Optimizing Xen VMM Based on Intel® Virtualization Technology

Xiantao Zhang[1], Yaozu Dong[2]
[1]School of Computer, Wuhan University, 430079, China
[2]Intel Open source Technology Center, 200241, China,
xiantao.zhang@intel.com; eddie.dong@intel.com

Abstract

To achieve full virtualization support on Xen, Intel extended the Xen project to support unmodified guest OSs using Intel® Virtualization Technology. In this paper, we describe the performance tuning work in the process of enabling full virtualization on Xen. Firstly, we briefly summarize our extensions on performance tuning tools. Then, through data analysis collected by the performance tuning toolkits, we observe that system performance bottlenecks are mainly introduced by memory and I/O virtualization overheads. Finally, we describe our optimizations and possible proposals to address these bottlenecks for resolving performance bugs in Xen/VT architecture. In that process, more specifically, we improved the virtual platform performance by 10% for compute-intensive benchmarks, by moving virtual PIC to hypervisor[1]. In the meantime, we enhanced virtual IDE disk DMA operation performance through concurrent processing and increased virtual network interface card performance more than ten times by reconstruction using an event-driven mechanism. In addition, to achieve higher quality in system X, we also optimized a virtual video graphic card, and several times we got performance gain through our shared virtual video memory approach.

1. Introduction

As early as the 1960s, IBM proposed the concept of virtualization as an approach to share access to expensive hardware resources. Subsequently, this technology was widely applied in IBM VM360/VM370 systems [9]. Recently, as a result of large gains in the computing system resources of one single hardware platform, virtualization has gradually begun to facilitate different applications' requirement demands, such as server consolidation, application performance isolation, dynamic migration of workloads, and research on security areas. Currently, a number of excellent projects have appeared in the virtualization world, such as VMware [11], Virtual Server [15], Xen [1, 5, 7 13], KVM [12], and Virtual Box [16].

Xen is a very famous open source hypervisor, originally developed at the University of Cambridge, which targets to support 100 para-virtualized guest OSs[1] on one physical hardware machine, and brings forth outstanding performance through para-virtualized approach[5]. However, unlike full system virtualization, the para-virtualized approach has its intrinsic shortcomings, because it has to modify the OS kernel to shut down the processor's virtualization holes. To implement full virtualization [17] on x86 platform, Intel, as the leading processor manufacturer, enhances x86 architecture to support full virtualization, as described in the Intel Virtualization Technology Specification [3]. The Intel Open source Technology Center (OTC) extended the Xen project with this novel technology and finally achieved full hardware virtualization and successfully ran unmodified guest OSs with high efficiency [13]. In this paper, we named this project, *Extending Xen with Intel® VT* (Xen/VT, for short), and also called full hardware virtualization with hardware-assisted virtualization. To fully use the new hardware virtualization feature, it is always critical to perform performance tuning work for a mature project. In this paper, we demonstrate such performance tuning exercises, and how to improve system performance with it in the Xen/VT project.

The rest of this paper is organized as follows: we begin with a brief overview of the software architecture used in the Xen/VT project, and we present an overview of VM Exit handling, along with our analysis of performance bottlenecks. In addition, we introduce the performance profiler tool in Section 2. Then, Section 3 describes the performance tuning environment we used in this paper. And, we demonstrated our observations in detail about performance issues in Section 4. Section 5 shows how to optimize performance bugs through our proposed solutions. At last, Section 6 studies the related works on performance tuning in the Xen project. The 7th Section concludes the paper.

2. Background

[1] Also known as Virtual Machine Monitor (VMM)

978-0-7695-3112-0/08 $25.00 © 2008 IEEE
DOI 10.1109/ICICSE.2008.81

In this section, we first briefly describe the Xen/VT architecture, VM Exit handling flow, and the performance profiler tool.

