Evaluating and Optimizing I/O Virtualization in Kernel-based Virtual Machine (KVM)

Binbin Zhang[1], Xiaolin Wang[1], Rongfeng Lai[1], Liang Yang[1], Zhenlin Wang[2], Yingwei Luo[1], and Xiaoming Li[1]

[1] Dept. of Computer Science and Technology, Peking University, Beijing, China, 100871
[2] Dept. of Computer Science, Michigan Technological University, Houghton, USA
{wxl,lyw}@pku.edu.cn, zlwang@mtu.edu

Abstract. I/O virtualization performance is an important problem in KVM. In this paper, we evaluate KVM I/O performance and propose several optimizations for improvement. First, we reduce VM Exits by merging successive I/O instructions and decreasing the frequency of timer interrupt. Second, we simplify the Guest OS by removing redundant operations when the guest OS operates in a virtual environment. We eliminate the operations that are useless in the virtual environment and bypass the I/O scheduling in the Guest OS whose results will be rescheduled in the Host OS. We also change NIC driver's configuration in Guest OS to adapt the virtual environment for better performance.

Keywords: Virtualization, KVM, I/O Virtualization, Optimization.

1 Introduction

Software emulation is used as the key technique in I/O device virtualization in Kernel-based Virtual Machine (KVM). KVM uses a kernel module to intercept I/O requests from a Guest OS, and passes them to QEMU, an emulator running on the user space of Host OS. QEMU translates these requests into system calls to the Host OS, which will access the physical devices via device drivers. This implementation of VMM is simple, but the performance is usually not satisfactory because multiple environments are involved in each I/O operation that results in multiple context switches and long scheduling latency. In recent versions, KVM tries to reduce the I/O virtualization overhead by emulating key devices in the KVM kernel module. However, the main I/O devices are still emulated by QEMU.

In this paper, we evaluate KVM disk and network virtualization overhead and try to optimize it by reducing the overhead of VM Exits[1] and simplifying the corresponding virtualization operations in Guest OS.

The rest of the paper is organized as follows. Section 2 evaluates disk and network performance in a KVM guest. Section 3 presents our optimizations and evaluations. Section 4 discusses related work. And we conclude the paper in Section 5.

[1] A VM Exit is a context switch from Guest OS to VMM. When a sensitive instruction is executed in the Guest OS, a VM Exit will happen. After the VMM emulates the sensitive instruction, the Guest OS can continue to work.

C. Ding, Z. Shao, and R. Zheng (Eds.): NPC 2010, LNCS 6289, pp. 220–231, 2010.

2 KVM I/O Performance Evaluation

A software emulation-based I/O virtualization technique, which is used by KVM and most other host-based VMMs, causes a significant performance overhead. To analyze the sources of the overhead, we begin with a serial of experimental evaluations of KVM's disk and network I/O virtualization performance. We use bonnie++ [6] to evaluate the performance of disk I/O operations, such as character/block read and write, random/sequential read and write. And we use netperf [7] to evaluate network I/O performance, including data throughput, latency, and CPU utilization rate during sending and receiving data using TCP and UDP protocols. Finally, we use SPECjbb [8] to emulate data warehouse's I/O workload.

2.1 Disk I/O Performance

Test-bed: Intel Core 2 Quad Q9550, 2.83GHz, 4G RAM, 500G SATA disk, Linux 2.6.27.7, KVM-76 with default configuration, and the virtual machine uses *raw* disk image.

Results: We run Bonnie++ which evaluates the file system calls in the Guest OS.

Figure 1 illustrates the throughput of the sequential block write benchmarking, as well as the CPU utilization rate during the experiments, of the virtual machine and the native machine.

