JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 36, 895-919 (2020)
DOI: 10.6688/JISE.202007_36(4).0013

A Methology for Improving the Performance of Paravirtual I/O Systems Based on Fast NVM Devices

MYOUNGWON OH[1], HANUL SUNG[1], SANGGEOL LEE[2],
HYEONSANG EOM[1] AND HEON Y. YEOM[1]
[1]*Department of Computer Science and Engineering*
Seoul National University
Seoul, 151-742 Republic of Korea
[2]*Semiconductor Department*
Memory Application Engineering Group
Samsung Electronics, 443-743 Republic of Korea
E-mail: mwoh@dcslab.snu.ac.kr; husung@dcslab.snu.ac.kr;
aiden.lee@samsung.com; hseom@cse.snu.ac.kr

Paravirtual I/O systems have been paid much attention to due to their reasonable performance levels. To increase these levels, Non-Volatile Memory (NVM) such as flash memory is considered and used as their storage media alternative to HDD. Although the storage media is fast, the performance of paravirtual I/O systems using the media is much lower than expected. The performance is lowered because the I/O process in the guest and host OSes is serialized and the OSes are run ignoring the processor affinity while the same software layers performing I/O are duplicated in the OSes. We present our methodology to employ towards optimizing the performance of NVM-based paravirtual I/O systems: the use of polling rather than interrupt, and parallel batching in order to maximize the parallelism in performing the sequence of I/O operations, and avoidance of context switches in order to consider the processor affinity. Our experiments with Kernel Virtual Machine (KVM) I/O systems using different NVM storage devices suggest that the use of the methodology can lead to enhancements in throughput by 50% to more than 80% while reducing CPU usage by up to 25% for a microbenchmark program and by up to 100% for workloads in mixed-read/write patterms.

Keywords: performance optimization, paravirtual I/O system, NVM

1. INTRODUCTION

As the need for sharing the resources of each of many large-scale computing facilities such as datacenters in executing a variety of workloads on it has increased, different kinds of virtualization techniques have been developed and used, providing virtualized computing environments. Since the amount of data to access and process in these environments is often huge, achieving high I/O performance in terms of latency and/ or throughput in such environments is one of the major issues regarding the virtualization.

Efficient I/O virtualization techniques have been developed. One of the most popular I/O virtualization techniques currently being employed is paravirtual I/O such as KVM's virtio [1] and VMware's VMXNET3 [2]. In paravirtual I/O, the host provides a virtual I/O

Received April 13, 2018; revised July 2, 2018; accepted September 21, 2018.
Communicated by Cho-Li Wang.

device to its guests. It has been found that traditional paravirtual I/O systems significantly slow down mainly when a guest and the host make context switches performing exits [3] or there are multiple competing I/O intensive guests. There have been attempts to address these problems. ELVIS is one of them, using a fine-grain I/O scheduler combined with exit-less interrupts on x86 processors in order to improve the paravirtual I/O performance [4]. As an alternative approach to enhance the performance of a paravirtual I/O system, it is natural to consider using fast storage technologies for the system. Many people have experienced performance boosts with their computing systems or devices by replacing HDDs with flash memory-based SSDs. Also, recent high-end flash memory-based SSDs have become even faster; for example, Samsung 845DC evo (a SATA SSD) and Samsung XS1715 (an NVMe [5] SSD), show about 70K IOPS (I/Os per second) and 750K IOPS, respectively, in physical machine environments. It is expected that the use of these fast flash memory-based SSDs in a computing system may lead to lower latency and higher throughput in the execution of the system. Moreover, these kinds of SSDs have been becoming cheaper. For these reasons, the use of these fast Non-Volatile Memory (NVM)-based SSDs may be adopted in the first place.