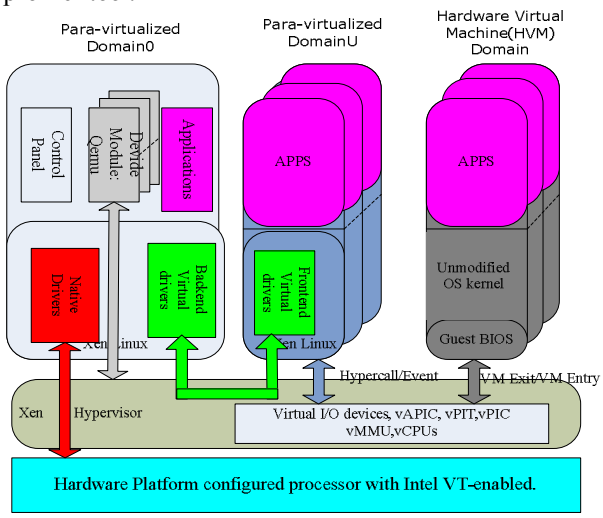

Figure 1. Xen 3.0 Architecture with Hardware-assisted Virtual Machine Support

2.1 Xen/VT Architecture

Figure 1 shows the high-level architecture layout of hardware-based virtualization support for the Xen/VT project [5, 13, 14]. Domain 0 (dom0, for short) is a privileged para-virtualized domain, and it has native device drivers to assist other domains in performing real IO operations [5]. Xen is also capable of running multiple para-virtualized domains, which are non-privileged, to perform I/O operations, named as DomainU, or User domain on one physical machine. Both dom0 and DomainU are available without hardware virtualization by modifying the operating system kernel. When hardware-based virtualization is present on the hardware platform, HVM (Hardware Virtual Machine) domains are available in Xen. For HVM domain, we leverage the processor's VT feature to achieve full virtualization without losing performance, when compared with para-virtualization.

In addition, Qemu [4], originally developed as a fast and portable dynamic translator, is running in dom0 and is used only to emulate virtual I/O devices. Device model process wait events sent by the HVM guest, for I/O operation through Linux system call "select" and response request's results from the event channel[2].

2.2 VM Exit Handling Overview in Xen/VT

To understand potential performance bugs, we briefly examine, in this section, how many VM Exit events are handled in the hypervisor and dom0, and we identify potential performance bottlenecks in the virtual platform. Through analysis, we found that VM Exits related to processor and memory management, like control register and MSR accesses, page table faults, and others, are handled directly in the Hypervisor without resorting to the service domain. A VM Exit caused by an I/O instruction has to require assistance from the device model in dom0. VM Exits caused by asynchronous events like asynchronous interrupt, may require a domain switch to dom0, which may trigger further domain switches. So, it is very critical to handle these kinds of VM Exits in a time-saving way.

To reduce the architecture complexity and development efforts, we initially chose to put the device models in dom0. Due to the long path between dom0 and the HVM domain, some IO-intensive applications had performance issues. One example is I/O operations of the HVM guest. The average cost may be very expensive because the whole process must go through VM Exit to hypervisor, domain switch, event channel, guest scheduler, and device model and back routine. The HVM domain design, however, has taken the advantage of device DMA operation, as much as possible, to reduce the I/O access numbers (such as IDE disk IO). Considering I/O ports, the PIC (both slave and master) are crucial to the overall virtual platform performance, because the guest interrupt handler needs to access these ports several times per interrupt, causing VM Exits, especially for the guest periodic timer interrupt, such as PIT. Frequent VM Exit handling caused by guest periodic timer interrupt will definitely impact all benchmark indicators as an overall virtual system cost.

2.3 Performance Tuning Tools

To better identify performance issues effectively in virtualization environment, we required some new toolkits for the VMM performance profiler, because its architecture is quite different from OSs or apps, which most existing tools were developed for. Before Xen/VT, *Xentrace* was also available to measure para-virtualized guests. Here we extended it to support HVM domains. As a result, it is very helpful in identifying the major performance bottlenecks.

2.3.1 *Xentrace*

Xentrace is a useful performance profiler, tracing all events occurring in Xen VMM. For HVM domain, as long as an unmodified guest OS touches a privileged resource (such as a control register, or fails on a page table walk, or encounters an external interrupt, or others), the processor will trigger a VM Exit event, and pass control to the Xen hypervisor. Accordingly, different VM Exit events need different handling costs in hypervisor. So, performance tuning work success can be measured on the cost data collected by *Xentrace*, and targeted for

[2] Communication mechanism between domains in Xen project.

minimizing the total cost for each VM Exit. Here, *Xentrace* is a perfect tool to analyze VM Exit behavior. *Xentrace* is a built-in method that Xen can use to trace important events happening within the hypervisor. As a whole, it consists of the following three modules:

- An event generator in the hypervisor that saves the event and its corresponding parameter into a trace buffer.
- A daemon in dom0 that saves all produced events into the log file.
- An offline python-based parser reporting statistics data from records into the log file.