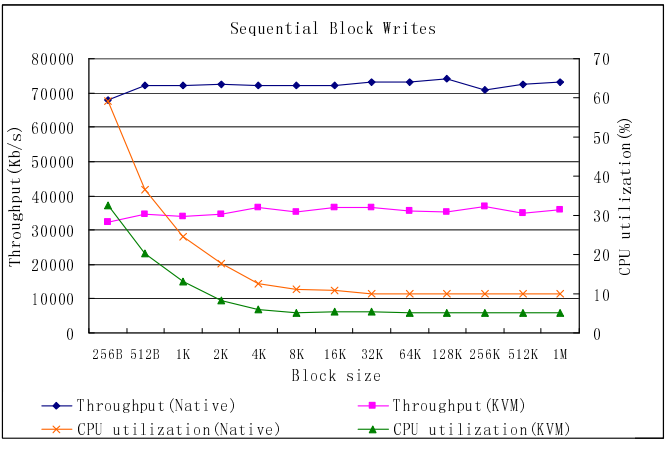


Fig. 1. Comparison of the performance between Native and KVM - Sequential block writes

From Figure 1, it can be observed that the throughput and CPU utilization of sequential block writes on the virtual machine is only about a half compared to the native machine.

Figure 2 illustrates the throughput of the sequential block read benchmarking, as well as the CPU utilization rate during the experiments, of the virtual machine and the Native machine.

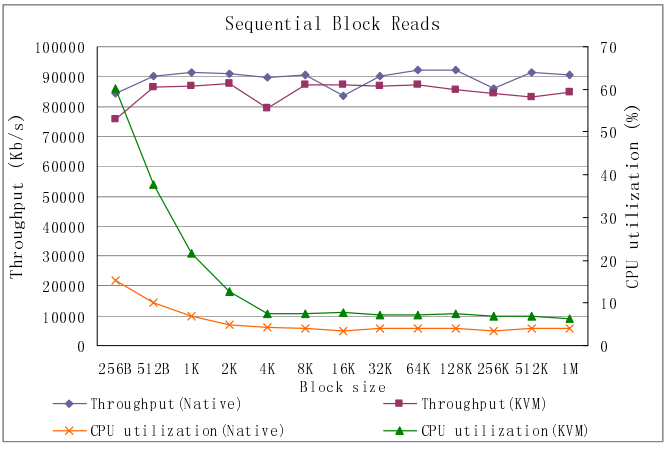


Fig. 2. Comparison of the performance between Native and KVM – Sequential block reads

It can be observed that the throughput of the sequential block read of the virtual machine is very close to that of the Native machine. However, when the size of the disk block is less than 4K, the CPU utilization of the virtual machine is much higher than that of the Native. The reason is that when bonnie++ is reading disk, the data can be frequently hit in the disk cache instead of actually accessing the actual disk during such experiments.

From the experimental data shown in Fig.1 and Fig. 2, it can be deduced that the key to achieve better performance of disk I/O is to improve write throughput and reduce the CPU overhead during disk-reading.

2.2 Network Performance

Test-bed: Two physical machines, HostA and HostB, both using Gigabit Ethernet. HostA works with 82566DC NIC and HostB works with 82567LM-2 NIC. One virtual machine (named GuestB) runs on HostB with KVM-76.

Ping-pong benchmarking: The Round Trip Time (RTT) between HostA and HostB, between HostB and GuestB and that between HostA and GuestB are measured and illustrated in Figure 3.

Based on Figure 3, we are able to estimate the additional network overhead brought by virtualization. Packet transmission time from HostA to GuestB is 0.378 ms, which is equivalent to the transmission time form HostA to HostB (0.234 ms) plus the transmission time from HostB to GuestB (0.126 ms). That is true when the transmission reverses, from GuestB to HostA. That means transmission time between HostB and GuestB can be considered as the virtualization overhead. More accurately, this time overhead accounts for 33% of the transmission time between HostA and GuestB.