However, even when very fast NVM-based SSDs are used for a system, the performance of the system may not be increased accordingly due to the software overhead. The more the performance of the SSDs is enhanced, the more does the overhead affect the performance. Therefore, in order to take the advantage of utilizing fast NVM-based SSDs for a system at maximum, it is crucial to minimize the software overhead. For instance, a study has recently showed that the performance of a flash memory-based system even in a physical machine environment can be significantly increased by minimizing software delays caused by additional I/O processing contexts such as interrupt bottom halves and background run queues [6]. Although the storage media for a paravirtual I/O system is fast, the performance of the system is expected to be the lower due to the inherent virtualization overhead; guest Virtual Machines (VMs) are emulated by the host as software components, and almost the same I/O stack is duplicated in each of the guests and the host. The overhead increases when guests perform random I/O because they need to communicate with the host more. In addition to the additional I/O layers, exit-based notifications between the guest and host OSes incur huge overhead also as explained in ELVIS [7, 8].

Fig. 1 shows the duplication of I/O stack and exit-based notifications between the guest and host OSes in the I/O flow in Linux KVM, a paravirtual system. The following is illustrated in the figure: an I/O request is issued in a guest VM, it is transferred to the virtio [1] block driver, finally being queued into the shared internal virtqueue. The guest can notify the host of it only indirectly by first performing an exit to the hypervisor, QEMU (Quick EMUlator) [9] to have the hypervisor do the notification. After the host performs I/O, it can notify the guest of the completion only indirectly by first causing the running guest to exit in order to run the hypervisor so that the hypervisor can inject the virtual interrupt to the guest.

A previous study suggests that the performance of an existing paravirtual I/O system using one SSD may be comparable to that of a non-virtual I/O system using the device, but the performance is degraded when multiple devices are used or when a PCIe or NVMe SSD with very high throughput is used as a file-based storage device [10]. It is found in the study that the performance of executing random read operations using a flash memory-based device can be improved if multiple I/O requests are issued and multiple completions are processed simultaneously, and that the performance of executing random read operations using multiple flash memory-based devices may be enhanced if the I/O delay caused by exits and context switches is reduced, combined with using a processor

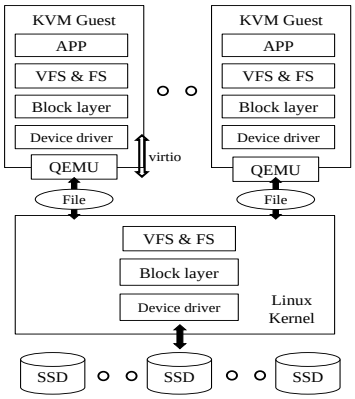


Fig. 1. I/O system of KVM.

affinity-aware method of scheduling I/O threads. By extending this study, it is analyzed why the performance of an NVM-based paravirtual I/O system is lowered even though the storage media is very fast.

The performance of a fast NVM-based paravirtual I/O system is degraded because the I/O process in the guest and host OSes is serialized and the OSes are run ignoring the processor affinity while the same software layers performing I/O are duplicated in the OSes. As explained for the I/O flow in Linux KVM, the steps of the I/O process in the guest and host Oses are taken in a serialized manner, causing two exits with two context switches. Even a third exit may be performed in some implementations in order for the guest to write into the End-of-Interrupt (EOI) register when it finishes processing the virtual interrupt for the notification of the I/O completion [8]. The serialized course-grained I/O processing of the paravirtual I/O system leads to the minimization of the parallelism in performing the I/O operations causing unnecessary waits. If the guest and host OSes are run on the same CPU ignoring the processor affinity, the performance is lowered by the delay due to cache misses.

We present our methodology to employ towards optimizing the performance of NVM-based paravirtual I/O systems. There are several component methods to use in order to maximize the parallelism of performing the sequence of I/O operations. Polling may be used rather than injecting an interrupt to send a notification so that as many non-serialized operations can be performed in parallel as possible. This non-serialization permits processing notified multiple events in a batch. Also, the guest and host OSes are run on different CPUs to avoid unnecessary context switches considering the processor affinity. In principle, the methodology should be hardware independent [10], based only on software without requiring special permissions not causing any security problem, affecting only the minimal part of the system [7, 8], and leading to high throughput and low latency without wasting CPU cycles. The CPU usage can be lowered by decreasing the lock contention in the execution of exits [10].

