Performance Analyses of Duplicated I/O Stack in Virtualization Environment

Minhoon Yi Dong Hyun Kang Minho Lee Inhyeok Kim Young Ik Eom
College of Information and Communication Engineering,
Sungkyunkwan University
Suwon 440-746, Korea
+82-31-290-7219
{gogodatem, kkangsu, minhozx, kkojiband, yieom}@skku.edu

ABSTRACT

Recently, many studies have focused on eliminating duplicated I/O stack of virtualization because I/O requests of a guest machine have to pass through the I/O stack of both the guest and host machine. In this paper, we analyze the effect of duplicated I/O stack (e.g., page cache, file system, I/O scheduler, and device driver) between the guest and host machine. Especially, we study an I/O scheduler of the guest machine and the virtio of QEMU to understand how the I/O scheduler affects the overall I/O performance. Our experimental results show that Noop scheduler can lead to better performance than other schedulers, such as CFQ and Deadline. As expected, virtio shows the best performance in sequential write pattern since it eliminates the overhead caused by I/O scheduler of the guest machine by bypassing its I/O scheduler. However, in this paper, we report that the virtio reveals performance limitation in random write pattern. Our experimental results clearly show that the duplicated I/O stack significantly impacts on the overall performance of the guest machine. Especially, using virtio to eliminate the I/O scheduler of the guest also has negative impacts on I/O performance according to the write pattern.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance

General Terms

Measurement, Performance, Experimentation, Verification

Keywords

Virtual Machine, Virtualization Environment, I/O Stack, Linux I/O Scheduler, Duplication, Virtio, KVM

1. INTRODUCTION

Nowadays, the virtualization has become a general technique and it is widespread in computing environments, such as desktop, server, cloud system, and mobile devices. However, I/O performance of the virtualization still remains a challenge [1, 2, 3, 4]. Especially, duplicated I/O stack (e.g., page cache layer, file system, I/O scheduler, and device driver) between the guest and host machine reveals serious performance degradation. Therefore, many researchers focused on minimizing the overhead caused by duplicated I/O stack in virtualized systems. Some studies proposed a framework that bypasses the duplicated I/O stack and uses a dedicated thread for improving I/O performance of guest VM (virtual machine) [4, 5]. Other researchers extended virtio [6], which provides a virtqueue to bypass an I/O block layer of guest VM, to address a scalability issue when SSD is used as a secondary storage [3].

In line with previous studies, this paper focuses on the duplicated I/O stack between the guest and host machine and analyze the effects of the duplicated I/O stack. In particular, we focus on I/O scheduler of a guest VM to understand how the I/O scheduler affects the overall I/O performance in detail. For performance measurement of the virtualization, we use KVM (Kernel-based Virtual Machine) with Linux kernel 3.13.0. The KVM provides the most popular virtualization environment by employing the QEMU emulator. In order to study the effectiveness of I/O scheduler, we first check I/O patterns of different I/O scheduler by using default I/O schedulers of Linux (e.g., CFQ, Deadline, and Noop). To analyze the effect of de-duplicated I/O scheduler, we also collect the I/O patterns when a guest VM employs the virtio of QEMU [7], which allows I/O requests of a guest VM to bypass its I/O scheduler by using the virtqueue. We also measure each performance and compare them with I/O benchmarks, such as FIO [8] and IOzone [9]. Our experimental results show that Noop scheduler can lead to better performance than other I/O schedulers because it only performs merge operations to reduce the number of I/O requests. As expected, the virtio achieves the best performance in sequential write pattern. However, virtio reveals the performance limitation in random write pattern. These results clearly show that duplicated I/O stack significantly impacts on the overall performance of a guest VM. However, using virtio to eliminate I/O scheduler of the guest also has negative impacts on I/O performance.

The remainder of this paper is organized as follows. In section 2, we provide background knowledge concerned with our approach. Section 3 explains experimental design for capturing I/O pattern and measuring performance improvement. The experimental result and analysis are presented in section 4. In section 5, we overview related work. Lastly, we conclude our paper in section 6.