The events captured by the hypervisor can be extended easily, by adding more *Xentrace* probes to capture the cost of certain events. For HVM domains, we extended *Xentrace* to add a probe in the beginning and end (that is, the VM Entry) of VM Exit handler, so that the parser can track the time spent handling each VM Exit event. This measures the duration from the last VM Exit to the VM Entry point, including the hypervisor and dom0 handling costs, such as device models, domain switching, and contexts switching. The log collected allows us to exercise further analysis of the CPU frequency level and the cost of each VM Exit event, which some workloads incur.

In our study, we focused on only hypervisor performance tuning. How guest process impacts the overall HVM guest performance is beyond scope of this paper, so we don't discuss the missed process and routine information for the HVM guest application.

3. Tuning Environment Setup

To measure performance of HVM guests during the tuning work, we leveraged some workloads, such as CPU2000 (compute-intensive) and kernel build (CPU, memory and I/O intensive). We also used Linux commands like *hdparm*, *cp,* and *tar* to measure the impact of virtual IDE optimizations, and to leverage *scp* to measure the impact of virtual NIC optimization. X11Perf [21] is also involved in virtual graphic adaptor acceleration. We used *Xentrace* to find the cost of each VM Exit event, including overhead in the hypervisor and dom0. Finally, to get the performance comparison between HVM domain and native with our optimizations applied, we also used some other benchmark tools, such as byte, CPU soak, *SpecJBB,* and *Sysbench,* for overall measurement.

As for our experiment environment, we set up the environment's configuring software and hardware as follows: We ran RHEL4U1 in the service domain, and the HVM domain. The dom0 is configured with two virtual processors and 256M memory, while the HVM domain is using UP kernel configured with 512 M memory and a separate 10 GB partition for its virtual IDE disk. The underlying hardware platform is a Xeon Server with 4 X, 3.8 GHZ Intel® Pentium® processors, which are VT enabled, include an 800 MHZ FSB, installed 4 GB memory, and a 120 GB Seagate SATA disk, and an Intel E100 Ethernet card. Source code used in these experiments is based on xen-unstable.hg [14] early in its development phase, and our optimizations have been pushed into Xen 3.0.

4. Performance Overhead in Xen/VT

In this section, we diagnose the performance overhead of Xen/VT architecture. First, we leverage our enhanced performance toolkits and benchmark tools to collect data to show the main overhead spent in hypervisor for HVM domains. Then, we present our solutions to address the potential performance issues.

Table 1: VM Exit event S/W costs for CPU2000

VM Exit event	Event counts	S/W handling (cycles)	Total handling time
IO_INSTRUCTION	37.90%	158413	90.06%
EXCEPTION_NMI	44.58%	10705	7.16%
EXTERNAL_INT	16.74%	10478	2.63%

Table 2: VM Exit event S/W costs for Kernel Build

VM Exit event	Event counts	S/W handling (cycles)	handling time
IO_INSTRUCTION	14.09%	202286	57.93%
EXCEPTION_NMI	77.19%	24757	38.83%
CR_ACCESS	3.09%	21064	1.32%

4.1 Diagnosing VM Exit handler cost

To get an overview of the total software overhead for handling all kinds of VM Exit events, we leveraged our enhanced *Xentrace* to track all VM Exit events happening when we exercised workload *CPU2000* and Kernel Build. The results are show in Table 1 and

Table 2, respectively. Then, we count all CPU cycles spent in handling each VM Exit event during both experiments, which show the main overheads are from I/O instruction emulation and exception/software interrupt/NMI, which occupy almost 97% of total VM Exit software handling costs.

From the event count point of view, I/O instruction events is less than exception/software interrupt/NMI events, but the average processor handling cost in an I/O emulation takes about 8-16 times that of the guest exception/software interrupt/NMI or host external interrupt. So, the result shows I/O emulation is the critical performance bug. It spends almost 90% of hypervisor time in CPU2000 and 58% of that in Kernel Build case.