The main contributions of this work are summarized as follows:

- We thoroughly describe our methodology to employ towards optimizing a given NVM-based paravirtual I/O system with a detailed demonstration for Linux KVM.
- We present that the performance of using multiple I/O queues can be enhanced by pinning a queue per vCPU because the context switches are reduced and the locality is improved.

- We show the result of the complete evaluation on write operations in addition to read operations.

This paper is organized as follows: Section 2 explains why the performance of NVM-based paravirtual I/O systems is low. Section 3 presents our performance optimization methodology. Section 4 describes the design and implementation of the methodology for KVM. Section 5 discusses design issues for KVM. Section 6 shows the result of validation of employing the methodology for KVM. Section 7 explains related work, and Section 8 concludes the paper.

2. PROBLEM DEFINITION

2.1 Terms and Definitions

- *Virtual storage device* is a device file in guest OS. It can be a virtual file or real one if the device is connected by pass-through.
- *File-based storage* is a device file in guest OS which represents a file in host OS as shown in Fig. 1.

2.2 Performance Degrations

We obtained the following results by conducting experiments with the fio benchmark performing random I/O in a flash memory-based SSD equipped KVM system in order to check the effectiveness of existing mechanisms designed for NVM-based paravirtual I/O systems.

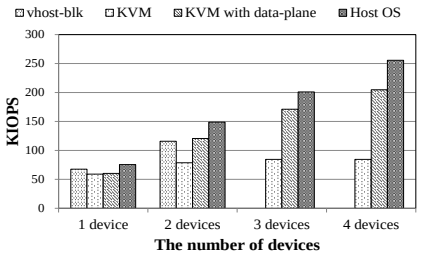


Fig. 2. IOPS for SSDs in direct-access tests (with 32 fio processes per device, performing 4KB random reads).

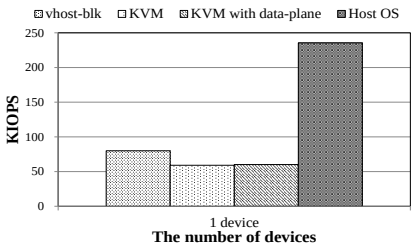


Fig. 3. IOPS for SSDs in direct-access tests (with 128 fio processes per device, performing 4KB random reads).

2.2.1 Performance for multiple devices

Fig. 2 shows the performance result for multiple devices, which means that each device corresponds to a file that is mapped to one virtual storage device in a VM. In other words, having four devices means that there are four virtual storage devices associated with four files in four different physical devices. As can be seen, the data plane from QEMU could achieve relatively good performance when there were multiple virtual storage devices in a single VM, but the performance was degraded by about 25% compared with the host OS case. In the vhost-blk case, the performance similar to that of host OS case was achieved. Note that it was not possible to perform the test if three or more devices were connected. Also, in the case of using KVM, the performance was significantly lowered, even if the tested VM was provided with sufficient CPU capacity.

2.2.2 Performance for a single device

Fig. 3 shows the performance result for four SSDs in RAID-0 which performed as a single (file-based) virtual storage device in the paravirtual environment. As shown in the figure, even though over 200K IOPS was achieved in host OS, there was significant performance degradation in the other cases. Particularly, a reasonable level of performance was achieved for each device when multiple devices were used in the data-plane case, but the performance was not improved even when a single fast device was used.

2.2.3 CPU usage

CPU usage of almost 1,000% was needed (which means that 10 cores were used on the 32 core machine) in the data-plane case although the number of vCPUs was six. This result was confirmed by the observation that the a very large amount of CPU was consumed for I/O requests to be issued in host OS as well as a VM. There was no case where bare-metal performance was reached for multiple devices and a single one, while using the CPU minimally.