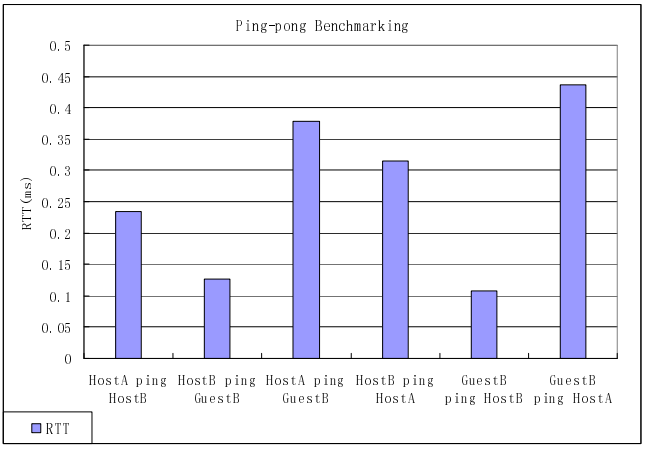


Fig. 3. Ping-pong benchmarking results

Netperf Tests: The throughput and latency between the client and the server is measured using Netperf, and illustrated in Figure 4.

Experiments: 1. Run the netperf server on remote HostA. 2. HostB tests using netperf client. 3. GuestB tests using netperf client, with various configurations, e.g. different virtual NIC and different drivers. Figure 4 shows the result.

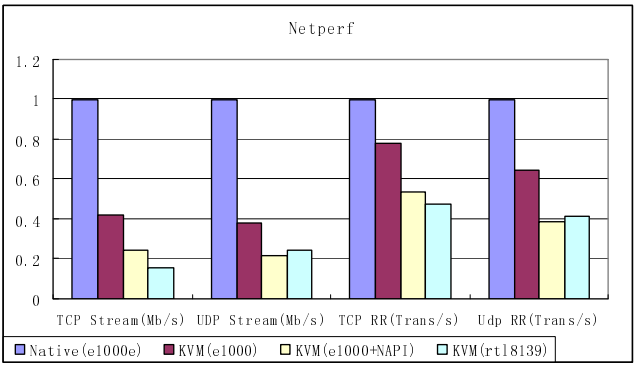


Fig. 4. Netperf benchmark test, Guest OS used different NIC configurations - virtual e1000 NIC (e1000 driver, e1000 driver with NAPI support), virtual rtl8139 NIC

According to the conclusion in Experiment 1 based on ping-pong benchmarking, the throughput on the virtual machine should be 2/3 of that of the Native. In other words, the throughput should be approximately 600Mb/s, not at the level of 400Mb/s as measured. This gap indicates that data processing in the VMM and the Host OS may cause an additional overhead.

From Figure 4, it can be also observed that the virtual device capability greatly influences network performance in a virtual environment.

2.3 Instruction Level Evaluation

To further analyze the reasons of such performance degradation, we intercept the operations issued by Guest OS that result in VM Exits during the SPECjbb and Netperf tests. The top 5 functions, named the *hot functions* which result in most VM Exits, are listed in Table 1 and Table 2. It can be observed that I/O operations have a certain hot code effect, that is, a small number of instructions cause lots of VM Exits. So we can optimize these hot functions to decrease the virtualization overheads.

Table 1. Hot functions during SPECjbb test on KVM VM

	Trap address	Trap count	Trap time(s)/percentage(%)	Trap function
1	0xc041be3f	607251	4.29 / 17.66	ack_ioapic_irq
2	0xc042ffa6	602546	0.10 / 0.41	__do_softirq
3	0xc0407cf9	600625	0.38 / 1.55	timer_interrupt
4	0xc041add2	600223	8.57 / 35.28	smp_apic_timer_interrupt
5	0xc0407cf7	600092	3.08 / 12.68	timer_interrupt

Table 2. Hot functions during Netperf test on KVM VM

	Trap address	Trap count	Trap time(s)/percentage(%)	Trap function
1	0xc04ee0fc	16545487	101.64 / 26.03	ioread16
2	0xc04ee0d1	14317411	85.33 / 21.85	iowrite16
3	0xc04ee153	7636364	62.26 / 15.94	ioread32
4	0xc04edfe4	6045896	44.53 / 11.40	ioread8
5	0xc0623d05	4401573	0.73 / 0.19	_spin_lock_irqrestore

Additionally, in the SPECjbb test, the timer interrupt routine is one of the top 5 time consumers. Therefore, optimization on timer interrupts will improve SPECjbb performance.

