M.S. THESIS

Enhancing the I/O System for Virtual Machines Using High Performance SSDs

SSD를 사용하는 가상머신 환경에서 I/O 성능향상 방법

FEBRUARY 2015

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Myoungwon Oh

Enhancing the I/O System for Virtual Machines Using
High Performance SSDs

SSD를 사용하는 가상머신 환경에서 I/O 성능향상 방법

지도교수 염헌영

이 논문을 공학석사학위논문으로 제출함

2014 년 10 월

서울대학교 대학원

전기.컴퓨터 공학부

오명원

오명원의 석사학위논문을 인준함

2014 년 12 월

위 원 장	이광근	(인)
부위원장	염헌영	(인)
위 원	엄현상	(인)

Abstract

Enhancing the I/O System for Virtual Machines Using High Performance SSDs

Myoungwon Oh
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

Storage I/O in VM (Virtual Machine) environments, which requires low latency, becomes problematic as the fast storage such as SSDs (Solid-State Drives) is currently in use. The low performance problem in the VM environment is caused by 1) the presence of additional software layer such as guest OS, 2) context switching between VM and host OS, and 3) scheduling delay for I/O process. These factors do not cause serious problems in the case of using HDD which leads to high latency batching. However, there will be significant performance degradation when fast storage devices are used. To address this problem, we have proposed the following methods to improve the performance of I/O stack in the VM environments by attempting to optimize the I/O stack: one is pipelined polling, and the other is multiple issues and multiple completions. We have found via experiments that our approach leads to increases in the performance of SSDs in a VM environment by up to 50% when multiple VM storage devices are used, and that it leads to improvements in the performance by more than 80% when a single VM storage device is used, with the CPU utilization reduced by up to 25%.

Keywords: SNU, Computer Science Engineering, thesis

Student Number: 2013-20825

Contents

Abstract **i**

Contents **iii**

List of Figures **v**

List of Tables **vii**

Chapter 1 Introduction **1**

Chapter 2 Problem definition **5**

2.1 Design and implementation 9

2.1.1 pipelined polling 9

2.1.2 multiple issues and multiple completions 12

2.1.3 NUMA awareness 15

2.2 evaluation 15

2.2.1 SSDs 16

2.2.2 Ramdisk 18

2.2.3 NVMe device 20

2.3 related work 24

Chapter 3 Conculsion **26**

Bibliography **28**

요약 **31**

Acknowledgements **32**

List of Figures

Figure 1.1 I/O system of KVM 2

Figure 2.1 IOPS for SSDs in direct access test (fio, direct, 4K random read & 32 processes per device) 6

Figure 2.2 IOPS for SSDs in direct access test (fio, direct, 4K random read & 128 processes) 7

Figure 2.3 sequence of pipelined polling 10

Figure 2.4 pipelined polling 10

Figure 2.5 mutiple issues and completions (Q: virtqueue in the virtio blk driver & DT: dedicated thread for completion) 13

Figure 2.6 IOPS for SSDs in direct access test (fio, direct, 4K random read & 32 processes per device) 16

Figure 2.7 CPU usage for SSDs (4 device case shown in Fig 7) 17

Figure 2.8 Performance of single device on ramdisk. (KVM with multi queue patch: default KVM performance with multi queue patch in guest OS.(data-plane + multi queue patch) Full optimization with the number of queues and a dedicated thread: our approach is taken.) a) IOPS in the aio test (fio, direct, 4K random read, iodepth is 32 per process & total 4 processes) b) IOPS in the direct access test (fio, direct, random read & 128 processes) 18
Figure 2.9 Performance of multiple devices on ramdisk a) IOPS in the aio test (fio, direct, 4K random read, iodepth is 32 per process & 4 processes) b) IOPS in the direct access test (fio, direct, random read & 32 processes per device) . 19
Figure 2.10 IOPS for an NVMe SSD in the aio test (fio, direct, 4K random read & iodepth 32 per process) 21
Figure 2.11 IOPS for an NVMe SSD in the direct access test (fio, direct & 4K random read) 22
Figure 2.12 IOPS for NVMe SSD in the direct access test (fio, direct, 4K random read & 128 processes) on single virtual storage 23
Figure 2.13 IOPS for NVMe SSD in the direct access test (fio, direct, 32 threads per device & 4K random read) on multiple virtual storage, our approach is taken except for the number of multiple queues and a dedicated thread to increate the cpu utilization 24

List of Tables

Chapter 1

Introduction

Flash-based fast storage such as SSD shows lower latency and higher bandwidth than the currently existing HDD. In addition to these advantages, the price of SSD has become cheaper, which has made the users have greater interest, and the data storage industry pay more attention to SSD as an alternative to HDD. The datacenter operators are also paying much attention because they need high performance storage such as SSDs, being expected to handle data processing faster by adapting them. The main reason why much attention has been drawn to SSDs is that they have the advantage of fast direct access; therefore they will be beneficial in running I/O intensive workload. For example, Samsung 845DC evo (SATA SSD) and Samsung XS1715 (NVMe[1] SSD), which were used in the experiments of our research explained in this paper, show about 70K IOPS (I/Os per second) and 750K IOPS in physical machine environments. And it is natural that most users expect higher performance when multiple devices are used.

If high performance in using SSDs is not guaranteed in datacenter environments, the operators of datacenters will be skeptical over their purchase and use of SSDs. One of the most important software components in datacenter

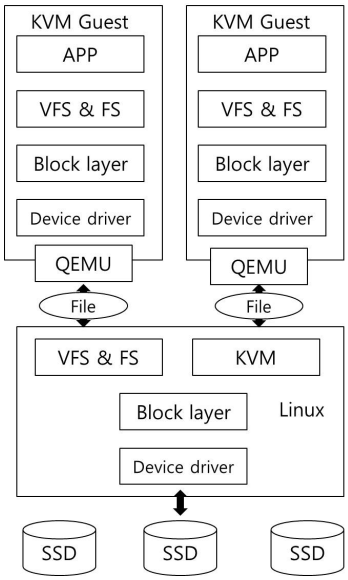

Figure 1.1 I/O system of KVM

environments is VM. A VM is not a physical machine but a structure that is emulated by the host as a software component; there thus inherent overhead exists regarding the use of VM. Specially, the overhead increases in the random I/O situation where it is required to ensure high IOPS, because a lot of communication between host and guest is needed. To address this problem, we have proposed ways to alleviate the bottleneck as much as possible by analyzing the performance in random I/O operations on VMs with SSDs.