IMCOM '16, January 04-06, 2016, Danang, Viet Nam
© 2016 ACM. ISBN 978-1-4503-4142-4/16/01…$15.00
DOI: http://dx.doi.org/10.1145/2857546.2857573

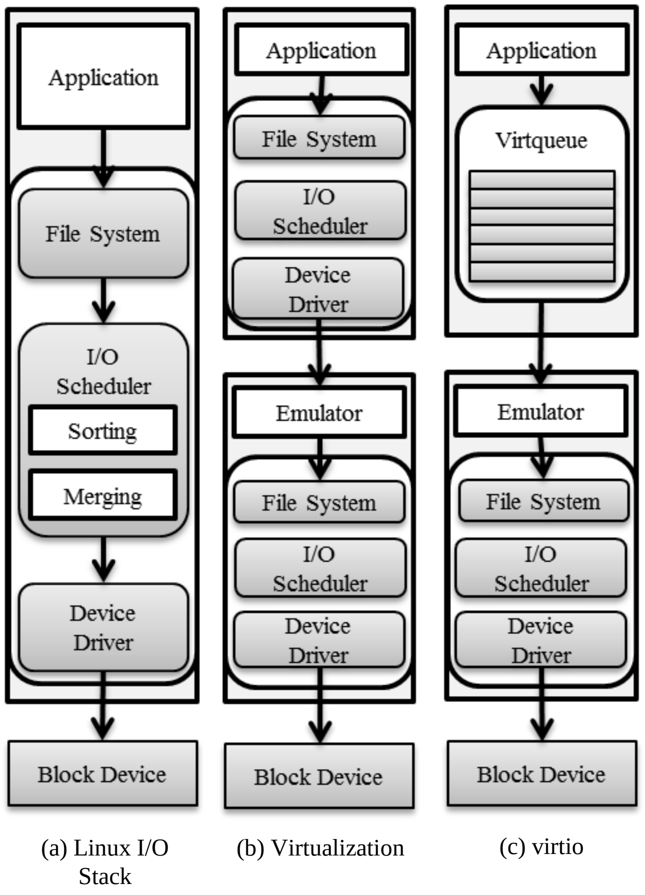


Figure 1. I/O Stack Structure

2. BACKGROUND

2.1 KVM

KVM (Kernel-based Virtual Machine) is included in Linux kernel since version 2.6.20 [10]. KVM, which is implemented as a kernel module, provides full-virtualization environment based on Linux. Therefore, hypervisor of KVM is handled as a sub-module of the kernel, such as memory management unit or file systems. In order to support full-virtualization, it needs processors having a hardware-based virtualization feature, such as Intel VT-x and AMD-V. Virtual machines run guest OS without any modification since KVM provides full-virtualization environment. KVM uses the QEMU which emulates physical devices, such CPU, memory, and storage. The virtual machine using KVM is identified by the host as a process. Accordingly, the host runs virtual machines among processes of the host. I/O scheduler of KVM uses existing Linux I/O schedulers without any modification.

2.2 Linux I/O Stack

Basically, Linux I/O stack consists of file system, I/O scheduler, and device driver as Figure 1a. File system is abstract layer for saving and controlling data. I/O scheduler sort and merge I/O requests. Device driver is a set of kernel routines for accessing hardware devices. I/O processes in the Linux I/O stack are described as in the following. I/O requests, which are generated by process, pass through the block layer of file system. Then I/O requests are converted into BIO which is block I/O unit. BIOs are sorted into block address, and then adjoined BIOs are merged into requests by I/O scheduler. Requests are inserted in request queue of device drivers. Next, request queue transfers requests to block device such as HDD or SSD. In virtualization environment, I/O stack exists in both host and virtual machine. Figure 1b shows I/O stacks in the virtualization environment. I/O requests of virtual machines are sent to I/O stack of host. Host gathers I/O requests of each virtual machine. After that, the host delivers gathered I/O requests to physical block devices by using own I/O stack. Host handles own processes and I/O requests of virtual machines in the same way. It is because host identifies a virtual machine as a process. Thus, I/O requests scheduled in I/O scheduler of the virtual machine are rescheduled in I/O scheduler of the host.

2.3 Virtio

Virtio, which is I/O virtualization driver, is included in Linux kernel since version 2.6.24 [6]. Virtual machine exploits virtio as a separate device driver. Virtio supports various drivers such as network driver, block device driver, and memory ballooning. Virtual machine has a virtqueue instead of the established I/O stack as Figure 1c. Host and virtual machine share the virtqueue for I/O processes. Vmexit, exception call, occur when I/O requests of virtual machine comes to the virtqueue. Vmexit switches mode of host cpu from guest mode to host mode. Then host can handle I/O requests. When I/O request is completed, a reply is inserted into the virtqueue and then cpu mode returns to guest mode. In order to bypass I/O stack of the virtual machine, host and virtual machine share the virtqueue that reduces overhead of I/O stack duplication.