4.2 Diagnosing IO cost in Xen/VT architecture

In Section 4.1, we discovered that IO emulation is the critical performance bug to be addressed. In this section, we discuss the experiments we designed to discover the CPU cycles spent in a guest I/O access, by inserting several probes to the sample time stamp of each point. To isolate impact of port emulation time in the device model, we used a dummy I/O port for experimental measurement. In this study, we used I/O port 0x22e, which is not implemented in the virtual platform, and we ran a simple application in HVM guest to circularly read this port. Besides that, we also disabled TSC virtualization in the HVM guest (that is, guest TSC read get the host TSC directly by enabling "use TSC offsetting" [3] in processor-based VM-execution controls and cleared "TSC offset"[3] VM-execution control), so that TSC timestamps sampled in dom0, hypervisor, and HVM guest came from the same hardware time source, and made the experimental result more precise.

To measure them in a direct way, we define the following probes in HVM guest, hypervior, and dom0:

(1) Before HVM guest read the I/O port;
(2) Enter VM Exit handler;
(3) HVM domain is scheduled out;
(4) Hypervisor returns to Domain;
(5) Begin dom0 evtchn upcall handler;
(6) End of dom0 evtchn upcall handler;
(7) Qemu process wakes up;
(8) Qemu emulation done;
(9) HVM guest is scheduled in;
(10) After HVM guest, I/O read completion.

We then classified the time cost into these five categories:

1. Hypervisor processing (1->3 and 9->10)
2. Domain switch to dom0 (3->6)
3. Process switch in dom0 (6->7)
4. Hypervisor returns to dom0 (7->8)
5. Domain switch back to HVM domain (8->9)

Table 3: Dummy I/O S/W handling cost breakdowns

	Cycles	Percentage
Hypervisor processing	16745	32.58%
Domain switch to dom0	4036	7.85%
Process switch in dom0	13051	25.39%
Device model	8183	15.92%
Switch back to HVM domain	9382	18.25%

Table 3 illustrates that the device model itself spends only about one sixth of the total time (51.4K), but the process switch in dom0, spending more cycles, is one of the key issues impacting performance. Actually, the time spent in domain switch back to HVM domain (that is, dom0 gives up processor quantum after the completion of I/O emulation), stems mainly from process switch in dom0 from device model to idle process, because it blocks dom0 through the hypercall when idle process is scheduled. So, in total, the process switch in dom0 is the key performance issue, and provides us room to improve I/O subsystem performance.

5. Xen/VT Performance Optimization

In this section, we discuss possible solutions to address the performance bugs we found. First, because IO emulation is very expensive, we moved PIC device model to hypervisor, and this reduced the cost for vast PIC port accesses. Second, to reduce unnecessary wait time when HVM guests perform DMA operations, we extended Qemu's IDE emulation model using multi-thread solution. Then, to obtain good performance for vNIC, we also introduced an event-driven mechanism to replace Qemu's native time-lapsed polling method to asynchronously wake up Qemu, when network data packets reach it. In addition, to enhance X system, a shared virtual video memory approach was employed to assist virtual video graphic card emulation.

5.1 PIC virtualization Optimization

Table 4: PIC I/O ports handling costs in all IO_Exit events within cpu2000: PIC I/O ports is the dominant (97.78%) performance bottleneck

IO ports	Total handling time	Event counts	Events per millisecond
0x20	10.14%	24.41%	1005
0x21	84.38%	72.99%	3006
0xa0	0.03%	0.08%	3.25
0xa1	0.12%	0.30%	12.51

From our previous experiments, we know I/O emulation is the key performance issue, especially for compute-intensive applications. We then profile all I/O port accessing events using *Xentrace* in the same environment as that in Section 4.1. The results shown in

Table 4 are really surprising. PIC ports, including master PIC (0x20 and 0x21) and slave PIC (0xa0 and 0xa1), take 97.78% of total I/O emulation cost, which means 88% of the total hypervisor cost.

5.1.1 Optimization Method

We suspect I/O emulation cost is mostly introduced by guest PIT timer interrupt handler that ticks at 1000 HZ in Linux 2.6.9 kernel. Statistical data in

Table 4 shows that PIC port 0x20 is accessed 1005 times per second and port 0x21 is accessed 3006 times per second, which obviously indicates that the PIC I/O access is mostly caused by guest Linux PIT interrupt handler, which access port 0x21 3 times and port 0x20 1 time per interrupt. This is exactly what Linux 2.6 does today where IMR (Interrupt Mask Register, 0x21) is dummy read and then written in mask_and_ack_8259A and re-written in

later end_8259A_irq, while port 0x20 is written only in mask_and_ack_8259A for EOI.