Figure 1 shows the stucture of Linux KVM. Here is a brief explanation of I/O flow from guest OS to SSD. When a guest application issues an I/O request, the request is delivered to VFS, the file system in guest kernel, and then finally the virtio[2] block driver through the block layer. In the virtio block driver, the delivered I/O request is queued to the shared internal virtqueue, and then QEMU is notified of the I/O event. At that time, a context switch occurs in QEMU, and QEMU issues an I/O request to the host OS. When the requested I/O is completed, QEMU injects a virtual interrupt to guest OS to notify it

of the I/O completion, and then the guest OS interrupt routine executes to complete the remaining processing. We found out that there is huge overhead incurred due to "exit" between guest OS and host OS, as mentioned in ELI[3] and ELVIS[4], and additional I/O layers.

Our initial study suggests that the performance enhanced by using the current methods may get close to the performance of using one SSD, but when we connect and use multiple devices at the same time, we may experience the problems of performance degradation with SSDs and high CPU utilization. Also, in the case of using a PCI-e or NVM-e SSD, which shows very high IOPS, it may not perform well when it is used as a file based storage device in VM.

We analyzed the root cause of the above mentioned problem of VM performance, and have proposed the following solution to improve the performance of direct access to SSD in VM: First, our approach reduces the cpu utilization and improves the multiple device performance as it removes the "exit" cost and I/O delay during context switch by applying polling and changing the I/O procedure to pipelining. Second, it improves the maximum performance of single device in VM by the method of multiple issue and multiple completion. Third, we have also suggested a NUMA friendly design, and additional optimization methods for fast storage. We have found that our approach may achieve nearly bare-metal performance with lower CPU utilization for SATA SSDs; the performance is improved by up to 50% in the Ramdisk case and by up to 80% in the NVMe case. We will explain our result in more detail in Section 4.

ELVIS[4] and ELI[3] also proposed enhancement methods called an "exitless approach" to eliminate VM context switches, but there are several differences between theirs and ours. 1) Their approaches can be used only in the x86 architecture. They are therefore hardware dependent. In contrast, our proposed approach is the one based on software only, thus supporting all hardware architectures. 2) There is a security problem explained in the papers[3][4] since guest can control IDT(Interrupt Descriptor Table) which requires permission.

Instead, our method does not change any basic logic related to permission. There could be performance issues for host OS because the shadow IDT checks interrupts from the host first on behalf of the host OS. In contrast, in our approach, only the performance of the guest OS is affected. Also, QEMU's data-plane method[5] leading to improvement in the performance of using multiple devices is different from ours for the following reasons: 1) We reduce "exits" in order to increase the performance and 2) lower cpu utilization by decreasing the lock contention which occurs during the execution of "exit," and 3) we use pipelined polling in order to process I/O in parallel with low latency.
Our study was conducted on KVM/QEMU[6] which based on Linux kernel virtualization. Each VM is a process to Linux, and the KVM module and QEMU are in charge of emulating VMs. We have implemented our approach by modifying the block layer and device driver in guest OS and I/O module in QEMU

This paper is organized as follows: Section 2 describes the performance problem and explains the result of evaluating its performance. Section 3 presents our approach and its implementation. We evaluated and analyzed the performance of the implementation in the SSDs, Ramdsik, and NVMe cases; Section 4 explains the result of this evaluation. Section 5 discusses related work, and Section 6 concludes the paper.

Chapter 2

Problem definition

We performed experiments with the fio benchmark for evaluating the performance of random I/O in the KVM environment in order to check the effectiveness of existing methods designed and for improving the I/O performance (Performed direct random read test for 4 SSD). as a result, the following outcomes could be found.

multiple device performance As shown in Figure 2, the data plane from QEMU could achieve relatively good performance for each device when there were multiple virtual storage devices in a single VM (each physical device assigned to each file, which represented a VM storage device; four devices mean that there are four virtual storage devices associated with four files in four different physical devices). However, the performance was degraded by 25% compared to the host OS case. Relatively good performance was achieved in the vhost-blk case compared with other cases, but it was not possible to perform the test if more than three devices were connected. Also, in the case of using KVM, significantly lower performance was shown because all the connected devices shared the I/O thread from QEMU, regardless of other factors, even if

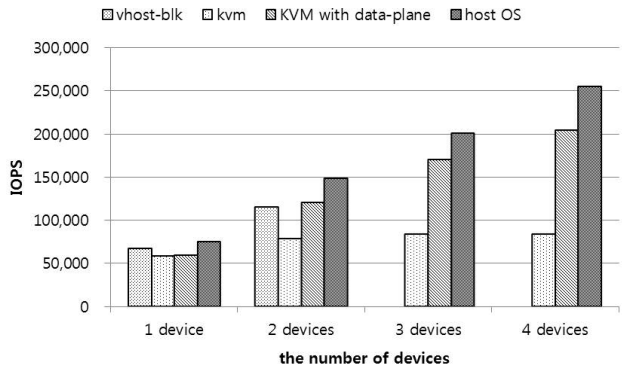

Figure 2.1 IOPS for SSDs in direct access test (fio, direct, 4K random read & 32 processes per device)

the tested VM was supplied with enough CPU power.

single device performance Figure 3 shows the result of the test with four SSDs in RAID0 which performed as a single fast device, being used as a (file based) virtual storage device in the VM environment. Even though over 200K IOPS was achieved in host OS, there was significant performance degradation in the other cases. In particular, constant performance was achieved for each device when multiple devices were used in the data-plane case, but the performance was not affected when a single fast device was used.