2.4 Linux I/O Scheduler

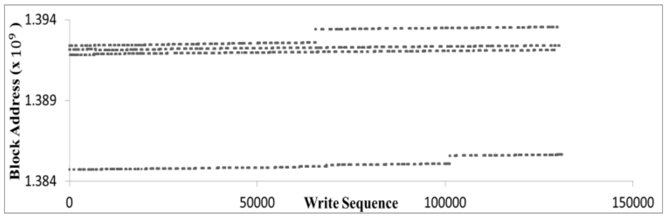
Linux I/O schedulers sort and merge I/O requests. Sorting reorders I/O requests for reducing seek time of physical disks. Merging reduces the number of I/O requests by combining adjacent I/O requests. Linux I/O schedulers have a request queue to contain I/O requests. Linux basically provided 4 I/O schedulers in kernel version 2.6.

CFQ (Complete Fairness Queueing) scheduler is default option in kernel version 2.6. It handles I/O request in I/O queue of each process by round-robin scheme. CFQ guarantees fairness I/O bandwidth of processes each other. Actually, CFQ shows regular performance in diverse situations.

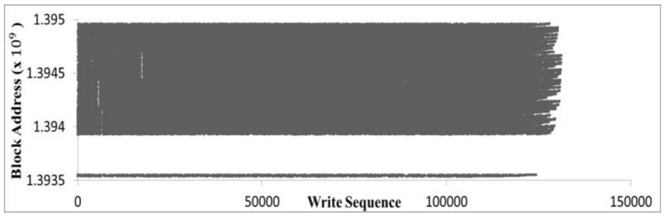
Deadline scheduler gives deadline to each I/O request. I/O requests are handled within the deadline. Deadline scheduler has 4 FIFO queues which are sorted into read deadline, write deadline and block address about read and write requests. This scheduler can prevent starvation of I/O requests by keeping deadline strictly.

Noop scheduler, main I/O scheduler in this paper, is the simplest scheduler. Noop scheduler operates just simple FIFO scheme. Basically, it does not support operation of seek time reduction like sorting for I/O requests. Noop scheduler only provides merging operation for adjoined I/O requests. It is the reason that Noop scheduler does not have overhead of sorting. Noop scheduler shows good performance in high speed random access device.

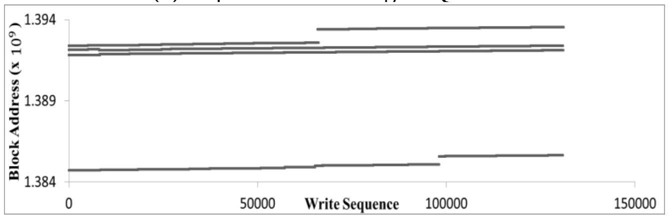
Anticipatory scheduler is based on Deadline scheduler that we mentioned. This scheduler is specialized in read requests. The reason is that Deadline scheduler predicts and waits other read requests when a read request occurs and then this scheduler handles those requests. While Deadline scheduler improves performance of read requests, it wastes time for write requests. Anticipatory scheduler was excluded from Linux kernel since version 2.6.33. For that reason, we do not treat anticipatory scheduler in this paper any more.