In hardware virtualization, we are unable to reduce the total number of guest PIC I/O accesses, but we can reduce the average cost of each PIC I/O access. In this case, because PIC is so critical to the virtual platform performance, we tried to move PIC device model from Qemu to hypervisor, estimating that the restructuring work would decrease the average cost to one tenth from 60-200K cycles to about 4-5K cycles, which is the cost of the simplest VM Exit handling, such as CPUID.

This restructuring effort was achieved by sharing virtual interrupt wire between device model and hypervisor. Devices in device model set or clear a virtual interrupt by asserting or de-asserting the wire, which in turn triggers hypervisor to transfer the wire status to hypervisor every time when resuming an HVM guest. Status changes are sent through an event channel, between the virtual wire and target processor.

5.1.2 Optimization Result

With this optimization, we clearly see that about 7%-14% performance enhancement as shown in Table 5 is achieved. But, in the meantime, we also see its side effects, because the wakeup frequency of the device model is reduced significantly from previously 4 times per millisecond to probably 1 time per 100 millisecond. So it may slow down the virtual network card performance.

Table 5: Performance enhancement with moving virtual PIC to hypervisor

Benchmark tool	Performance enhancement
CPU2000	6.58%
Kernel Build	14.44%
Cycle Soak	12.07%

In summary, because the virtual PIC emulation code device model of dom0 consumes a lot of CPU cycles, it is a must to move virtual PIC to hypervisor, and we also get 7-14% performance gain, as a whole. Although one side effect, such as virtual NIC performance slowdown, is raised.

5.2 IDE disk DMA Optimization

With virtual PIC in hypervisor, we achieve almost 93%-95% performance of native system in CPU2000. We then move our focus to device side performance and choose kernel build as the benchmark to look for the next performance bottleneck.

We used *Xentrace* again to track the access of all of the various I/O ports, and their handling time. Table 6 shows a great decrease in the handling time of the PIC IMR port, becoming only 2.67% of all I/O ports, while Bus Master IDE Command Register (BMICR) (that is, port 0xc000), occupies 87% that dominates the I/O cost, due to the extremely long average handling time.

Table 6: I/O handling costs with kernel build after virtual PIC was moved to hypervisor: Bus Master IDE Command Register (0xc000) dominant I/O handling cost

I/O Port	Handling time	Access number	Average cost (cycles)
0xc000	62.61%	2.52%	2775696
0x0064	22.78%	0.40%	6340282
0xc002	4.50%	2.52%	199403
0x01f6	3.36%	1.01%	372690
0x0021	2.67%	61.96%	4499

In addition to the average handling time of BMICR, we did another detailed study to determine the distribution of handling time in different access. Table 7 shows that read access looks normal, both in average handling cost and its distributions, but write access costs too much under some situations (24% of write access takes more than 5 M cycles), and finally it causes the average handling time to surge.

Table 7: Bus Master IDE Command Register access handling costs statistics: write operation sometimes spends huge cycles.

	<50K	50K-500K	500K - 5M	>5M	Average cost(cyc)
read	60.74%	38.78%	0.19%	0.29%	66367
write	32.99%	30.61%	12.24%	24.15%	4581915

5.2.1 Optimization Method

According to Table 7, at least one third of write access processing costs are similar to the typical cost of I/O emulation (<50 K cycles) such as dummy I/O port, but in the extreme cases (24%), it can take huge processing cycles (>5 M cycles). This likely implies that device model may trigger great effort to complete a long task at a particular condition of BMICR write. This is proven when checking the QEMU device model code that emulates BMICR write access, which may start DMA emulation when certain conditions are met.

The IDE DMA emulation will, in turn, read or write the block device to complete emulation. The Qemu code is frequently used in Xen, while it is still a single thread approach and has to wait till the completion of data write or read to/from the real physical device. In this case, it needs a long time and blocks the guest processor to schedule other tasks. We improved the IDE device model to be concurrent with DMA emulation by creating a second thread to wait for the completion of physical device operation, and signal interrupt to guest once the underlying native device driver completes the operation. Thus, the main thread receiving the guest I/O request can return directly, without waiting for the completion of DMA operation, which means the guest processor is no

longer blocked, and get CPU cycles when the physical device is doing disk read or write.