CPU utilization As for the CPU usage, almost 1000% CPU utilization was shown in the data-plane cases although the number of vCPUs was six. This confirmed that the CPU was significantly consumed for I/O requests to be issued in a VM and in the real host OS as well. In summary, no existing methods could reach the bare-metal performance for multiple devices and a single one while using the CPU minimally.

Regarding the I/O performance for VM, the first problem is context switching between the guest and host OSes, which is called ”exit.” The previous

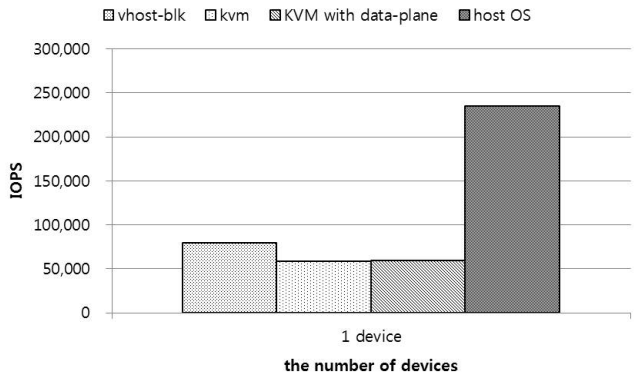

Figure 2.2 IOPS for SSDs in direct access test (fio, direct, 4K random read & 128 processes)

studies[8][3] showed that ”exit” becomes a large portion of the overhead in the VM operation when a switch occurs between the guest and host OSes. When an ”exit” occurs, the system needs to load the guest state into hardware, and resume the host operations, these operations incur the overhead such as cache pollution, delay and serialized I/O because they wait for response of the host. As mentioned in ELI[3], three ”exit” occurs in the KVM implementation. The first ”exit” occurs to notify QEMU of issuing an I/O request. The second ”exit” occurs in order to inject a virtual interrupt for completed I/O, and the third ”exit” occurs due to EOI(end of interrupt) in the VM for accessing the APIC (Advanced Programmable Interrupt Controller). These operations are problematic because not only such an operation itself incurs overhead but also is designed to stop the guest operation and resume the host operation, which hampers efficient parallel execution. In addition, a relatively small number of ”exits” occur in the case of batch operation. On the other hand, an ”exit” cannot be overlooked in the direct access case, which requires quick response. It is thus necessary to alleviate the ”exit” overhead to achieve the bare-metal performance.

The second problem is the scalability one for I/O thread. This problem is due to the internal structure of QEMU that lacks scalability. There is only one I/O thread per VM allocated in KVM and an I/O thread is shared in a single VM. Its capacity is not sufficient to reach the maximum I/O performance for multiple storage devices. Therefore, as it needs more capacity, its I/O performance is limited to the maximum one of a single I/O thread. To address this problem, data-plane[5] is implemented, where an I/O thread is used per device. But it is still problematic in that it does not consider parallelism because a single I/O thread handles both the I/O request and reply. Moreover, it is from QEMU itself, which is taken into account, and it thus has the drawback that the VM overhead is not considered.

The third problem is I/O delay. As shown in Figure 3, the length of I/O flow is almost doubled because there are additional OS and application. This can be solved by removing the unnecessary components in the I/O layer[7], including the deletion of the I/O scheduler in the guest block layer and the submission of guest's bio to the host block layer directly. But this solution has the drawback that the VM is not able to use file based storage since it is necessary to allocate physical device to guest OS directly. The guest application that requests I/O will be scheduled when the I/O is completed in host OS but in the VM environment, I/O delay is much longer than the time of I/O on host OS because it should schedule a vCPU process that is in charge of emulating a physical CPU first then the process scheduler in guest OS schedules the guest application.

The fourth problem is parallelism. As mentioned in splitX[9], Each layer needs to run independently in order to fully utilize the multi-core parallelism but KVM cannot currently operate in this way. In order to maximize the I/O performance, not only multiple vCPUs are needed for ensuring parallelism, but also the I/O layer in KVM/QEMU needs to be changed to permit parallel processing.

The fifth problem is CPU utilization. The current method enhancing the performance of I/O in VM leads to high CPU utilization as we mentioned in Section 2. Even though the CPU utilization is high, the performance gain is not optimal. We thus consider a low CPU utilization and high performance method.

We have addressed the above mentioned problems for file based storage in VM since file based storage is widely used in datacenters even though SR-IOV which is connected directly to VM achieves good performance because it is easy to use software techniques such as migration and copy on write.

Our approach focuses on random read workload for several reasons. First, read is more sensitive regarding the time of response from device than write. And high read performance is required for real workload. Second, our goal is to reduce context switching between guest and host, so we need to exclude other factors that cause performance degradation; for example, there is some overhead incurred by garbage collection or cache effect in the case of random direct write. Third, the I/O flow of write is same as that of read. So if we reduce the flow of read, the write performance should be improved.

2.1 Design and implementation

Our approach, as a software one, is independent of the hardware architecture and is applicable to any architecture. In addition, we consider the performance and efficiency further compared to the conventional methods. We have devised pipelined polling and multiple issue & completion that address the problems explained in the previous section.

2.1.1 pipelined polling

Figure 4 shows the design of pipelined polling. Whenever a guest application requests I/O, the virtio block driver queues this request but does not notify QEMU; therefore, ”exit” does not occur. The request polling thread in QEMU polls the memory space which is shared with guest OS in order to check I/O