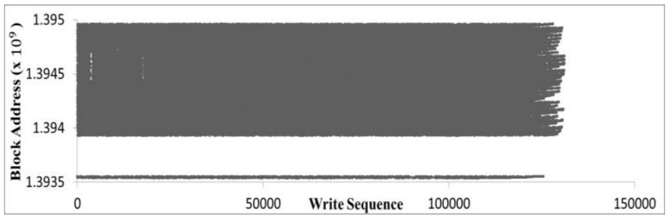
(a) Sequence Write using CFQ in VM



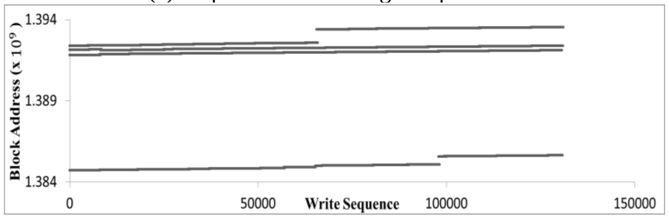
(b) Random Write using CFQ in VM



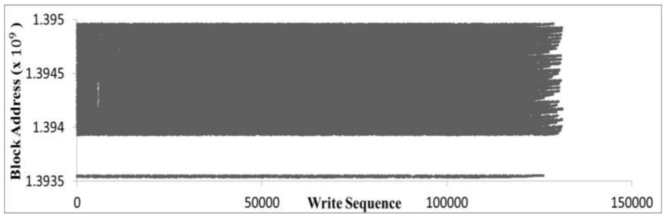
(c) Sequence Write using Noop in VM



(d) Random Write using Noop in VM



(e) Sequence Write using virtio in VM



(f) Random Write using virtio in VM

Figure 2. Block Address Patterns of I/O Workloads

3. EXPERIMENTAL DESIGN

All experiments were performed on a system equipped with Intel Core i5-4570, 8 GB of RAM, and 500 GB of Western Digital HDD. In addition, we conducted a virtualized system by using KVM, which utilizes QEMU emulator to employ virtio and device emulation. The virtual machine was configured with 4 GB RAM and 4 vcpu. Both host and virtual machine run Ubuntu 14.04 LTS with Linux kernel 3.13.0. For evaluation, we designed several types of configuration: (1) virtual machine using CFQ, Noop, and Deadline scheduler (2) virtual machine using virtio. To measure the I/O performance, we used a series of representative benchmarks including FIO [8] and IOzone [9].

These benchmarks made I/O workloads which have sequential writes and random writes. We employed libaio as ioengine by setting FIO option. FIO benchmark can measure I/O bandwidth, iops, and latency according to job file which is a special configuration file. Users can also define particular I/O operations in the job file as options. IOzone benchmark is one of file system benchmarks for measuring file system performance. It uses file I/O workloads of diverse sizes for various situations. It can be used in many operating systems, such as Linux, Windows, and Mac OS. We used these two benchmarks for generating I/O workloads and measuring I/O performance factors, which are I/O bandwidth, latency, and throughput of test cases. Benchmark tools usage and description of workloads in this paper are summarized in the Table 1.

4. EVALUATION

4.1 Block access pattern

To compare block access patterns of each experimental group in the virtual machine, we captured I/O block access patterns by using FIO and blktrace [11]. In this experiment, we used default I/O schedulers including CFQ, Deadline, and Noop scheduler without any modification. To check the effectiveness of eliminating I/O stack in the virtual machine, we also used virtio. We created 128 MB file with sequential and random write patterns.

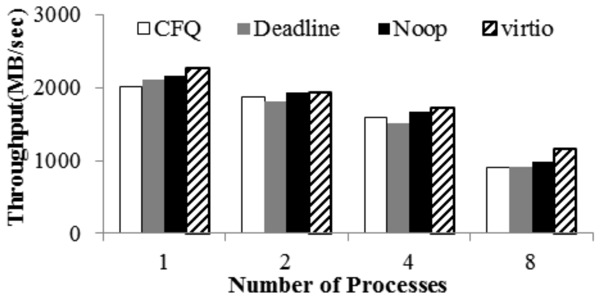
In case of the sequential write pattern, a result of using Noop scheduler is similar to that of using virtio as shown in Figure 2. It is because that Noop scheduler just merges adjoined I/O requests without sorting. Virtio only carries out I/O requests by using the virtqueue without I/O stack of virtual machine. However, CFQ scheduler made a dotted write pattern as shown in Figure 2a. The dotted lines mean switching I/O request of each process by CFQ scheduler. It is because that CFQ scheduler emphasizes fairness among processes as aforementioned in Section 2.4. This scheduler deals with I/O requests using round-robin scheme which guarantees fairness and the minimum processing time to each I/O request. For this reason, CFQ scheduler prevents that a particular process monopolizes the processing time. In case of random write pattern, the results show that block access patterns are almost same in all cases. It demonstrates that scheduling and I/O stack in virtual machine do not affect block access patterns.