We also intercept hot functions in the Bonnie++ benchmark, and the top 5 hot functions are shown in Table 3. Because these functions are frequently called here and there, we further intercepted the caller functions.

Table 3. Hot functions during Bonnie++ test on KVM VM

	Trap address	Trap function	Caller function address	Caller function
1	0xc06442dd	acpi_pm_read	0xc06442e7	verify_pmtmr_rate
2	0xc0547047	iowrite8	0xc05e8fe3	ata_sff_dev_select
3	0xc0546f3c	ioread8	0xc05eaa44	ata_bmdma_status
4	0xc05470ff	iowrite32	0xc05ea9c2	ata_bmdma_setup

We observe that the top caller function *verify_pmtmr_rate* is used to read the clock on motherboard. It is redundant in the virtual environment and thus can be eliminated.

3 Optimization and Evaluation

Based on the experiment results in Section 2, we focus on reducing context switches and simplifying the Guest OS to optimize KVM I/O virtualization. We discuss our optimization methods in this section.

3.1 Reducing the Context Switching

There are multiple occasions for context switching at each I/O operation in KVM, including the context switching among the Guest OS, KVM, and the Host OS. Most switches are caused by the KVM architecture, but the switches between the Guest OS and KVM depend on the behavior of the Guest OS, which may lead to other context switches. If we can modify Guest OS' behavior and remove some VM Exits, the context switches will be reduced. From the instruction test results in Section 2.3, it can be observed that I/O operations in Guest OS are clustered to a certain extent. Some I/O instructions bring a lot of VM Exits, and it can be further optimized.

3.1.1 Merging Successive I/O Instructions

When an I/O instruction is executed in a Guest OS, it will cause a VM exit to KVM. In the Guest OS disk driver, some code fragments include successive I/O instructions, which will cause multiple consecutive VM Exits. If we merge these instructions into a single operation, only a single VM Exit is needed to handle the multiple Guest I/O instructions.

We merge the successive I/O instructions to a single *vmcall*, which will exit to KVM actively. The method is to put information of each instruction into a queue, including IN/OUT, port number, and the value. The address and the length of the queue are passed to KVM as the parameters of *vmcall*. KVM will get information of each instruction from the queue, and emulate them one by one.

For example, Figure 5 is a code fragment in a function (*__ide_do_rw_disk*) in the disk driver code (/driver/ide/ide-disk.c).

```
/driver/ide/ide-disk.c
hwif->OUTB(tasklets[1], IDE_FEATURE_REG);
hwif->OUTB(tasklets[3], IDE_NSECTOR_REG);
hwif->OUTB(tasklets[7], IDE_SECTOR_REG);
hwif->OUTB(tasklets[8], IDE_LCYL_REG);
hwif->OUTB(tasklets[9], IDE_HCYL_REG);
hwif->OUTB(tasklets[0], IDE_FEATURE_REG);
hwif->OUTB(tasklets[2], IDE_NSECTOR_REG);
hwif->OUTB(tasklets[4], IDE_SECTOR_REG);
hwif->OUTB(tasklets[5], IDE_LCYL_REG);
hwif->OUTB(tasklets[6], IDE_HCYL_REG);
hwif->OUTB(0x00|drive->select.all,IDE_SELECT_REG);
```