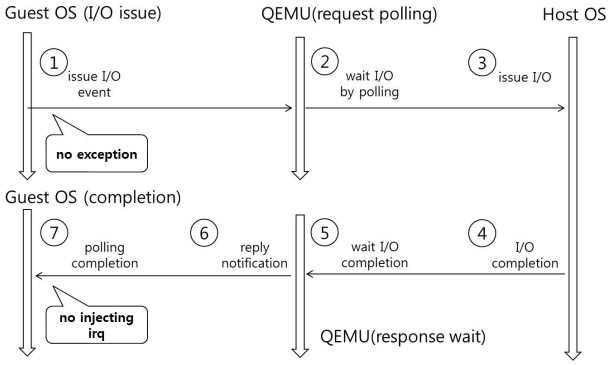

Figure 2.3 sequence of pipelined polling

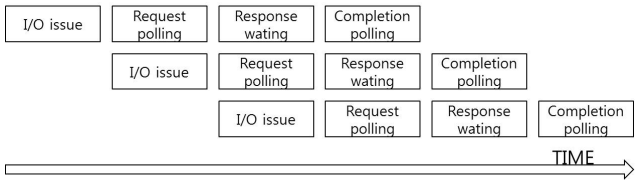

Figure 2.4 pipelined polling

requests from guest. If there are any I/O requests in the queue from the guest, they are sent to host OS, and a response wait thread waits for completion in the blocking state. When the requested I/O is completed, the response wait thread that has waited for the I/O completion wakes up and then notifies the guest of the I/O completion to guest by writing the shared memory area between the guest and host using virtio. Instead of conventional interrupts in quest, the dedicated thread in guest checks the I/O completion by polling, and completes I/O. In this way, each of the modules such as guest I/O request by guest, QEMU request polling by QEMU, response waiting by QEMU, and completion polling by guest works independently; one part does not rely on another, which makes it possible to process them in parallel. Pipelined polling not only gets rid of ”exits” by polling but also leads to high performance.

As shown in Figure 5, the process of direct access I/O is not serialized but

can be performed in a pipelined manner because 1) ”exits” are removed, which makes each module independently work. An ”exit” is needed when notifying I/O or delivering I/O completion without taking an ”exitless” approach. So other I/O modules wait for ”exits.” But if we remove an ”exit,” the I/O module does not need to wait for one. 2) The I/O module is divided independently. So the I/O module can be run in a single thread which means that it can run on multiple CPUs concurrently. It is performed faster not only in the single I/O thread case due to pipelining but also in the multiple I/O thread case because of its parallel structure.

In this design, the method of pipelined polling per device can be used if higher performance is required; otherwise, the method of shared pipelined polling can be used, where each of the modules can handle multiple devices as long as there is no performance degradation, which leads to higher CPU utilization.

We believe that taking a polling based approach is effective in achieving higher multiple device performance and CPU utilization for several reasons. The first reason is the interrupt overhead as found in the previous studies[10][11] mentioned, there will be significant performance degradation due to processing data by interrupt when future storage becomes faster than older storage, and thus if faster I/O is required, the polling as provided via NAPI(New API) in Linux kernel has more benefit than interrupt for I/O intensive workload because polling is able to remove the interrupt overhead and reduce the context switching cost. The second reason is that polling permits removing ”exits,” without being dependent on the hardware architecture. Use of polling requires only monitoring the memory space that is shared between guest and host; in this method, we don’t need any hardware support such as posted-interrupt[12] unless an additional “exit” occurs. In addition, our design could be simply implemented without modifying the existing software since it is based on virtio which is already developed for communication between KVM and QEMU.; all

we need to do is to monitor the shared memory space. The third reason is possibly lower CPU consumption. One of the most significant problems regarding the use of polling is high CPU consumption. Nevertheless, our polling based approach leads to lower CPU utilization than the interrupt based one because it does not incur overhead that is caused by "exits" and reduces lock contention in the KVM module in Linux kernel by avoiding control via APIC. As explained in Section 2, in the data-plane test which was based on six vCPU and four storage devices, nearly 1000% CPU usage was shown, which indicates that about additional 400% CPU usage was shown for the use of data plane since the vCPU capacity was 600% at maximum. We analyzed these observations by using Linux performance monitoring tools (perf), and found that most of CPU was consumed for the execution of the spin_lock function of the KVM module in Linux kernel. We also found that most of the lock contention occurs when making APIC related accesses such as kvm_ioapic_set_irq() and kvm_ioapic_update_eoi(). So if we reduce this contention, we may solve the problem of high CPU utilization problem. We finally thought out that bypassing "exits" in the KVM module is better than modifying the lock structure because modifying the KVM module in order to reduce the lock contention requires modifying the host kernel, and it may reduce the lock contention but not eliminate the contention.

2.1.2 multiple issues and multiple completions

Pipelined polling explained in the previous subsection can increase not only the multiple device performance and CPU utilization but also the single device performance (that of one of virtual storage devices in guest) but this method is not sufficient for a single device such as an NVM-express based SSD which shows more than 700K IOPS. We thus thought both the I/O layer in QEMU and the block layer that includes the device driver in guest OS require overall improvement.

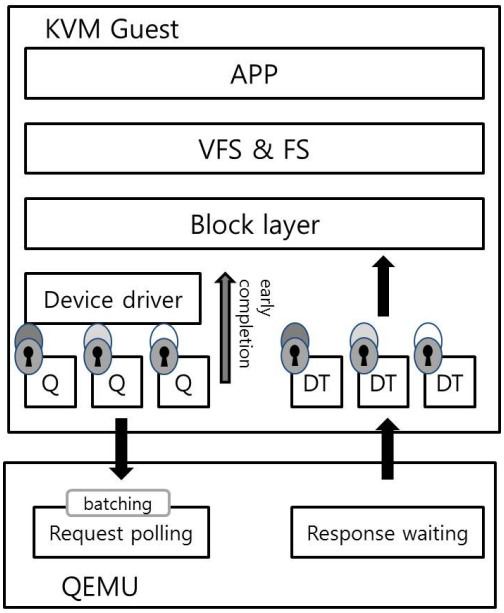

Figure 2.5 mutiple issues and completions (Q: virtqueue in the virtio blk driver & DT: dedicated thread for completion)

We devised multiple issue and multiple completion schemes based on pipelined polling as follows: 1) I/O requests are issued as fast as possible to fully utilize the SSD characteristic. 2) the lock contention is reduced for processing in parallel because the lock contention causes high CPU consumption and poor performance 3) multiple instances of I/O completion are not handled by multiple CPUs as in the MSI-x interrupt, but with CPU affinity considered, the dedicated thread instead of interrupt is used to reduce the latency.