2.3 Causes of the Degrations

2.3.1 Serialization of the process in the guest and host OSes

Only one I/O thread per VM is traditionally allocated in KVM, and therefore an I/O thread is shared in a single VM. Its capacity is not sufficient to reach the maximum I/O performance achievable for multiple storage devices. Even though it requires larger capacity, its I/O performance is limited to a single I/O thread at maximum due to the internal structure of QEMU that lacks scalability. To address this problem, the data plane [11] is implemented in KVM, where an I/O thread runs per device. However it is still problematic in that parallelism is not considered because a single I/O thread handles both the I/O request and completion. Moreover, since it is in QEMU itself, only QEMU is taken into account, and thus it has the drawback that the VM overhead is not considered. There is also a parallelism issue for the multi-core architecture. As mentioned in splitX [12], each layer needs to run independently in order to fully utilize the multi-core parallelism, but KVM cannot currently run in this way. In order to maximize the I/O performance, not only multiple vCPUs are required for ensuring parallelism, but also the process in the I/O layer in KVM/QEMU needs to be changed to allow for parallel processing.

2.3.2 Execution of OSes ignoring the processor affinity

The I/O performance for a VM is degraded mainly due to context switches between the guest and host OSes, which are called exits. The previous studies [7, 13] showed that exits incur a large portion of the overhead in the VM operation when switches occur between the guest and host OSes. As mentioned in ELI [7], three exits occur in KVM. The first exit occurs to notify QEMU of issuing an I/O request, the second one occurs in order to inject a virtual interrupt for completed I/O, and the third one occurs due to End Of Interrupt (EOI) in the VM for accessing the Advanced Programmable Interrupt Controller (APIC). These operations are problematic because not only such an operation itself incurs overhead but also it is intended to stop the guest operation and resume the host operation, which hampers efficient parallel processing. It is notable that a relatively small number of exits occur in the case of batch operations. On the other hand, even an exit cannot be overlooked in the direct-access case, which requires quick response. It is thus necessary to alleviate the exit overhead in order to achieve the bare-metal performance. The overall process in the I/O layer is divided into two parts: host and QEMU I/Os. They are pinned

to different CPUs in a NUMA (Non-Uniform Memory Access) machine in order to limit the number of unnecessary switches between the CPUs to one as the processor affinity is very crucial in the NUMA architecture with NVM-based storage devices.

2.3.3 Duplication of the I/O processing stack

Fig. 1 illustrates that the length of I/O flow is almost doubled because there are an additional OS and an application. This problem may be solved by removing the unnecessary duplicated parts in the I/O layer, including the deletion of the I/O scheduler in the guest block layer and the transmission of guest's bio to the host block layer directly. But this solution has the drawback that VMs cannot use file-based storage since it is necessary to connect physical devices to the guest OS directly. The guest application that requests I/O will be scheduled when the I/O is completed in host OS but the delay in I/O in the VM environment is much longer than the time it takes to perform I/O in host OS because the guest application is assigned to a vCPU process that is in charge of emulating a physical CPU first, and then the process scheduler in guest OS schedules the guest application.

We have addressed the above mentioned performance problems for file-based storage in NVM-based paravirtual systems since file-based storage is widely used in datacenters even though SR-IOV [14] which is connected directly to a VM achieves good performance. Such storage is preferred because it is easy to use software techniques such as migration and copy on write. We have focused on random read workload for several reasons even though we have also considered random write workload. First, read is more sensitive with respect to the time of response from device than write. Also high read performance is often required for real workload. Second, to reduce the context switches between the guest and host OSes, we need to exclude other factors that cause performance degradation; for example, there is some overhead incurred by garbage collection in SSD in the case of random direct write. Third, the I/O flow of write is basically the same as that of read in the case of direct-I/O mode. Thus if we improve the flow of read, we should be able to enhance the write performance by taking the same approach for improvement; as expected, the performance random write is also improved in the same way, which will be shown in Section 6.

3. PERFORMANCE OPTIMIZATION METHODOLOGY

We have developed a methodology to enhance the low performance of NVM-based paravirtual systems: the use of polling rather than interrupt and parallel batching in order to maximize the parallelism in performing the sequence of I/O operations, and avoidance of context switches in order to consider the processor affinity.

3.1 Maximizing the Parallelism in I/O Operations

3.1.1 Polling

Injecting an interrupt is the traditional method for notification of I/O completion. However it has disadvantages such as latency and fit only for sequential I/O which is based on batching. Since NVM- based storage shows high random I/O performance, we need to reexamine and consider using polling which may replace interrupt. Polling was also used in an optimization study on an on-board SATA controller [6]. Not only on-board SATA controllers which are based on a single interrupt but also RAID controllers which are based on multiple interrupts (MSI-X interrupt) can perform better than interrupt-based

ones. This result implies that the polling-based method may fit well the current fast storage devices.