Fig. 5. A code fragment including successive I/O instructions

11 I/O instructions will be executed concecutively which yield 11 VM Exits. Our approach merges them into one vmcall as the following. The fragment after replacement is illustrated in Figure 6.

```
/driver/ide/ide-disk.c
struct io_insn io_out[11]; // the queue to restore instruction information
usigned long io_gpa, io_len;
#define IO_OUT(x, _type, _val, _port)   \
   io_out[x].type = _type,            \
   io_out[x].port = _port,            \
   io_out[x].val  = _val

//put information of I/O instructions into the queue:
IO_OUT(0, OUTB, tasklets[1], IDE_FEATURE_REG);
IO_OUT(1, OUTB, tasklets[3], IDE_NSECTOR_REG);
IO_OUT(2, OUTB, tasklets[7], IDE_SECTOR_REG);
IO_OUT(3, OUTB, tasklets[8], IDE_LCYL_REG);
IO_OUT(4, OUTB, tasklets[9], IDE_HCYL_REG);
IO_OUT(5, OUTB, tasklets[0], IDE_FEATURE_REG);
IO_OUT(6, OUTB, tasklets[2], IDE_NSECTOR_REG);
IO_OUT(7, OUTB, tasklets[4], IDE_SECTOR_REG);
IO_OUT(8, OUTB, tasklets[5], IDE_LCYL_REG);
IO_OUT(9, OUTB, tasklets[6], IDE_HCYL_REG);
IO_OUT(10, OUTB, 0x00|drive->select.all, IDE_SELECT_REG);

//because the address space is different between KVM and Guest OS, we should translate
io_insn address into physical address:
io_gpa = virt_to_phys((unsigned long)io_out);
io_len = 11; // the number of I/O instructions we have replaced
vmcall(XKVM_IO_COALESCE, io_gpa, io_len, 0); // vmcall to exit into KVM, the
parameters include the address and the length of io_insn.
```

Fig. 6. The code fragment - The successive I/O instructions are replaced into a *vmcall*

We have modified two fragments which include successive port I/O instructions. And the number of VM Exits caused by I/O instructions is reduced. We count the number of VM Exits caused by *inb* and *outb*, and the result is in Table 4:

Table 4. The number of VM Exits caused by inb and outb

Instruction address	Before modification	After modification
0xc054fcc0	111050	17226
0xc054ecf0	15864	2343

This method is actually borrowed from para-virtualization[5], which reduces context switches between the Guest OS and the VMM via modifying the Guest OS. We now only implement static modification. In the future we will try to implement dynamic

modification, which is to replace the code fragments on the fly when the Guest OS is running. One method is to monitor Guest OS execution, locate the fragment which present lots of VM Exits caused by I/O operations, merge the successive I/O instructions and produce the new fragment, then insert jump instructions to replace the old fragment. Another method is to prefetch following instructions after a VM Exit occurs. If there are I/O instructions following the current one, emulate them all at once, and then switch back to Guest OS.

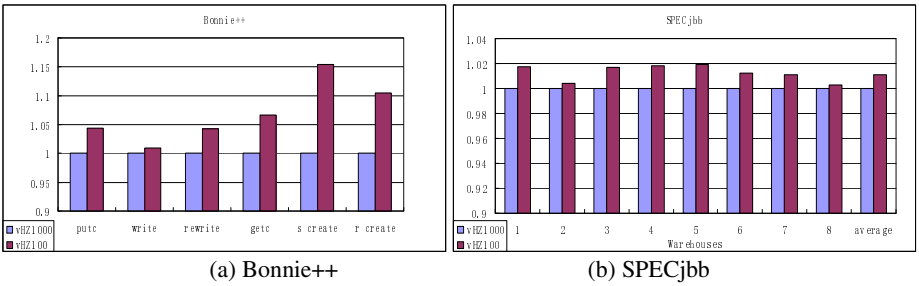
We evaluate the performance of our optimization. Unfortunately, the result is not promising. The CPU overhead is somewhat reduced, but I/O throughput is nearly the same. This may be because we only reduce the overhead brought by context switches, but the operation after VM Exit is not modified. Therefore the cost of VM Exit and VM Entry is not the main reason for the performance overhead.