We made our design by considering the following factors as shown in Figure 6: 1) as the result of our analysis of the virtio block driver in guest, there is one queue in charge of issuing I/O requests to host. And the guest driver layer that is based on a single queue and single interrupt can reach only the I/O performance for a single CPU even if VFS and the block layer in guest OS can process I/O in parallel. It is therefore necessary to change the current scheme to the multiple issue and completion based one. In addition, 2) we added a function called "early completion." The early completion means checking the I/O completion in the issue context whenever issuing an I/O request, and processing it if there is any completed request. Basically, a lock should be held when the guest issues an I/O request. It is thus more efficient to process it in the issue context if there is any completed request because this will reduce the overhead incurred by the lock contention and completion thread. 3) A lock per queue is used in order to process I/O requests in parallel when the guest issues them and I/O is completed. Storage devices shown at the user level operate based on request queues allocated on a per device basis in Linux. The performance may be degraded in the case of high IOPS and parallelism because a queue lock should be held when the guest issues I/O requests and has them completed I/O through this queue. Therefore, the guest should be changed to have the parallel structure as suggested in [13] in order to reduce the dependency on lock and permit parallel processing. 4) The "exit" overhead is reduced via batch processing batching based on a threshold value when QEMU checks an I/O

request from guest. As for the structure of SSD, it consists of a number of channels. An SSD can handle multiple I/O requests concurrently. In order to take full advantage of this characteristic, it is important not only to issue I/O requests as fast as possible but also to issue multiple I/O requests at one time to maximize the SSD performance with reduced context switching from host OS. 5) The completion polling thread is modified by adding a peek function to avoid holding any unnecessary lock during the polling process unless there is any completed request in queue.

2.1.3 NUMA awareness

In our design and implementation, the overall I/O layer is divided into two parts: one is host I/O, and the other is QEMU I/O. They are pinned to different NUMA nodes. Because of this, unnecessary switching between NUMA nodes is limited to one time. The CPU affinity of VM, I/O thread or completion thread is very important in the NUMA architecture because different amounts of overhead are incurred depending on the location of memory node.[15] This overhead is more important in the case of SSD that shows high IOPS because the system cannot be fully utilized due to the overhead such as that of context switching or memory copy operation. According to the result of our experiment, vCPU and I/O thread of QEMU switch their operating CPU depending on the system load. Therefore, this may cause problems regarding cache locality and delay.

2.2 evaluation

Our approach of enhancing the I/O systems for VMs was evaluated in the following environment: An Inter Xeon(R) E5-2690 with 2.90GHz 2 CPUs (with 16 cores each) and 128G RAM was used. Host OS was ubuntu 12.04, The Linux kernel version was 3.2.0, guest os kernel versions were 3.5.0 rc7 & 3.15.0, QEMU version was 1.6.2. the number of vCPUs was six with a single VM based on each,

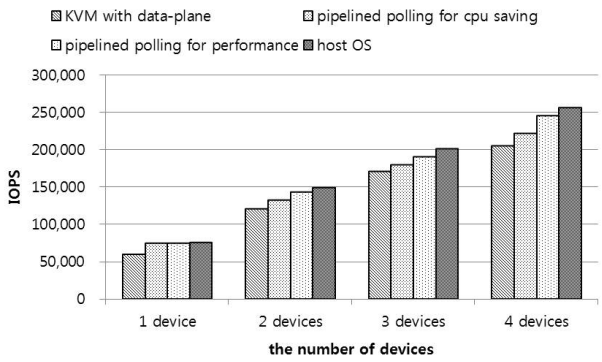

Figure 2.6 IOPS for SSDs in direct access test (fio, direct, 4K random read & 32 processes per device)

and the ext4 file system was used both host and guest OSes.

2.2.1 SSDs

The tested SSD is Samsung 845DC evo, the capacity of which is 960G, SATA3. It was tested with LSI megaraid 9361-8i

multiple device performance and cpu usage

There are four test results shown in Figure 7. The first one is that of enhancing the data plane which leads to the best performance among the existing solutions. The second one shows the performance of pipelined polling, a high performance version without CPU saving. The third one shows the performance of pipelined polling with CPU saving. The fourth one shows the ideal performance of host OS.

multiple device performance: In the CPU saving test, the performance was measured using a request polling thread in QEMU allocated for two devices, which means that two devices share a single request polling thread. In the case of the high performance version, request polling thread and reply polling thread

	max cpu utilization	min cpu utilization
data-plane	1000%	950%
piplelined polling (performance)	850%	820%
piplelined polling (cpu saving)	750%	710%

Figure 2.7 CPU usage for SSDs (4 device case shown in Fig 7)

were allocated for each of the devices. As a result, the performance version performed almost the same (240K IOPS) compared with the host OS case (250K IOPS), and the performance of the CPU saving version was improved nearly by 10% compared with the previous data-plane case.

maximum cpu usage: As mentioned in Section 2, up to 1000% CPU usage was measured for random I/O in the data-plane case. But the CPU saving version led to only 750% CPU usage, which means nearly 150% CPU usage was required to perform I/O processing. Because the number of vCPUs is six, the maximum cpu usage of a single VM is 600%. The high performance version also showed 850% CPU usage, which indicates that the reduced percentages in CPU usage were 25% and 15% for the implementations (Figure 8).

We expect that this improvement will be better if the system is in a more congested situation where a lot of devices are connected, because the lock contention is reduced significantly in the KVM module. Also, regarding to high CPU consumption which is one of the problems regarding polling, the response waiting thread in QEMU consumes only the CPU when the CPU computation is needed because it waits for an I/O event in the blocking state.
The request polling thread checks I/O requests from guest OS periodically. So it could need to consume CPU more than any exiting solution. However it uses busy polling if there are many I/O requests; otherwise, it get into the sleep state when the I/O queue in the request thread is empty. Thus it does not incur high overhead. The completion polling thread in guest checks completion periodically using a peek function, but we changed this scheme so that the completion