3.1.2 Batching

An SSD can process multiple I/O requests at the same time. Thus issuing I/O requests directly without batching can be beneficial. But issuing a single I/O request directly has a bad effect if other I/O requests are issued at different times within a short interval because the lock contention occurs due to the shared data structure leading to the waste of CPU cycles. Therefore temporal merge [15] which permits issuing I/O requests in a batch without any alignment in a short interval may be a good approach in the case of using NVM-based devices. In our methodology, polling rather than injecting an interrupt is used to send a notification and notified multiple events are processed in a batch, and exitless polling allows for non-serialization of many I/O operations so that as many non-serialized operations can be overlapped in a pipelined fashion as possible.

3.2 Considering the Processor Affinity

3.2.1 Exitless

Each of guest and host needs its I/O module in order to communicate with each other in a paravirtual environment. Mode switching such as context switching between guest and host, and notifications for requesting I/O and completing it are also required. For example, in KVM on an x86 machine, VM entries and exits are needed for mode switching and virtio is used for communication. A VM performs more context switches than the host for the reason above described. Context switching causes high latency, and thus I/O cannot be performed fast. We need to devise an I/O processing mechanism without context switching.

3.2.2 Processor and vCPU affinity

CPUs of VM can represent threads in host OS. A thread is a minimal scheduling entity. So allocating each of the vCPUs with CPU affinity is advantageous. This is true for processes in a VM. Processes on a VM have more effect if it is scheduled in another vCPU in the case of showing high IOPS. In our methodology, the guest and host OSes are run on different CPUs in a NUMA system in order to avoid unnecessary context switches considering the processor affinity.

In principle, the methodology should be hardware independent [10], based only on software without requiring special permissions not causing any security problem, affecting only the minimal part of the system [7, 8], and leading to high throughput and low latency without wasting CPU cycles. The CPU utilization can be lowered by decreasing the lock contention in the execution of exits [10].

4. DESIGN AND IMPLEMENTAION OF THE METHODOLOGY FOR KVM

We have applied the hardware-independent methodology to KVM/QEMU in order to demonstrate how to employ it to a given paravirtual I/O system. Polling rather than injecting an interrupt is used to send a notification and notified multiple events are processed in a batch so that as many non-serialized operations can be overlapped in a pipelined fashion as possible. Also, host I/O and QEMU I/O are performed in different CPUs to avoid unnecessary context switches considering the processor affinity.

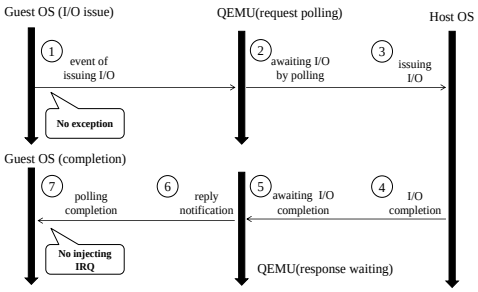


Fig. 4. Sequence of pipelined polling.

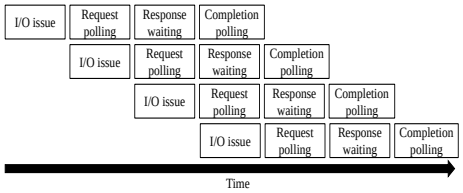


Fig. 5. Pipelined polling.

4.1 Exitless Polling for Overlapping the I/O Process in a Pipelined Fashion

In a KVM/QEMU system, the I/O process is performed as follows: when a guest application issues an I/O request, the virtio block driver queues this request and notifies QEMU of it by executing an exit. By employing our methodology as illustrated in Fig. 4, the driver does not notify QEMU without performing any exit. Instead, the request polling thread in QEMU polls the memory space which is shared with the guest OS in order to check I/O requests from the guest. If there are any queued I/O requests from the guest, they are sent to the host OS, and a response waiting thread may be blocked waiting for the I/O completion. If there are many I/O requests in Guest OS, the request polling thread in QEMU checks I/O request via polling. Otherwise if there are few request in Guest OS, the request polling thread works as interrupt based.