Table 1. Bandwidth Usage

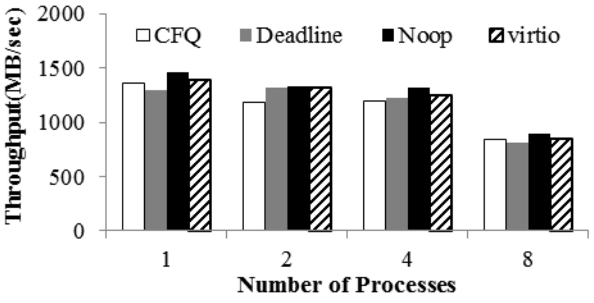
Benchmark	*I/O Workload*	*Measurement Factor*
FIO	Sequential/Random Write	Bandwidth, Latency
IOzone	Sequential/Random Write	Throughput

4.2 I/O Performance

We measured overhead of I/O stack in second experiment. We used 4 test groups which are using CFQ, Deadline, Noop scheduler, and virtio in virtual machine. The host uses default I/O scheduler without any modifications. Each case consists of 1 to 8 processes having sequential and random writes workloads.

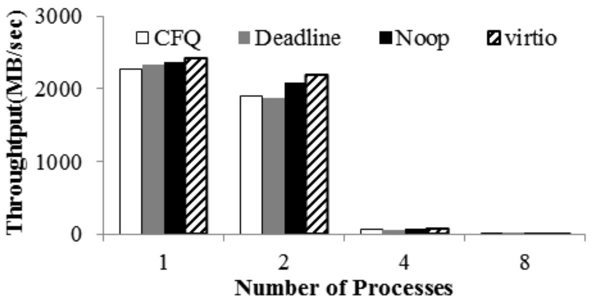


(a) Throughput of 32MB Sequential Write

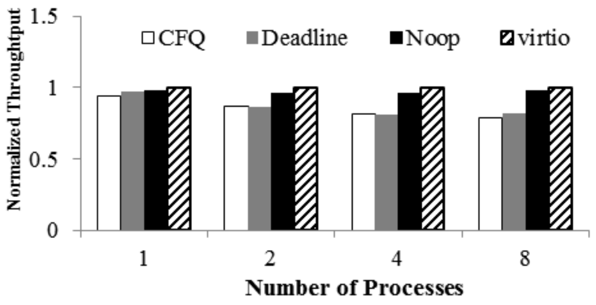


(b) Throughput of 32MB Random Write

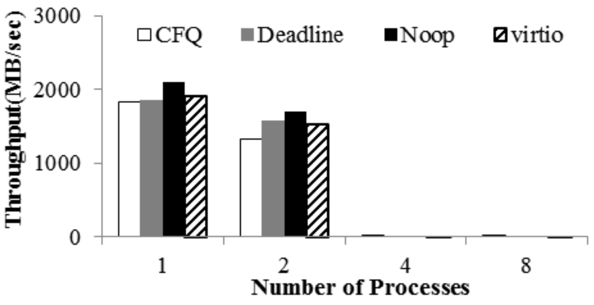
Figure 3. Throughput of 32MB Writes with IOzone



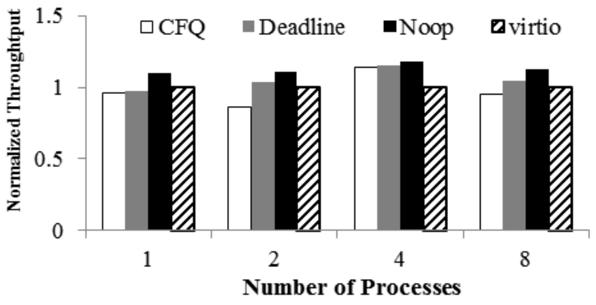
(a) Throughput of 128MB Sequential Write



(b) Throughput of 128MB Random Write



(c) Normalized Throughput of Sequential Write



(d) Normalized Throughput of Random Write

Figure 4. Throughput and normalization of 128MB Writes with IOzone

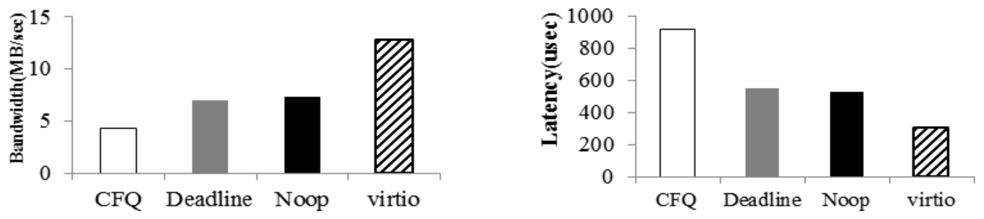
We performed the experiments by utilizing IOzone benchmark and creating 32 MB file and 128 MB file, respectively. Then, we measured the performance in terms of throughput. As shown in Figure 3 and Figure 4, virtio outperforms the other I/O scheduler by up to 22% in sequential writes. The major reason is that merging and sorting in I/O scheduler and I/O stack in virtual machine make performance degradation. Virtio does not have I/O stack of the virtual machine. It only stacks I/O requests in the virtqueue in the order of arrival. Thus, sorting is an unnecessary process and merely overhead for sequential write. Noop scheduler also well performs among I/O schedulers. However, CFQ shows the worst I/O performance because it has overhead of context switching among processes.