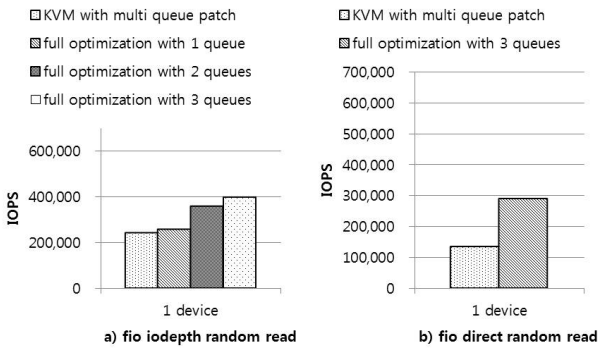

Figure 2.8 Performance of single device on ramdisk. (KVM with multi queue patch: default KVM performance with multi queue patch in guest OS.(data-plane + multi queue patch) Full optimization with the number of queues and a dedicated thread: our approach is taken.) a) IOPS in the aio test (fio, direct, 4K random read, iodepth is 32 per process & total 4 processes) b) IOPS in the direct access test (fio, direct, random read & 128 processes)

polling thread could become active only when receiving an I/O request to avoid high overhead.

2.2.2 Ramdisk

single device performance and maximum performance

A test was carried out based on a ramdisk to check the maximum performance in the case of using single fast storage in VM.

First, Our approach led to much higher performance than any existing approach by applying pipelined polling for maximum performance. The performance for a single device mentioned in Section 2 (65K IOPS) was improved up to 140K IOPS. But this was still less than that in the RAID0 case (about 250K) where four SSD devices were connected or in the case of using a single NVM-express device. We thought that this problem was caused due to lack of

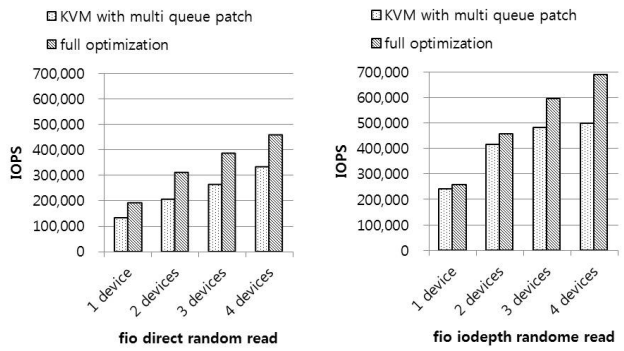

Figure 2.9 Performance of multiple devices on ramdisk a) IOPS in the aio test (fio, direct, 4K random read, iodepth is 32 per process & 4 processes) b) IOPS in the direct access test (fio, direct, random read & 32 processes per device)

scalability in the block layer in guest OS. As result of applying optimization, we got almost same result compared with the case that only applied pipelined polling. We thought that this problem is caused by scalability in block layer in guest OS. We found that the I/O layer in guest except for the device driver cannot currently issue I/O requests as fast as possible even if fast processing is required in order to achieve the maximum hardware performance. The implementation remains as future work.

There is another approach (Linux multi-queue) that solves the problem of scalability in the Linux block layer. We thus tested it in our guest OS version 3.15 to confirm performance improvement without incurring the guest block layer overhead. As a result, the base performance was improved achieving up to 150K IOPS. When pipelined polling was applied, it reached up to 250K IOPS. When multiple issue and completion plus pipelined polling were employed, it reached up to 280K IOPS. But this was not a linear result which we wanted.

We performed experiments after changing the fio benchmark options such as iodepth to 32 from 1 and the number of process to 4 from 128 because we believed that a lot of threads would incur context switching overhead in pro-

cess scheduling. As shown in Figure 9, multiple issue and completion improved the I/O performance by more than 50% compared to the baseline one after changing the fio benchmark options. But there was limitation as shown in Figure 9; although more queues and completion threads were added, we could not achieve over 400K IOPS, which is not the maximum performance for a single VM because multiple devices can lead to more than 600K IOPS and the ideal performance is 1400K according to our simulation. It thus seems that it is a problem regarding the Linux block layer. If this limitation is overcome, it is possible to make better performance improvement.

multiple device performance and cpu utilization

Figure 10 shows the result of applying our approach compared with the data plane with multi-queue; in our approach, only the multi queue and completion thread are excluded and the multiple issue and completion function is used. As seen in the figure, higher performance gain was obtained when an additional device was connected. Also the performance is enhanced in the iodepth test where less ”exits” occur.

In the case of using the existing method, more ”exits” and lock contention occur as more devices are connected due to accessing APIC and performing internal KVM operations in the KVM module compared to our approach. But our approach does not cause this problem by performing polling. The performance improvement is this much greater when more devices are attached. 50-100% more CPU is consumed than any existing approach because busy polling is used for utilizing the disk bandwidth fully.

2.2.3 NVMe device

This test was performed with a Samsung NVMe SSD 800GB named XS1715.

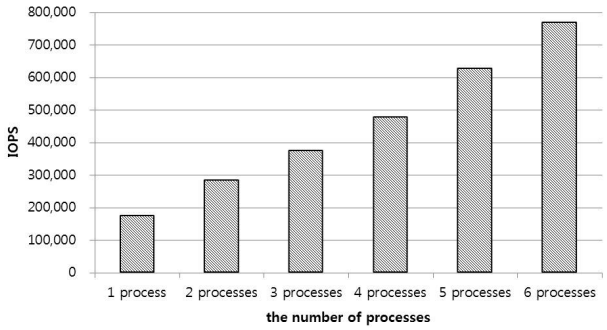

Figure 2.10 IOPS for an NVMe SSD in the aio test (fio, direct, 4K random read & iodepth 32 per process)

NVMe random read performance

We tested the NVMe device on host OS for checking the maximum performance. As seen in Figures 11 and 12, we were not able to achieve the maximum NVMe performance in the case of a single thread or a small number of iodepth. In the case of direct access, 128 threads should be used for achieving the maximum performance of device, and in the iodepth test, six threads should be used to reach the maximum performance. As mentioned above, in the random I/O case, the most important thing is that serialized I/O cannot lead to the achievement of the maximum NVMe device performance. In order to reach the maximum NVMe performance, parallel I/O is needed.