When the requested I/O is completed, the response waiting thread that has waited for the I/O completion wakes up and then notifies the guest of the I/O completion to the guest by writing the memory area shared by the guest and host using virtio. Conventionally, the guest is notified via an interrupt. Instead, the dedicated thread in the guest checks the I/O completion via removing an exit and polling, and completes I/O, in applying our methodology. As a result, the I/O process consisting of guest's I/O request, QEMU's request polling, QEMU's response waiting, and guest's completion polling is performed independently; one part does not depend on another, which makes it possible to perform them in parallel leading to high performance.

The process of direct-access I/O is not serialized by removing exits and using polling instead because each process can be performed independently. An exit is required when notifying an I/O request or I/O completion in the case of taking the traditional exit approach. When an exit is removed, there is no need for waiting for one in the I/O process, and the process can be divided independently so that it can be performed on multiple CPU cores concurrently in a pipelined fashion as shown in Fig. 5. It is performed efficiently not only in a single I/O thread because of pipelining but also in multiple I/O threads due to its structure which permits being executed in parallel. The I/O process may be overlapped per device via exitless polling in order to achieve high performance. Alternatively, it is overlapped for multiple devices as long as the performance is not degraded, leading to higher CPU utilization.

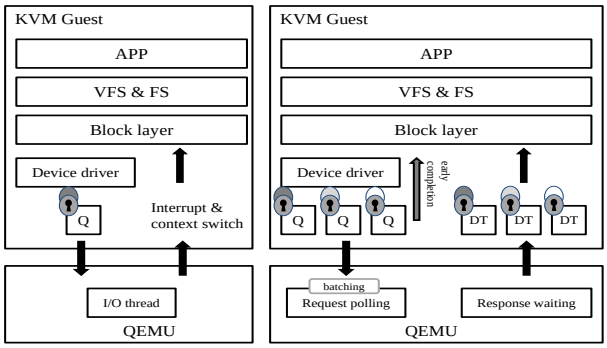
There are several reasons why overlapping the I/O process based on exitless polling may lead to higher throughput saving CPU cycles in the case of using multiple NVM-based storage devices. First, the performance of processing data via interrupts will be significantly degraded as future storage devices become faster, and if faster I/O is needed, the polling as provided via NAPI (New API) in Linux kernel may be more beneficial than

interrupts for I/O intensive applications because the interrupt overhead can be eliminated and the cost of context switch may be reduced, as shown in the previous studies [16] [15]. Second, the use of polling permits removing exits, making the methodology hardware independent. The use requires only monitoring the memory space shared by the host and guest without requiring any hardware support such as posted interrupt [17] unless an additional exit is executed, or any modification of the existing software because it is sufficient to use just virtio which is already available for communication between KVM and QEMU. Third, it is possible to lower the CPU usage even in the case of polling compared with that of interrupts because the overhead of exits is not incurred and the lock contention in KVM is reduced by avoiding control via APIC. It is found in a previous study [10] that 40% of the total CPU cycles are used for the data plane and most of the CPU cycles are consumed in the execution of the spin lock in KVM in the case of using KVM with the data plane. It is also found that most of the lock contention occurs in making APIC related accesses such as the invocations of *kvm_ioapic_set_irq()* and *kvm_ioapic_update_eoi()*. It is possible to further lower the CPU utilization if this contention is reduced, but it is better to bypass exits than modifying the lock structure to reduce the contention because the modification does not result in the elimination of the contention.

Exitless polling is implemented by removing VM entries and VM exits [18]. As described in the previous section, three VM exits may occur in single I/O process. Among them, we try to delete two VM exits which are one that occurs injecting an interrupt in order to notify the guest of I/O completion, and the other that occurs when the guest writes to APIC in order to process the EOI. When a VM issues an I/O request, the host OS handles the request eventually. When the I/O is completed, QEMU that executes the VM by executing the while loop that is in charge of emulation is interrupted, and then injects an interrupt to the VM. When QEMU receives the request of injecting an interrupt from the host OS, QEMU stops execution of the emulation loop and confirms the exit code.