5.2.2 Optimization Result

To evaluate multi-thread IDE approach, we used Kernel Build, CP, tar, and hdparm Linux commands as benchmark tools. As a result, the concurrent DMA emulation approach improves performance 8-48% in various benchmark tools as shown in Table 8, and it can be extended to generate multiple auxiliary threads bases on the number of guest IDE controller, which may help SMP guests with multiple IDE controllers.

Table8: Performance gain with concurrent ide DMA model

	Kernel Build	Cp -a	tar xjf	hdparm
Perf gain	8%	30%	47%	48%

5.3 Virtual NIC card virtualization

There are two network device emulation drivers in the device model, one is Ne2000 and the other is PCNet[3]. The Ne2000 NIC can work either as an ISA device or a PCI device, while Pcnet is a PCI device. In our test, we found the network performance was so low that only about 174-290 KB/s can be gained with virtual Pcnet, and similar result in NE2000. Comparing with IDE disk, which is also emulated in device model, the performance regression of the virtual network device to a native system is much greater. With moving virtual PIC to hypervisor, we found that network performance became even lower, to about only 30-174 KB/s. By looking at how the NIC device model is emulated, we found it was very likely to be caused by the time-lapsed polling driven mechanism in Qemu, which can be woken up from at least 4 times per millisecond to 1 time per 100 milliseconds.

5.3.1 Optimization Method

We then performed an experiment by creating a separate thread, which was woken up every 5 milliseconds, to poll virtual NIC in and out packets status, instead of 1 time per 100 milliseconds in the main thread. The result showed great improvement, up to 2.2-3.5 MB/s bandwidth, in same experiment environment, which proved our previous suspicion. To reduce the overhead, caused by additional thread that introduces performance regression for compute-intensive benchmark, and also to avoid the complexity of communication between threads, we developed an event-driven solution that allows Qemu to wake up whenever a network packet is sent or received, by making Qemu sleep on a network device file descriptor (tun or tap device4 in dom0). Therefore, in this condition, dom0 would wake up Qemu thread in time to handle possible packets in or out, rather than depend on a time-lapsed polling mechanism.

Table 9: Scp performance in different cases: Event driven NIC model gets great performance with less complexity.

SCP	Original	vPIC in HV	Polling per 5ms	Event driven
To guest	174 KB/s	30 KB/s	3.5 MB/s	4-8 MB/s
From guest	290 KB/s	148 KB/s	2.2 MB/s	2-4 MB/s

5.3.2 Optimization Result

In this section, we evaluate performance gain with an event-driven mechanism. Since performance gain is expected to be large, we don't need to consider the accuracy issue introduced by the benchmark tool at this stage. We used a simple Linux tool, "scp", to copy data to or from the HVM guest to see the transfer speed in both situations. To get a more clear-cut performance picture, we measured the side impact after moving PIC to hypervisor as well. As shown in Table 9, although virtual PIC in HV causes some performance loss compared with Original, both event-driver mechanism and "polling per 5 ms" methods result in great performance gains. Ultimately, considering the factors of code complexity and CPU use, we chose to adopt event-driven mechanism.

5.4 Virtual Video Memory Optimization

For graphic adaptor support, Qemu has two options, one is a standard VGA adapter and the other is cirrus logic GD5446 PCI VGA adapter. The latter one supports high resolutions of 1024x768, 24 bit color, so that an X window system can run efficiently with it.

From the first trial of running an X system as a guest, performance was unacceptably slow. Referring to the analysis in Section 4.2, we realized that a possible reason was that high overhead is introduced when drawing pixels, because drawing a pixel is done as a write operation to video memory, as follows:

1. Video memory (VM) access;
2. VM Exit caused by page fault;
3. Send I/O request to device model;
4. Domain switch to dom0;
5. Process switch to device model;
6. Device model handles the request

The whole process costs a lot of CPU cycles, similar to I/O emulation, which is about 51.4 K cycles. For X system, due to its high resolution, worst case is that millions of IO operations are required. So, it is a big bottleneck to run X application with the existing model.

5.4.1 Optimization Method

After carefully analyzing the VGA MMIO mechanism, we discovered that MMIO operations only write data to MMIO address space, and no read access is involved.

[3] Up to Xen 3.0, Ne2000 and Pcnet NIC card are supported.

[4] Tap and tun are used to implement NIC virtualization in Linux.