If we want to fully utilize an NVM-e SSD, we should use more than six I/Os with 32 as iodepth or 96 I/O threads. But it is not efficient if we add this thread to each device that is attached to a VM. We thus need to use dynamic allocation of I/O threads or a CPU saving method such as ours as explained in Section 2.

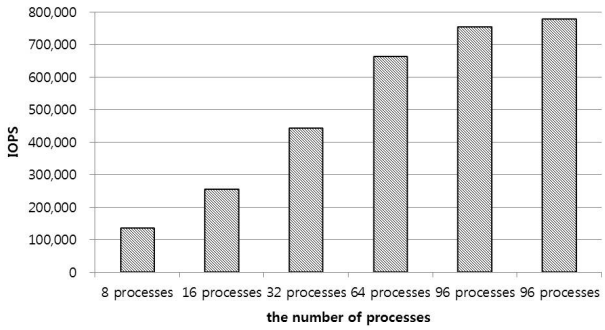

Figure 2.11 IOPS for an NVMe SSD in the direct access test (fio, direct & 4K random read)

single device performance

Testing was conducted in order to identify performance of a single virtual storage device in guest OS regarding the NVMe device.

Figure 13 shows the result of taking our approach and using the current technologies in the case of fio. Each result in the graph includes that in the left; for example, the result of pipelined polling means that of numa awareness plus pipelined polling. The result of the iodepth test was slightly better than that of direct access test. But the overall tendency was similar. The original result shown in Figure 13 indicates that about 95K IOPS was obtained without any optimization performed. The NUMA awareness test was performed with vCPU pinning and memory batching with consideration of NUMA nodes, and 135K IOPS was obtained. The pipelined polling result was 200K IOPS which was obtained while numa awareness was applied. Multiple issue and completion with three queues led to 250K IOPS, but it was not linear improvement as we had expected. The result of applying early completion and other optimization shows improvement but does not show a big difference because I/O was faster when the null test was performed, without causing congestion right, and thus early completion made no big impact. In the null test, QEMU returns in I/O

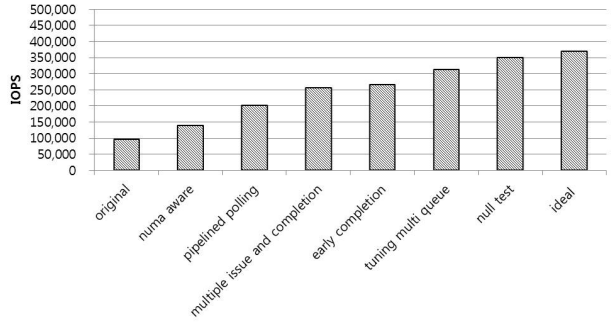

Figure 2.12 IOPS for NVMe SSD in the direct access test (fio, direct, 4K random read & 128 processes) on single virtual storage

immediately with null data. This means that the other processing in host is eliminated. There was no big performance improvement even though multiple issue and completion can handle three I/O requests concurrently, because there are three queues and a completion thread. We expect that this phenomenon happens because guest OS cannot issue I/O requests fast enough. The ideal result is the same as that of three threads and 32 iodepth with the NVM-e device because we implemented multiple issue and completion based on the aio interface.

multiple device performance

For this experiment, we made only one partition in an NVM-e SSD. Then we created four files in ”raw” format, and each of the files is connected to guest as a virtual storage node in order to export four storage devices in VM. We restricted the number of queues and dedicated threads to ones. We assumed four processes that lead to 500K IOPS which is the ideal result as shown in Figure 11 since the numbers of devices and I/O threads are fours. Figure 14 shows the result. The performance in the original case is improved when more devices are attached. But the degree of performance improvement is getting smaller in

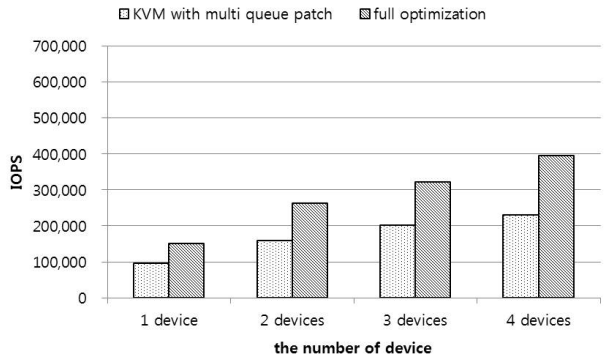

Figure 2.13 IOPS for NVMe SSD in the direct access test (fio, direct, 32 threads per device & 4K random read) on multiple virtual storage, our approach is taken except for the number of multiple queues and a dedicated thread to increate the cpu utilization

the data plane. On the contrary, compared with our approach, the performance gain is small in the case where the number of attached devices is small, but the more devices are connected to VM, the more the performance is improved. So when four devices are connected, the performance has improved by about 80% compared with the original case. Note that when a VM makes and uses multiple file based storage devices from a single physical device, this performs better compared to a single storage device with multiple thread connections. This means that a VM has a high amount of I/O processing capability but that single VM storage will face a scalability problem when high IOPS is required.

2.3 related work

There was research to optimize the host block layer to fully utilize SSD[14]. In this work, they introduced a request queue incurring the block layer overhead and tried to delete it. The authors of the [13] proposed a new request queue for multi-core architecture by splitting the queue and reducing the lock contention.

We also thought that the current request queue incurs overhead for SSD.

On the other hand, the authors of the papers [13][14] took a host OS based approach. We should minimize the work for queueing and scheduling in the guest context. The authors of the paper[11] proposed a minimized request queue by temporal merging. And we believe that this can be effective in the guest block layer, too. The guest block should be optimized by a queue which is in charge of communication between guest and host.

The paper[16] explains the problem that scheduling delay in VMM (VM Monitor) leads to delay in the I/O process in VM. This is a serious problem in a low latency required environment. This problem should be addressed in the cases of using NVMe and SATA based SSDs. In KVM, VMs are emulated by threads that consist of vCPUs with I/O threads. I/O in VM is thus different from I/O process in host. Scheduling delay in VM is more important than that in host OS because the I/O process in QEMU decides if sending I/O requests to host OS or sending them back to VM after it receives them from VM for the first time. Our approach can solve this problem because the process that in charge of I/O in our approach checks I/O requests by polling, which should be more effective by considering cpu affinity.