If QEMU stops executing the emulation loop due to I/O completion, it injects an interrupt to guest's IDT (Interrupt Descriptor Table). The existing code is revised as follows in order to remove the context switch to the host to manage interrupts in the context of VM operation: there is a callback function which sends the information on I/O completion to KVM; when the I/O request from QEMU is completed, the callback function is invoked. This callback function is in charge of delivering the information on I/O completion and the request of injecting an interrupt. The callback function is removed and replaced by the polling thread. It is thus possible to remove the exit which was caused by a context switch between guest and host OSes. Also, the VM entries are removed because there is no need for entering the guest context because the guest can recognize I/O completion without performing an exit. In guest OS, the traditional completion scheme that is based on interrupt is removed. Also, exitless polling is implemented in order to reduce the latency by recognizing I/O completion without injecting an interrupt. In the case of the VM exit which is related to EOI, the VM exit is needed because EOI requires accessing APIC and writing the completion flag. Accessing APIC is a privileged operation, and thus the guest OS should be context switched out with the host OS is switched in, and then KVM in host OS handles the APIC control. This exit-based method is replaced by the polling thread-based one, and thus the guest OS can process I/O completion without EOI because any interrupt is not being registered or used.

Also, there is an exit for issuing an I/O request from the guest OS because the virtio blk driver in guest OS uses the *io_write()* in the last stage of I/O request in guest OS. At this point of time, the guest OS enqueues data to a virqueue that can be shared with



a) Original b) Multiple issues and completions

Fig. 6. Multiple issues and completions (Q: virtqueue in the virtio blk driver & DT: dedicated thread for completion).

QEMU and then calls the *io_write()* in order to notify QEMU that there is an I/O request. The *io_write()* is a privileged one which can perform communication between guest and host OSes by accessing a specific memory region. When the guest OS uses the *io_write()*, QEMU performs the exit emulation loop and then checks the exit code. Therefore, if the *io_write()* in guest OS is removed and just the shared memory for QEMU is monitored, then the I/O request is processed if the queue is not empty, and thus the exit can be removed.

4.1.1 Processing multiple notification events in a batch and in parallel

Although the use of exitless polling leads to improvements in throughput and CPU utilization (saving CPU cycles) in the case of using multiple storage devices and a single virtual device in guest as explained in the previous subsection, the performance can be further enhanced for a single device such as an NVMe-based SSD showing more than 700K IOPS. It is possible to improve both the I/O layer in QEMU and the block layer including the device driver in guest OS by making them process multiple events in a batch or in parallel. For this parallel batching, the following methods based on exitless polling are devised and implemented: first, I/O requests are issued as fast as possible to fully utilize the fast characteristic of NVM-based SSDs. Second, the lock contention is reduced for parallel processing because the contention leads to high CPU consumption and low throughput. Third, multiple instances of I/O completion are handled to reduce the latency, using the dedicated thread rather than interrupts and considering the processor affinity, although such instances are not handled by multiple CPUs as in the MSI-x interrupt.

We have implemented the parallel batching by considering the followings as shown in Fig. 6: first, since the virtio block driver in the guest uses one queue for issuing I/O requests to the host, and the guest driver layer based on a single queue and single interrupt can achieve only the I/O performance for a single CPU even if VFS and the block layer in guest OS can perform I/O in parallel, multiple queues can be used for issuing I/O requests and multiple dedicated threads can be used for checking I/O completion. Second, the early completion functionality has been added to check I/O completion in the issue context whenever issuing an I/O request, and to process it if there is any completed request. A lock should be held when the guest issues an I/O request, and thus it is more efficient to process it in the issue context if there is any completed request because the overhead incurred by the lock contention and completion thread can be reduced. Third, a lock per queue is used

