buffers.

We have implemented a PV driver prototype with multi-queue support to evaluate its perfomance impact. However, we did not have a NIC with multi-queue support available in our prototype. We used instead a traditional single queue NIC to estimate the benefits of a multi-queue device. We basically modified the device driver to dedicate the single RX queue of the NIC to the guest. We also modified netback to forward guest buffers posted by netfront to the physical device driver, such that the single queue device could accurately emulate the behavior of a multi-queue device.

The fourth bar in Figure 9 shows the performance impact of using multi-queue devices. As expected the cost of grant copy is eliminated as now the packet is placed directly in guest kernel memory. Also the number of CPU cycles in *xen* for guest context is reduced since there is no grant copy operation being performed. On the other hand the driver domain has to use two grant operations per packet to map and unmap the guest page as opposed to one operation for the grant copy, increasing the number of cycles in *xen0* and reducing the benefits of removing the copy cost. However, the number of CPU cycles consumed in driver domain kernel is also reduced when using multi-queue devices. This is due to two reasons. First, packets are forwarded directly from the device driver to the guest avoiding the forwarding costs in the bridge. Second, since netback is now involved in both allocating and deallocating socket buffer structures for the driver domain, it can avoid the costs of their allocation and deallocation. Instead, netback recycles the same set of socket buffer structures in multiple I/O operations. It only has to change their memory mappings for every new I/O, using the grant mechanism to make the socket buffers point to the right physical pages containing the guest I/O buffers. Surprisingly, the simplifications in driver domain have a higher impact on the CPU cost than the elimination of the extra data copy.

In summary, direct placement of data in guest memory reduces PV driver cost by 700 CPU cycles per packet, and simplified socket buffer allocation and simpler packet routing in driver domain reduces the cost by additional 1150 cycles per packet. On the other hand the higher cost of grant mapping over the grant copy, increases the cost by 650 cycles per packet providing a net cost reduction of 1200 CPU cycles per packet. However, the benefit of multi-queue devices can actually be larger when we avoid the costs associated with grants as discussed in the next section.

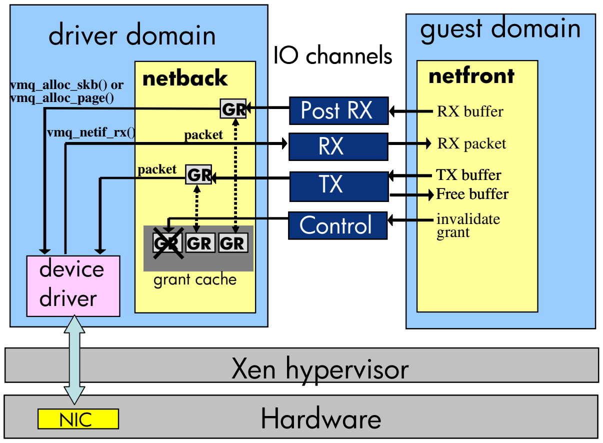

Figure 11: Caching and reusing grants

4.4 Caching and Reusing Grants

As shown in Section 3.5.3, a large fraction of Xen CPU cycles consumed during I/O operations are due to grant operation functions. In this section we describe a grant reuse mechanism that can eliminate most of this cost.

The number of grant operations performed in the driver domain can be reduced if we relax the memory isolation property slightly and allow the driver domain to keep guest I/O buffers mapped in its address space even after the I/O is completed. If the guest recycles I/O memory and reuses previously used I/O pages for new I/O operations, the cost of mapping the guest pages using the grant mechanism is amortized over multiple I/O operations. Fortunately, most operating systems tend to recycle I/O buffers. For example, the Linux slab allocator used to allocate socket buffers keeps previously used buffers in a cache which is then used to allocate new I/O buffers. In practice, keeping I/O buffer mappings for longer times does not compromise the fault isolation properties of driver domains, as the driver domain still can only access the same set of I/O pages and no pages containing any other guest data or code.

In order to reuse grants, Xen PV driver needs to be modified as illustrated in Figure 11. Netback keeps a cache of currently mapped grants for every guest. On every RX buffer posted by the guest (when using a multi-queue device) and on every TX request, netback checks if the granted page is already mapped in its address space, mapping it only if necessary. When the I/O completes, the mapping is not removed allowing it to be reused in future I/O operations. It is important, though, to enable the guest to explicitly request that a cached grant mapping be invalidated. This may be necessary, for example if the guest repurposes the page and uses it somewhere else in the guest or if it returns the page back to the hypervisor. In that case, it is desirable to revoke the grant and unmap the granted page from the driver domain address space, in order to preserve memory isolation between driver domain and guest[5]. A new I/O control channel between netfront and netback is used for this purpose. Netfront sends grant invalidation requests and netback sends confirmation responses after the granted page is unmapped. In summary, this mechanism preserves the isolation between driver domain and guest memory (only I/O buffer pages are shared) and avoids the overhead of mapping and unmapping pages on every I/O operation.