3.1.2 Reducing the Timer Interrupts

Timer interrupt is another cause of VM Exits. When a timer interrupt happens, a VM Exit will occur. For non-real-time applications, we can reduce the timer frequency to reduce VM Exits.

KVM emulates a PIT (Programmable Interval Timer) for each Guest. The PIT can trigger timer interrupts at a programmed frequency. A PIT consists of an oscillator which produces clock signal at the frequency of (roughly) 1193182HZ. When a clock signal generated, the counter in PIT channel 0 is decreased by 1. When the counter reaches 0, PIT will generate a timer interrupt. The virtual PIT works in a similar way. So if we modify the initial value of the counter in PIT channel 0, the timer frequency is modified. For example, if the counter in PIT channel 0 is initialized 1193, the timer frequency is 1000HZ (1193182/1193). And if the counter is initialized 11932, the timer frequency is 100HZ. If the value initialized by the Guest OS is modified, the timer frequency is modified transparently to Guest OS. The implementation is simple: We only need to modify *pit_load_count* function.

We decrease the Guest actual timer frequency to 100HZ, and we compared its I/O performance to the Guest when the timer frequency is 1000HZ.



(a) Bonnie++ (b) SPECjbb

Fig. 7. Timer Interrupt Evaluation

It can be observed in Figure 7 that the lower timer frequency results up to 15% performance improvement.

3.2 Simplifying Guest OS

A virtual environment is different from a native one. There may be some Guest OS operations that become redundant when the OS operates in a virtual environment. If these redundant operations are removed, the Guest OS can be more efficient.

3.2.1 Removing the Redundant Operations

We try to locate and remove redundant operations in a Guest OS. An example is *verify_pmtmr_rate*. This function causes the most VM Exits during bonnie++ running (see Section 2.3). It is used to adjust timer on the motherboard, which is useless in a virtual environment. We modify the Guest OS by simply removing this function.

The result is illustrated in Figure 8. The write performance is improved across all block sizes.

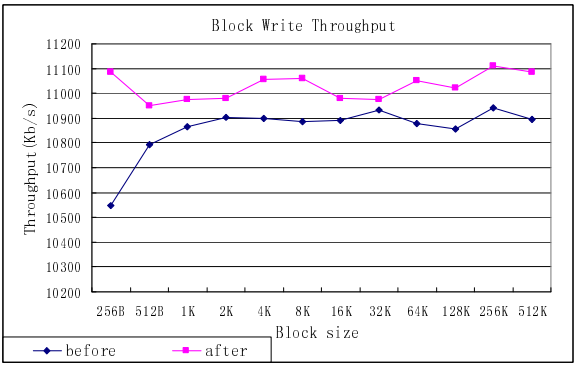


Fig. 8. Write performance (before and after *verify_pmtmr_rate* removed)

Another redundant operation is I/O scheduling in the Guest OS, because all the I/O requests scheduled by the Guest OS will always rescheduled in the Host OS. The scheduler in the Guest OS is thus redundant. For one thing, the Guest OS is unaware of the physical disk information, therefore the results of this scheduler may not be helpful anyway. For another, when multiple Guest OSes operate simultaneously, the Host OS must reschedule I/O requests from all Guest OSs and try to maximize the whole system performance. We remove the I/O scheduler in the Guest OS, and I/O requests are directly submitted to drivers. After our optimization, the throughput of block writes on the virtual machine increases by 6.01%; while throughput of block reads increases by 5.64% (the Host OS uses CFQ I/O scheduler [12] by default). The disadvantage is that all the I/O requests will be submitted in the order of FIFO when the I/O scheduling is eliminated. We cannot arrange set up performance quotas without the I/O scheduler.