In case of random writes, throughput of Noop scheduler has better by up to 10% than that of virtio. This difference comes from merging in Noop scheduler. Merging operation combines adjoined I/O requests, and then it reduces the number of I/O requests. Merging assists to reducing overhead of I/O requests. Especially, sharp performance degradation is detected in Figure 4a and Figure 4c. It is the result from lock contention problem. The more processes simultaneously request I/O, getting the lock is more difficult. It affects not only performance of I/O but also performance of whole system. In order to compare throughputs easily, we normalized by throughput of virtio in Figure 4b and Figure 4d.

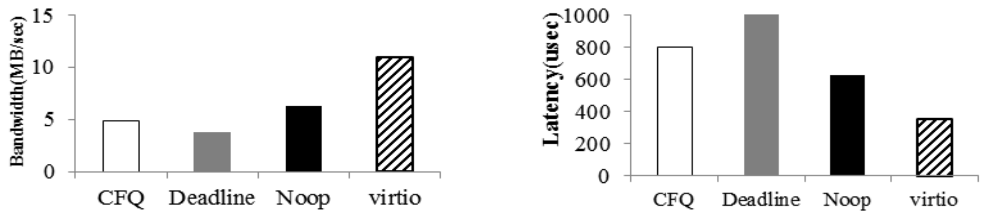
We also evaluated the I/O performance using FIO that is for measuring bandwidth and latency. We created 4 I/O processes and performed the experiment with 32 MB and 128 MB file. Figure 5 shows that virtio has the best performance in terms of bandwidth and latency in sequential writes. Noop scheduler also represents the best bandwidth and latency in random writes. It is the same reason as above mentioned. However, in some experiments, Deadline scheduler generates the worst case as shown in Figure 5b and Figure 5c. It is basically more specialized in read operation. This scheduler gives a strict deadline to read operations.

5. RELATED WORKS

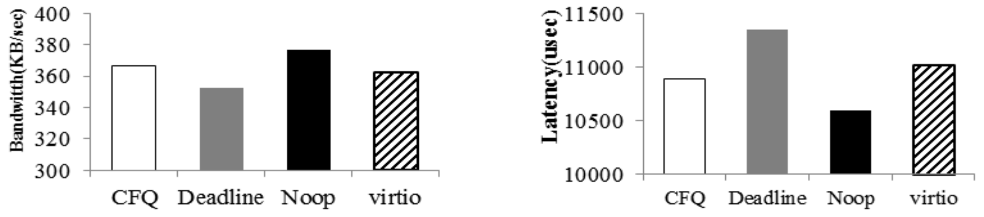
Previous studies focused on I/O overhead in virtualization environment. The reason of overhead is complicated interaction between host and virtual machine. There are two approaches for reducing this overhead of virtualization. The first studies remove and bypass I/O stack for elimination of overhead caused by I/O stack duplication. The second studies use existing Linux I/O



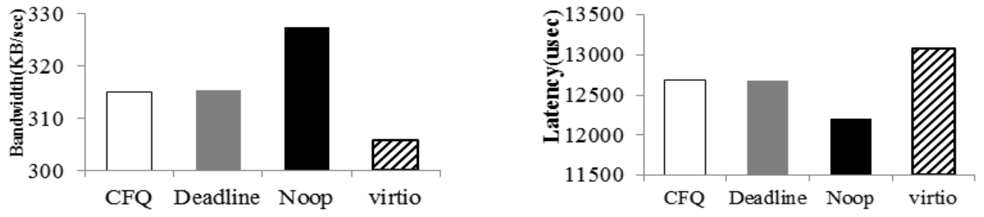
(a) Bandwidth and Latency of 32MB Sequential Write



(b) Bandwidth and Latency of 128MB Sequential Write



(c) Bandwidth and Latency of 32MB Random Write



(d) Bandwidth and Latency of 128MB Random Write

Figure 5. Bandwidth and Latency of workloads with FIO

schedulers and combinations of them without modification to find appropriate one.