The author of the paper[17] optimized networking I/O in VM by taking a polling based approach. This work however focused on networking I/O which is processed in a stack different from that for storage I/O. It was necessary to modify the host network layer in kernel. On the other hand, in our work only QEMU and the guest device driver were modified, and the storage stack was optimized by using the existing methods which might minimize changes in QEMU and the guest device driver.

Chapter 3

Conculsion

The I/O performance of direct access in guest is as important as that in host in order for datacenters to adapt SSDs in the VM environments. In this paper, we have proposed an "exitless" approach by polling, which is based on only software, specifically pipelined polling for parallel processing. We have also developed and used the optimization method of multiple issues & completions, early completion, and batching to ensure high performance in the case of using a single device in VM. We believe that our approach will lead to improving the performance of the I/O layer for both guest and host when not only current fast storage but also future enhanced storage are used.

We are investigating the following issues to improve I/O efficiency for VM as well. 1) It is possible to take a hybrid approach that ensures low latency based on combination of polling and interrupt for the VM block layer, 2) the number of issue and completion threads can be increased or decreased dynamically whenever resources are needed for increasing the efficiency, and 3) low delay scheduling for I/O threads is considered and designed for guest and host. 4) As a comprehensive solution for VM which considers VFS, the file system and block layer are needed to issue I/O requests as fast as possible. 5) There is

the I/O scheduler mechanism both in the guest and host sides. These duplicate software layers may incur overhead, and it can become greater for faster storage [7]; therefore, the guest I/O scheduler should be removed when fast storage is used.

Bibliography

[1] HUFFMAN, A.NVMExpress specification 1.1a http://www.nvmexpress.org/specifications/, September 2013

[2] Rusty Russell: virtio: towards a de-facto standard for virtual I/O devices. Operating Systems Review 42(5): 95-103 (2008)

[3] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, Dan Tsafrir: ELI: bare-metal performance for I/O virtualization. ASPLOS 2012

[4] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, Razya Ladelsky: Efficient and Scalable Paravirtual I/O System. USENIX Annual Technical Conference 2013

[5] Qemu data-plane, ftp://public.dhe.ibm.com/linux/pdfs/KVM_Virtualized_IO_Performance_Paper.pdf

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori.KVM: the Linux virtual machine monitor. In Ottawa Linux Symposium (OLS), 2007.

[7] Vhobs-blk, http://www.linux-kvm.org/wiki/images/f/f9/012-forum-virtio-blk-performance-improvement.pdf

[8] Hui Lv, Yaozu Dong, Jiangang Duan, Kevin Tian: Virtualization challenges: a view from server consolidation perspective. VEE 2012

[9] Alex Landau, Muli Ben-Yehuda, Abel Gordon: SplitX: Split Guest/Hypervisor Execution on Multi-Core. WIOV 2011

[10] Jisoo Yang, Dave B. Minturn, Frank Hady: When poll is better than interrupt. FAST 2012

[11] Youngjin Yu, Dongin Shin, Woong Shin, Nae Young Song, Hyeonsang Eom, Heon Young Yeom: Exploiting Peak Device Throughput from Random Access Workload. HotStorage 2012

[12] Posted-interrupt, http://www.linux-kvm.org/wiki/images/7/70/2012-forum-nakajima_apicv.pdf

[13] Matias Bjørling, Jens Axboe, David W. Nellans, Philippe Bonnet: Linux block IO: introducing multi-queue SSD access on multi-core systems. SYSTOR 2013

[14] Eric Seppanen, Matthew T. O'Keefe, David J. Lilja: High performance solid state storage under Linux. MSST 2010

[15] ZHENG, D., BURNS, R., AND SZALAY, A. S. Toward millions of file system IOPS on low-cost, commodity hardware. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ' 13 (New York, New York, USA, Nov. 2013), ACM Press, pp. 1–12.

[16] Cong Xu, Sahan Gamage, Hui Lu, Ramana Rao Kompella, Dongyan Xu: vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. USENIX Annual Technical Conference 2013: 243-254

[17] Jiuxing Liu, Bülent Abali: Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization. ICS 2009: 225-234

[18] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae-Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, Heon Young Yeom: Optimizing

the Block I/O Subsystem for Fast Storage Devices. ACM Trans. Comput. Syst. 32(2): 6 (2014)

[19] Dongin Shin, Youngjin Yu, Hyeong Seog Kim, Jae-Woo Choi, Do Yung Jung, Heon Young Yeom: Dynamic Interval Polling and Pipelined Post I/O Processing for Low-Latency Storage Class Memory. HotStorage 2013

[20] Timer-Based Interrupt Mitigation for High Performance Packet Processing", 5th International Conference on High-Performance Computing in the Asia-Pacific Region, Sep. 2001

요약

SSD(solid-state drives) 같은 flash device가 등장함에 따라 VM(virtual machine) 환경에서 storage에 관한 I/O는 low latency가 요구되는 random I/O에서 문제가 되고 있다. VM 환경에서의 성능문제는 gues os로 인한 추가적인 software layer 존재와 guest와 host os 상의 context switching 에 따른 오버헤드, I/O process scheduling delay같은 문제가 있는데 이와 같은 요인은 HDD같은 high latency 를 가지고 batching이 전제된 device에서는 문제가 없지만 fast storage에서는 심각한 성능 저하를 가져온다. 따라서 이를 해결 하기 위해 본 논문에서는 첫번째로 pipelined polling 과 두번째로 multiplr issue and multiple completion 방법을 제안하고 VM I/O를 최적화 한다. 본 방법으로 다수의 vm storage들을 사용할시 성능이 50퍼센트 향상이 되었고 cpu 사용률은 최대 25퍼센트 감소 하였으면 하나의 vm storage 성능도 80퍼센트 이상 가능한 구조를 얻었다.

주요어: 서울대학교, 전기 컴퓨터 공학부, 졸업논문
학번: 2013-20825