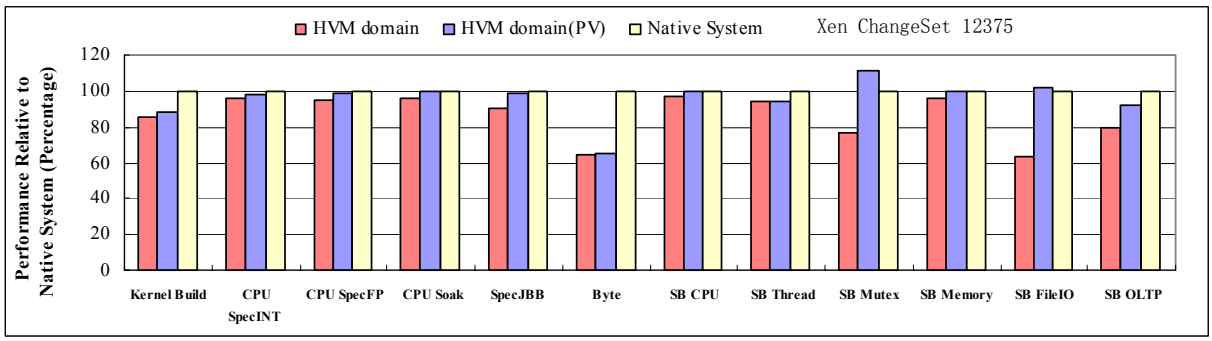

Figure 2. Performance comparison of HVM Domain and Native system with Xen #Cset 12375. To demonstrate our optimizations, we chose an old Change Set source for performance benchmarking. The current Xen/VT project has better performance than that showed in this figure. In addition, we assume the native system has 100% performance relative to itself.

To optimize guest video memory operation, we briefly let the guest access the memory directly, by shared virtual video memory (SVVM) for the graphic adaptor memory, filling the guest physical-to-machine mapping table when the virtual PCI device assigns an MMIO (memory mapped I/O) address to the adaptor. When the guest accesses the video memory page for the first time, a VM Exit is triggered due to the page fault, but then the shadow mechanism will set up a shadow page for the page, so there are no further page faults. As for Qemu VGA display, it updates the screen with SVMM at a fixed frequency, and makes display effect smooth.

5.4.2 Optimization Result

Table 10: Performance comparison of 2 typical X applications: SVVM (sharing virtual video memory) improves performance dramatically

	Original	SVVM	Service OS(dom0)
time of startx	130s	28s	29s
time of ls –l in xterm	40s	1.6s	<1s

Table 11: X11Perf benchmark result: SVVM (sharing virtual video memory) improves performance dramatically

	Original	**SVVM.**	**Service OS(dom0)**
-ellipse500	9.5/s	71300/s	130000/s
-oddstrap10	306/s	29000/s	64400/s
-copywinpix10	319/s	33000/s	520000/s
-putimagexy10	5.9/s	786/s	32100/s

To evaluate the SVVM algorithm we proposed, we designed two experiments to diagnose performance gain. One contrasted the total time lapse of startx in two situations, and the other one compared benchmark data with X11Perf. Table 10 and Table 11 demonstrate the performance data sampled with the guest running a Redhat FC3, with 256 M memory. According to the result, there are obviously great performance gains with SVVM, and the user experience actually becomes comfortable.

6. Related works

Several previous studies [2, 6, 18, 19, 20] have documented performance analysis and tuning on Xen projects. *Menon* et al.[2] use their performance toolkits to identify performance issues on network applications in Xen, and then they extended their work to optimize network virtualization in [6], and improve the network performance of guest domains by 4.4. Also, *Willmann et al*. [18] propose the CDNA architecture for network virtualization in Xen, to improve the performance of networking virtualization with better scalability. *Liu et al*.[19] obtains high performance I/O virtualization by removing the hypervisor and the driver domain from the normal I/O critical path. However, it can't address the scalability issues, since it heavily relies on the intelligence provided by the hardware device. In research [20], *L.Cherkasova* proposed a light-weight monitoring system for measuring the CPU usage of different virtual machines which can include the overhead in the device driver domain caused by I/O processing, on behalf of a particular virtual machine, and they also tried to quantify and analyze overhead for a set of I/O intensive workloads with their system. These research achievements are totally based on the para-virtualized environment in Xen, and try to analyze and identify performance issues in special components of Xen. And they subconsciously enlighten us during our optimizing Xen/VT project as well. Our paper differs in that we focus on investigating and optimizing overall system performance in a full-virtualized environment in the process of extending Xen, using Intel® Virtualization Technology.