The Turtles Project [12] of IBM is a most of representative study about I/O stack bypass. This project is used to improve security and live migration of hypervisor in cloud environment. In order to implement, it exploits a technique called the nested virtualization that overlaps virtual machine within virtual machine. This project provides multi-level device assignment for nested virtual machines. However, multi-level device assignment needed IOMMU for supporting I/O of virtual machine in the nested virtualization. It is because multi-level device assignment is based on direct device assignment. Higher level virtual machine, the nested virtual machine, can directly access hypervisor and physical device of the lowest level without passing through middle level hypervisors and virtual machines.

ELVIS [5] is based on virtio that we mentioned in this paper. It is a solution of the vmexit problem. Vmexit, one of exception calls, is generated by change of cpu modes. Virtual machines gather I/O requests for sending those to host. For handling I/O requests of virtual machines, cpu mode is switched from guest mode to host mode. Frequently occurred vmexit decreases performance of virtualization. ELVIS exploits the poling mechanism and an exclusive thread for reducing vmexit exception calls. Polling mechanism can get I/O requests in shared memory between host and virtual machines without vmexit calls. When I/O requests come to shared memory, thread uses polling mechanism to process I/O requests instantly.

As mentioned above, hypervisor bypass and removal of I/O stack can also improve performance of I/O virtualization. J. Liu et al. [13] suggested VMM-bypass in Xen environment. When virtual machine uses physical devices, additional overhead occurs. The reason is that I/O requests have to pass through device driver of virtual machine and hypervisor. For removing this overhead, VMM-bypass, which is based on OS-bypass, makes that virtual machine access I/O device directly. This approach reduces the number of context switches and then improves I/O performance in virtualization environment.

D. Boutcher et al. [14] focused on characteristic of existing Linux I/O scheduler and combination of them. They research into ways of improving I/O performance in virtualization by experiments. I/O schedulers are duplicated between host and virtual machines in virtualization environment. Duplication of I/O schedulers affects I/O performance, and therefore using of the proper combination of I/O schedulers is helpful for the performance. Their study shows actual effects of I/O scheduler combinations on the performance. Noop scheduler, Linux I/O scheduler, is the worst in fairness by their experiments. However, it demonstrates the best performance regardless of hypervisor, workloads and virtual machines. As mentioned earlier, Noop scheduler does not sorting, but merging for I/O requests.

Appearance of high speed I/O device like SSD (Solid Storage Drive) brings new studies of I/O virtualization. J. Kim et al. [15] consider characteristic of Linux I/O schedulers. In SSD environment, this paper has performance tests of CFQ, Deadline, Noop schedulers. Noop scheduler is the best scheduler in their experiments. While CFQ and Deadline scheduler sort and merge I/O request, Noop scheduler just supports merging. Modified I/O scheduler used for traditional HDD is proposed as I/O performance improving method. In addition, Y. Son et al. [16] modified BIO structure in SSD environment. PIO removes traditional I/O stack and optimizes I/O for SSD.

6. CONCLUSION

In this paper, we focused on measuring the overhead of I/O stack duplication between host and virtual machine. We experimented on performance degradation caused by I/O stack duplication. By removing I/O stack of the virtual machine, performance of I/O requests is improved. The result of experiments demonstrated that duplication I/O stack can be the overhead. According to the first experiment, unnecessary duplication of I/O stack exists in the virtual machine. Whether I/O requests pass through I/O stack in the virtual machine or not, that does not affect block access patterns. In the second experiment, we found the overhead of I/O stack duplication. Removal of I/O stack shows the performance improvement in sequential writes. However, elimination of I/O stack can't always guarantee the best performance. In random writes, merging assists I/O performance by reducing the number of requests. For improving I/O performance in the virtualization environment, the minimum function of I/O stack, like merging, is necessary as the occasion demands.

Virtual machine uses only one of architectures, which are I/O stack with I/O scheduler or virtio bypassing the I/O stack, at a time. In this case, I/O requests of virtual machines can't have the best performance both sequential and random writes. We plan implementation of a module which classifies with sequential and random I/O requests, for bimodal architecture. To improving I/O performance, this module automatically selects the path either the existing I/O stack or virtio without I/O stack. In addition, we adapt this module to virtualization environment using SSDs.