Since the amount of memory consumed for each grant cached in netback is relatively small when compared with the page size, the maximum number of cached grants should be limited only by kernel address space available in the driver domain. The address space reserved for the kernel is significantly larger than the size of a typical active set of I/O buffers. For example, 1GB of the Linux address space is reserved for the kernel; although some of this space is used by other kernel functions, a large fraction of this space can be used for dynamic mapping of guest I/O buffer pages. The size of the active set of I/O buffers is highly dependent on the workload, but typically it should not exceed a few megabytes. In practice, the driver domain should be able to map most of the active I/O buffer pages in its address space for a large number of guests. Thus we expect that the grant reuse mechanism will provide close to a 100% hit rate in the netback grant cache, except for unlikely scenarios with more than hundreds of guests. Thus, the overhead of grant mapping can be reduced to almost zero in practical scenarios, when the guest buffer allocation mechanism promotes buffer reuse.

We have not yet implemented the complete grant reuse mechanism described above. Instead, for evaluation purposes we implemented a simplified mechanism that avoids the use of grants at all. We modified the I/O channel to use physical page addresses directly instead of grants to specify RX buffers. Netfront specifies the machine physical addresses of I/O buffers in the I/O channel requests, and the driver domain uses these addresses directly when programming the DMA operations. Note that this is not a safe mechanism since there is no validation that the physical page used for I/O belongs to the corresponding guest and no guarantee that the page is pinned. Thus, this mechanism is used here just for performance evaluation purposes. The mechanism completely avoids the use of grants and estimates the benefit of the grant reuse mechanism when the hit rate on cached grants is 100%. Although this is an optimistic estimation it is expected to accurately predict actual performance, since we expect to achieve close to a 100% hit rate on cached grants by using appropriate buffer allocation mechanisms. However, validation of this estimation is planned as future work.

The fifth bar in Figure 9 shows the performance benefit of caching and reusing grants in the driver domain. As expected most of the benefit comes from reduced number of cycles in *xen0* used for grant operations. In summary, grant reuse reduces the cost of PV driver by 1950 CPU cycles per packet, which combined with all other optimizations reduces the cost by 10300 CPU cycles. Although the optimizations described so far provide significant cost reduction, Xen I/O virtualization still has twice the cost of native I/O in Linux. However, it is possible to reduce this cost even further by properly setting system configuration parameters as described in the next section.

5 Tuning I/O Virtualization Configuration

5.1 Decoupling Driver Domain from Domain 0

Although the current architecture of the Xen PV driver model is flexible and allows the use of dedicated driver domains used exclusively for doing I/O, in practice most Xen installations are configured with domain 0 acting as the driver domain for all physical devices. The reason is that there is still no mechanism available in Xen that leverages the fault isolation properties of dedicated driver domains. For example there is currently no available tool that automatically detects driver domain faults and restarts them. Since hosting all drivers in domain 0 is simpler to configure this is typically the chosen configuration.

However, hosting all device drivers in domain 0 prevents tuning some configuration options that optimize I/O performance. Since domain 0 is a general purpose OS it needs to support all standard Linux utilities and Xen administration tools, thus requiring a full fledged kernel with support for all features of a general purpose operating system. This limits the flexibility in configuring the driver domain with optimized I/O configuration options. For example, disabling the bridge netfilter option of the kernel significantly improves performance as shown in Section 3.5.4. However, network tools such as iptables available in standard Linux distributions do not work properly if this kernel configuration option is disabled. This has prevented standard Linux vendors such as RedHat and Novell to enable this kernel option in their standard distribution, thus preventing this I/O optimization in practice. Separating the driver domain from domain 0 allows us to properly configure the driver domain with configurations that are optimized for I/O.

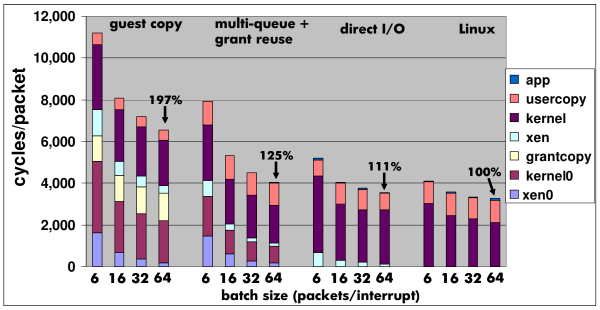

Figure 12: Interrupt throttling

5.2 Reducing Interrupt Rate

The architectural changes discussed in Section 4 address two important sources of overhead in I/O virtualization: extra data copies and the Xen grant mechanism. These are the most important overheads that are directly proportional to the amount of received traffic. Data copy overheads are proportional to the number of received bytes while grant overheads are proportional to the number of received packets.