7. Conclusions

In this paper, first of all, we briefly presented an architectural overview of the Xen/VT project, and we demonstrated our extensions to profiling tools that

support HVM domains. Subsequently, we specifically demonstrate our methodology, analysis, and results for various optimizations related to PIC, IDE, NIC, and VGA virtualization. To evaluate overall performance gains in Xen, we employed many benchmark tools, such as *Kernel Build, CPU2000, CPU soak, byte, Specjbb2005,* and *Sysbench* to contrast, in the environment established in Section 3, the performance of the HVM Domain and the HVM domain with para-virtualized driver [14]. With these optimizations, as demonstrated in Figure 2, HVM domain achieves around 80%-95% system performance comparable with the native system. In addition, if we use a para-virtualized driver for disk and network IO (HVM domain (PV) in Figure 2), it achieves performance very closely to the native system. We can believe that hardware-assisted full virtualization should demonstrate more and more excellent performance with Intel's future technologies which are targeted to enhance VT at the fields of processor and devices virtualization.

8. Acknowledgement

We are grateful to other contributors for performance tuning work of the Xen/VT project. We would like to thank Asit.K.Mallick, Jun. Nakajima, and other team colleagues for their outstanding performance on tuning work, and thank Wilfred Yu, Fleming Feng and Susie Li for their excellent leadership on the Xen/VT project. Aso, Ian.Pratt and Keir Fraser from Cambridge University provided excellent insights and suggestions to the tuning effort. We are grateful to Noel C. Arnold and other reviewers for their valuable comments. In addition, Qi Li, from Tsinghua University also provided some valuable suggestions to improve this paper. Especially, we also thank Sihan Qing and Huanguo Zhang of Wuhan University for their helpful comments and guidance.

9. References

[1] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. "Xen 3.0 and the Art of Virtualization," *Ottawa Linux Symposium*, Volume 2, 2005

[2] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel, "Diagnosing Performance: Overheads in the Xen Virtual Machine Environment," *1st ACM/USENIX Conference on Virtual Execution Environments* (VEE'05), June 2005

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

[4] F. Bellard. QEMU, "a Fast and Portable Dynamic Translator," *Proc. of USENIX Annual Technical Conference 2005 (FREENIXTrack)*, July 2005

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *Proceedings of ACM Symposium on Operating Systems Principles*, 2003

[6] A. Menon, A. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," *USENIX Conference*, June 2006

[7] J. Nakajima, A, Mallick, I. Pratt, and K. Fraser, "X86-64 XenLinux: Architecture, Implementation, and Optimizations," *Ottawa Linux Symposium*, 2006

[8] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A.V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel Virtualization Technology," *IEEE Computer* Volume 38, Issue 5, pp. 48–56, May 2005.

[9] R. Creasy, "The Origin of the VM/370 time-sharing system," *IBM Systems Research Journal*, 1981

[10] Oprofile. http://oprofile.sourceforge.net

[11] VMware Inc. VMware virtual machine technology. http://www.vmware.com/.

[12] Qumranet Inc. KVM: Kernel-based Virtual Machine. http://kvm.sourceforge.net/.

[13] Y. Dong, S. Li, A. Mallick, J. Nakajima, K. Tian, X. Xu, F. Yang, and W. Yu, "Extending Xen with Intel Virtualization Technology," *Intel Technology Journal* , 10(3), 2006.

[14] http://xenbits.xensource.com/xen-unstable.hg

[15] Microsoft Corporation. http://www.microsoft.com

[16] Innotek GmbH Inc.http://www.virtualbox.org

[17] K. Adams, O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, California, USA, 21–25 October 2006

[18] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel, "Concurrent Direct Network Access for Virtual Machine Monitors," *The International Symposium on High Performance Computer Architecture* Phoenix, AZ, February, 2007.

[19] J. Liu, W. Huang, B. Abali, and DK Panda, "High Performance VMM-Bypass I/O in Virtual Machines," *USENIX Annual Technical Conference,* June 2006.

[20] L. Cherkasova and R. Gardner, "Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor," Proceedings of th*e USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference,* April 2005, Anaheim, CA, USA

[21] X11Perf. http://www.xfree86.org/4.2.0/X11Perf.1.html