3.2.2 Optimizing NIC Driver in Guest OS

A NAPI function is used to reduce interrupt frequency via interrupt masking. But in a virtual environment, a virtual NIC can't generate interrupts directly. So the NAPI support in the NIC driver in the Guest OS is useless and time-consuming. We remove

the NAPI support from the NIC driver in the Guest OS, and evaluate the performance. The result is illustrated in Figures 9.

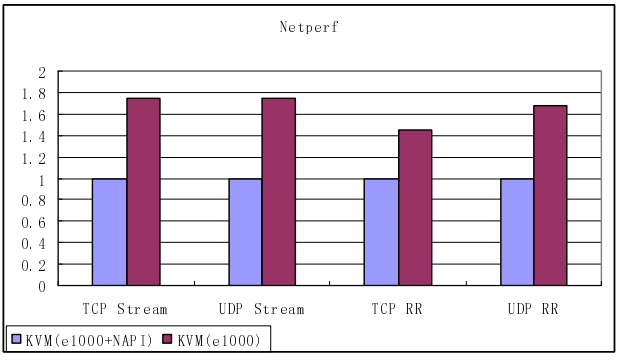


Fig. 9. Netperf performance (with/without NAPI support)

It can be observed that this optimization improves network throughput. In TCP /UDP Stream and UDP Request/Response test, after NAPI support is removed from the NIC driver, the throughput is increased by more than 60%.

4 Related Work

This section discusses the related work that focuses on reducing context switching overhead and modifying Guest OS.

4.1 Reducing the Context Switching Overhead

There are two ways to reduce context switching cost. One is to lower the frequency of such operations, and the other is to reduce the cost of each switching.

Submission in batch is one way to reduce context switch frequency. And another way is to reduce device interrupt if possible. Sugerman et al. [1] have implemented batch submission based VMware Workstation. When the context switch frequency exceeds a certain threshold, the requests will be queued in a cache until the next interrupt happens, and then all queued requests can be handled at one time. This method can also reduce the IRQ transfer cost because only one IRQ is needed once the batch of requests is complete. Another optimization by Sugerman et al. [1] is to improve the Guest driver protocol. They design an interface suitable for a virtual environment. For example, this interface avoids the I/O instructions accessing device status. And it can reduce the number of virtual IRQs which cause context switching between a VMM and a VM. Virtio (R. Russell [4]) is a general device driver which provides the same operation process for block and network devices. Virtio uses para-virtualization for reference, and can be used by various VMMs. The implementation is to maintain buffer rings based on shared memory between a Guest OS and a VMM. One of them posts to the rings, while the other consumes them. And an event notification mechanism is

implemented. When buffers are added to the ring, the other side will be notified. This notification can be masked, to facilitate batching and reduce context switching.

Reduce the cost of each switch is to simplify the operations during each switching. Sugerman et al. [1] modify the process switch operation in a Guest OS on VMware Workstation. When switching to the idle task, the page table is not reloaded, since the idle task is a kernel thread, and can use any process's page table. This optimization cuts the virtualization overhead caused by MMU by a half.

In addition, Sugerman et al. try to optimize VMware Workstation via Host OS bypassing. The optimization is to make the VMM access hardware device directly, and thus avoid context switching between the VMM and the Host OS. This method is employed by full virtualization systems, e.g., VMware ESX Server.

Another cost is due to TLB misses and cache misses after context switches. Aravind Menon et al. [2] enhance the Guest OS in Xen and make it support advanced virtual memory management, e.g. superpage and global page mapping, which greatly reduces TLB misses caused by context switches.

4.2 Simplifying Guest OS

Ram et al. [3] manage to reduce the overhead in a Guest OS in three aspects. Firstly, they implement LRO (Large Receive Offload) which combines a number of data packets into one large-sized TCP/IP packet, so that a large amount of data can be handled during a single protocol stack process. Secondly, they reduce the buffer size to half-page which can reduce the working set and thereby reduce the TLB miss rate.