7. ACKNOWLEDGMENTS

This work was supported by ICT R&D program of MSIP/IITP. [R0126-15-1065, (SW StarLab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays]. Young Ik Eom is the corresponding author of this paper.

8. REFERENCES

[1] Agesen, O., Mattson, J., Rugina, R., and Sheldon, J. 2012. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proceedings of the USENIX Annual Technical Conference* (Boston, USA, June 13 - 15, 2012). ATC'12. USENIX, Berkeley, CA, 373-385.

[2] Landau, A., Ben-Yehuda, M., and Gordon, A. 2011. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In *Proceedings of the Workshop on I/O Virtualization* (Portland, USA, June 14, 2011). WIOV'11. USENIX, Berkeley, CA, 1-7.

[3] Kim, T. Y., Kang, D. H., Lee, D., and Eom, Y. I. 2015. Improving Performance by Bridging the Gap between Multi-queue SSD and QEMU I/O Virtualization. In *Proceedings of the International Conference on Massive Storage Systems and Technology* (Santa Clara, USA, May 30 - June 5, 2015). MSST'15. IEEE, New York, NY, 1-11. DOI= http://dx.doi.org/10.1109/MSST.2015.7208295.

[4] Lee, K., Lee, D., and Eom, Y. I. 2015. Power-efficient and High-performance Block I/O Framework for Mobile. In *Proceedings of the ACM International Conference on Ubiquitous Information Management and Communication* (Bali, Indonesia, January 08 - 10, 2015). IMCOM'15. ACM, New York, NY, 1-7. DOI= http://dx.doi.org/10.1145/2701126.2701228.

[5] Har'El, N., Gordon, A., Landau, A., Ben-Yehuda, M., Traeger, A., and Ladelsky, R. 2010. Efficient and Scalable Paravirtual I/O System. In *Proceedings of the USENIX Annual Technical Conference* (San Jose, USA, June 26 - 28, 2013). ATC'13. USENIX, Berkeley, CA, 231-242.

[6] Russell, R. 2008. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*. 42, 5 (July, 2008), 95-103. DOI= http://dx.doi.org/10.1145/1400097.1400108.

[7] Bellard, F. QEMU. [Online]. Avaliable: http://www.qemu.org/

[8] Axboe, J. Fio-Flexible IO Tester. [Online]. Available: https://github.com/axboe/fio

[9] Norcott, W. D. IOzone Filesystem Benchmark. [Online]. Available: http://www.iozone.org/

[10] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. 2007. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Ottawa Linux Symposium* (Ottawa, Canada, June 27 - 30, 2007). OLS'07. Linux Symposium, Ottawa, Canada, 225-230.

[11] Axboe, J. Blktrace. [Online]. Available: http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html

[12] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., and Har'El, N. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (Vancouver, Canada, October 04 - 06, 2010), OSDI'10. USENIX, Berkeley, CA, 423-436.

[13] Liu, J., Huang, W., Abali, B., and Panda, D. K. 2006. High Performance VMM-Bypass I/O in Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference* (Boston, USA, May 30 - June 03, 2006). ATC'06. USENIX, Berkeley, CA, 29-42.

[14] Boutcher, D. and Chandra, A. 2010. Does Virtualization Make Disk Scheduling Passé?. *ACM SIGOPS Operating Systems Review*. 44, 1 (January, 2010), 20-24. DOI= http://dx.doi.org/10.1145/1740390.1740396.

[15] Kim, J., Oh, Y., Kim, E., Choi, J., Lee, D., and Noh, S. H. 2009. Disk Schedulers for Solid State Drives. In *Proceedings of the ACM International Conference on Embedded Software* (Grenoble, France, October 12 - 16, 2009). EMSOFT'09. ACM, New York, NY, 295-304. DOI= http://dx.doi.org/10.1145/1629335.1629375.

[16] Son, Y., Han, H., and Yeom, H. 2015. Optimizing File Systems for Fast Storage Devices. In *Proceedings of the ACM International Systems and Storage Conference* (Haifa, Israel, May 26 - 28, 2015). SYSTOR'15. ACM, New York, NY, 1-6. DOI= http://dx.doi.org/10.1145/2757667.2757670.