Most of the remaining overheads are proportional to the number of times the driver domain and the guest are scheduled (this is different from the number of packets since both domains can process multiple packets in a single run). These additional overheads include processing physical interrupts, virtual interrupts, event delivery, domain scheduling, hypercalls, etc. Even though some of these overheads are also present in Linux, they have a higher impact when I/O is virtualized due to the additional levels of software such as the hypervisor and the driver domain. For example, a device interrupt for a received packet causes CPU cycles to be consumed both in the hypervisor interrupt handler and in the driver domain handler. Additional CPU cycles are consumed when handling the virtual interrupt in the guest after the packet is delivered through the I/O channel. Increasing the number of packets that are processed each time the guest or the driver domain is scheduled should reduce the remaining performance overheads.

On the receive path the driver domain is typically scheduled to process I/O when an interrupt is generated by the physical device. Most network devices available today can delay the generation of interrupts until multiple packets are received and thus reduce the interrupt rate at high throughputs. The interrupt rate for received packets can be controlled by special device driver parameters usually known as interrupt coalescing parameters. Interrupt coalescing parameters specify not only the number of packets per interrupt, but also a maximum delay after the last received packet. An interrupt is generated when either the specified number of packets is received or when the maximum specified delay is reached. This mechanism allows us to limit the interrupt rate at high I/O rates while preserving low latency at low I/O rates.

Configuring device coalescing parameters enables us to change the number of packets processed in each run of the driver domain and thus amortize the overheads over a larger number of packets. All packets received in a single interrupt by the device driver are queued in netback queues before being processed. This causes netback to process packets in batches of the same size as the device driver. Since netback only notifies netfront after all packets in the batch are processed and added to the I/O channel, the device coalescing parameters also limit the virtual interrupt rate in the guest domains, and thus amortize the I/O overheads in the guest as well.

Figure 12 shows the effect of interrupt coalescing on the CPU cost of I/O virtualization. The graph shows CPU cost for four cases: 1) Optimized PV driver using a traditional network device (guest copy), 2) Optimized PV driver using a multi-queue device (multi-queue + grant reuse), 3) Direct I/O, 4) native Linux. The figure shows the CPU cost for different interrupt rates for each of the four cases. The Linux default interrupt coalescing parameters for our Broadcom NIC is 6 pkt/int (packets per interrupt) with a maximum latency of 18 μs. While we varied the batch size from 6 to 64 pkt/int we kept the default maximum interrupt latency of 18 μs in all results presented in Figure 12, preserving the packet latency at low throughputs.

The graph shows that the CPU cost for the PV driver is significantly reduced when the number of packets processed per interrupt is increased, while the effect is less pronounced for both Linux and Direct I/O. This result confirms that most of the remaining I/O virtualization overheads are proportional to the interrupt rate. In Linux, the default value of 6 pkt/int performs almost as well as a large batch of 64 pkt/int. This suggests that the default interrupt coalescing parameters for network device drivers that work well for native Linux, are not the best configuration for Xen PV driver.

Experimental results not shown here due to space limitations show that interrupt coalescing achieves approximately the same CPU cost reduction for the original Xen PV driver configuration as it does for our optimized PV driver without hardware multi-queue support, i.e., approximately 4600 cycles per packet for batches of size 64. This indicates that interrupt coalescing and the other optimizations presented in this paper are complementary.

In summary, the results show that software-only optimizations reduce I/O virtualization overheads for the Xen driver domain model from 355% to 97% of the Linux cost for high throughput streaming traffic. Moreover, the use of hardware support for I/O virtualization enables us to achieve close to native performance: multi-queue devices can reduce the overhead to only 25% of the Linux cost while direct I/O has 11% overhead. The main difference is due to the cost of executing netfront and netback when using Xen PV driver. For real applications, the effective cost difference between direct I/O and the driver domain model should much be lower, since applications will use CPU cycles for additional work besides I/O processing. The low cost of the driver domain model combined with multi-queue support suggest that it is an attractive solution for I/O virtualization.

6 Conclusion

The driver domain model used in Xen has several desired properties. It isolates the address space of device drivers from guest and hypervisor code preventing buggy device drivers from causing system crashes. Also, driver domains can support guest VM transparent services such as live migration and network traffic monitoring and control (e.g. firewalls).