Menon et al. [2] improve Xen's virtual network interface to support offload features which is supported by most NICs (if the hardware does not support offload, then the Driver domain can simulate it, which can also improve performance). Offload functions include scatter/gather I/O, TCP/IP checksum offload, TCP segmentation offload (TSO). Scatter/gather I/O supports continuous DMA operation with non-contiguous memory. TSO reduces the number of packets to be processed. Checksum offload reduces the Guest OS loads.

Most other optimizations specifically focus on the architecture of Xen, which are not very helpful to other VMMs.

A series of hardware assistance and specifications try to assist device virtualization from the hardware level, including Intel VT-d [9], AMD IOMMU [10], and PCI-SIG IOV [11]. VT-d and IOMMU are similar. They ensure the isolation of I/O address space between different VMs. An I/O MMU, similar to MMU, is installed on the PCI bridge to translate DMA addresses to machine memory addresses. And an IOTLB accelerates this translation. PCI-SIG IOV includes ATS (Address Translation Services), SR-IOV (Single Root IOV), and MR-IOV (Multi-Root IOV). A series of specifications help I/O address translation and let the devices provide multiple interfaces for multiple VMs direct access.

These new technologies and specifications help a guest to access hardware devices directly, avoiding virtualization overhead and simplifying implementation of I/O virtualization. Future research on optimizing I/O virtualization must try to focus on how to optimize the Guest OS to make it more efficient in the virtual environment. Another direction is to examine how to make the Guest OS, the VMM, and the Host OS work better in coordination with each other.

5 Conclusion

We evaluate KVM I/O performance and propose some methods to optimize it. We reduce VM Exits by merging successive I/O instructions and decreasing the frequency of timer interrupt. And we simplify Guest OS by removing redundant operations in the virtual environment.

Guest OS simplification will be an important direction in future research to optimize VMM performance. We will continue to research on how to make a Guest OS more efficient in a virtual environment and how to make the Guest OS, the VMM, and the Host OS coordinate with each other better.

Acknowledgement

This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2007CB310900, National Science Foundation of China under Grant No.90718028 and No. 60873052, National High Technology Research 863 Program of China under Grant No.2008AA01Z112, and MOE-Intel Information Technology Foundation under Grant No. MOE-INTEL-08-09. Zhenlin Wang is also supported by NSF Career CCF0643664.

References

[1] Jeremy, S., Ganesh, V., Beng-Hong, L.: Virtualizing I/O Device on VMware Workstation's Hosted Virtual Machine Monitor. In: Proceddings of the 2001 USENIX Annual Technical Conference (June 2001)

[2] Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in Xen. In: Proceedings of the Annual Technical Conference on USENIX 2006 Annual Technical Conference (2006)

[3] Ram, K.K., Santos, J.R., Turner, Y., Cox, A.L., Rixner, S.: Achieving 10 Gb/s using safe and transparent network interface virtualization. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (2009)

[4] Russell, R.: Virtio: Towards a De-Facto Standard For Virtual I/O Devices. Operating System Review 42(5), 95–103 (2008)

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proceedings of the 19th ACM SOSP (2003)

[6] bonnie++, `http://www.coker.com.au/bonnie++/`

[7] netperf, `http://www.netperf.org/netperf/`

[8] SPEC JBB (2005), `http://www.spec.org/jbb2005/`

[9] Intel Corporation. Intel® Virtualization Technology for Directed I/O Architecture Specification (2007), `http://download.intel.com/technology/computing/vptech/Intelr_VT_for_Direct_IO.pdf`

[10] Advanced Micro Devices, Inc. ADM I/O Virtualization Technology (IOMMU) Specification (2007), `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`

[11] PCI-SIG. I/O Virtualization (2007), `http://www.pcisig.com/specifications/iov/`

[12] Axboe, J.: Time Sliced CFQ I/O Scheduler, `http://kerneltrap.org/node/4406`