However, the driver domain model needs to overcome the address space isolation in order to provide I/O services to guest domains. Device drivers need special mechanisms to gain access to I/O data in other guest domains and to move the I/O data bytes to and from those domains. In Xen this is accomplished through the grant mechanism. In this paper we propose several architectural changes that reduce the performance overhead associated with driver domains models. First we propose two mechanisms that reduce the cost of moving the I/O data bytes between guest and driver domains: 1) we increase the cache locality of the data copy by moving the copy operation to the receiving guest CPU and 2) we avoid the data copy between domains by using hardware support of modern NICs to place data directly into guest memory. Second we mimimize the cost of granting the driver domain access to guest domain pages by slightly relaxing the memory isolation property to allow a set of I/O buffers to be shared between domains across multiple I/O operations. Although these architectural changes were done in the context of Xen they are applicable to the driver domain model in general.

Our work demonstrates that it is possible to achieve near direct I/O and native performance while preserving the advantages of a driver domain model for I/O virtualization. We advocate that the advantages of the driver domain model outweigh the small performance advantage of direct I/O in most practical scenarios.

In addition, this paper identified several low-level optimizations for the current Xen implementation which had surprisingly large impact on overall performance.

References

[1] Netperf. www.netperf.org.

[2] Oprofile. oprofile.sourceforge.net.

[3] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel® virtualization technology for directed I/O. *Intel® Technology Journal 10*, 3 (August 2006). www.intel.com/technology/itj/2006/v10i3/.

[4] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS* (2006), J. P. Shen and M. Martonosi, Eds., ACM, pp. 2–13.

[5] ADVANCED MICRO DEVICES. AMD64 architecture programmer's manual volume 2: System programming. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf, Sept 2007.

[6] ADVANCED MICRO DEVICES, INC. IOMMU architectural specification. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf, Feb 2007. PID 34434 Rev 1.20.

[7] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T. L., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP* (2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 164–177.

[8] BEN-YEHUDA, M., MASON, J., KRIEGER, O., XENIDIS, J., VAN DOORN, L., MALLICK, A., NAKAJIMA, J., AND WAHLIG, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *Ottawa Linux Symposium* (2006).

[9] BEN-YEHUDA, M., XENIDIS, J., OSTROWSKI, M., RISTER, K., BRUEMMER, A., AND VAN DOORN, L. The price of safety: evaluating IOMMU performance. In *Ottawa Linux Symposium* (2007).

[10] CHINNI, S., AND HIREMANE, R. Virtual machine device queues. softwaredispatch.intel.com/?id=1894. Supported in Intel® 82575 gigE and 82598 10GigE controllers.

[11] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI* (2005), USENIX.

[12] DEVINE, S., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMware US Patent 6397242, Oct 1998.

[13] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMS, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (October 2004).

[14] GROSSMAN, L. Large Receive Offload implementation in Neterion 10GbE Ethernet driver. In *Ottawa Linux Symposium* (2005).

[15] INTEL® CORPORATION. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. www.intel.com/products/processor/manuals/index.htm.

[16] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI* (2004), pp. 17–30.

[17] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in Xen. In *USENIX Annual Technical Conference* (June 2006).

[18] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)* (June 2005).

[19] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. In *USENIX Annual Technical Conference* (June 2008).

[20] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel® virtualization technology: hardware support

for efficient processor virtualization. *Intel® Technology Journal 10*, 3 (August 2006). www.intel.com/technology/itj/2006/v10i3/.

[21] NELSON, M., LIM, B.-H., AND HUTCHINS, G. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference* (April 2005).

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

[23] QUMRANET. KVM: Kernel-based virtualization driver. www.qumranet.com/wp/kvm_wp.pdf.

[24] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC* (2007).

[25] SANTOS, J. R., JANAKIRAMAN, G., AND TURNER, Y. Network optimizations for PV guests. In *3rd Xen Summit* (Sept 2006).

[26] SAPUNTZAKIS, C., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *5th Symposium on Operating Systems Design nad Implementation* (December 2002).

[27] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track* (2001), Y. Park, Ed., USENIX, pp. 1–14.

[28] WALDSPURGER, C. A. Memory resource management in VMware ESX Server. In *OSDI* (2002).

[29] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the Denali isolation kernel. In *OSDI* (2002).

[30] WILLMANN, P., COX, A. L., AND RIXNER, S. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference* (June 2008).

[31] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *High Performance Computer Architecture (HPCA)* (February 2007).

Notes

[1] xen-unstable.hg changeset 15521:1f348e70a5af, July 10, 2007

[2] linux-2.6.18-xen.hg changeset 103:a70de77dd8d3, July 10, 2007

[3] Using two cores in the same CPU would have improved performance due to the shared L2 cache. We chose the worst case configuration since we cannot enforce that all guests will share the same socket with the driver domain.

[4] The message size includes IP and UDP headers but does not include 14 bytes of Ethernet header per packet.

[5] Even if the guest does not request that the grant be revoked Xen will not allocate the page to another guest while the grant is active, mantaining safe memory protection between guests.