in order to process I/O requests in parallel when the guest issues them and has the I/O completed. Storage devices presented at the user level operate using each request queue allocated on a per device basis in Linux. The performance may be degraded in the case of using NVM-based SSDs with high IOPS and parallelism because a queue lock should be held when the guest issues I/O requests and has them completed through this queue. Thus, the guest has been changed to have the parallel structure as suggested in a study [19] in order to reduce the dependency on lock and permit parallel processing. Fourth, batch processing performed based on a threshold value when QEMU checks whether there is an I/O request from the guest leads to reduction in the exit overhead. Since an SSD consists of a number of channels, it can handle multiple I/O requests concurrently. In order to take full advantage of this characteristic, not only I/O requests are issued as fast as possible but also multiple I/O requests are issued at once in order to maximize the performance with reduced cost of context switch from the host OS. Finally, the thread polling I/O completion is modified by adding a peek function executed to avoid holding any unnecessary lock during the polling process unless there is any completed request in the queue.

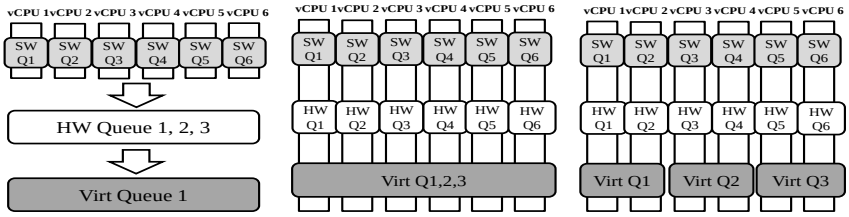
Batch processing is implemented by using the traditional method based on virtio for virtqueues. The number of virtqueues is increased in the modified guest OS and QEMU. In guest OS, an I/O request is issued in a round robin fashion with queues for a single storage device. Dedicated threads are also constructed in order to process I/O completion. In QEMU, for each of the queues, a response waiting thread is created. Early completion is performed as follows: when an I/O request is issued in guest OS, an added routine for checking I/O completion and performing I/O is run after issuing I/O. There is an advantage of using early completion. Early completion permits sharing the jobs of completion in the I/O issue context. Thus, the dedicated thread can consume less CPU cycles, pinned onto a single core. This leads to reduction in CPU usage for polling and enhancement in locality.

4.1.2 Ensuring the processor affinity

The overall I/O layer for KVM consists of host and QEMU I/Os. They are pinned to different CPUs on a NUMA machine. The number of unnecessary context switches between the CPUs is limited to one. Ensuring the processor affinity of VMs, I/O threads or completion threads are very crucial in the NUMA architecture because different amounts of overhead are incurred depending on the location of memory node [20]. This overhead is more important in the case of using NVM-based SSDs with high IOPS because the maximum capacity of the system cannot be fully utilized due to the overhead such as that of context switch or memory copy operation. In a KVM system, the vCPU and the I/O threads of QEMU switch their CPUs depending on the system load, which is shown in a previous study [10]. Therefore, this may be problematic regarding the cache locality and delay.

5. DESIGN ISSUES FOR KVM

As a result of applying our methodology to KVM, the performance is improved as explained in the previous section. However, the same level of performance cannot be achieved for NVMe-based SSDs or Ramdisk as that of using only the host OS. Specially, the performance result after using the methodology is not linear in the case of using a single device even though parallel processing is perform as a result for enhancing the performance of the single device. Low performance in the case of using a single device



(a) Linux multi queue block layer with virtio blk driver

(b) Round-Robin multi queue block layer with virtio_blk dirver

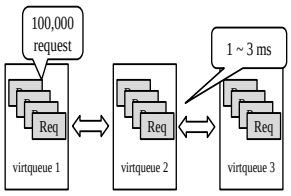
(c) Pinning multi queue block layer with virtio_blk dirver

Fig. 7. Comparison between default multi queue and tunned multi queue.

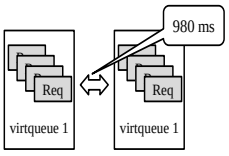
comes from: 1) parallel batching with the Linux multiple queue block layer; and 2) the scalability problem of the host file system. Thus we have extended the methodology to further consider the following characteristics observed from our experiments for validating the methodology: first, the performance of multiple devices has been higher than that of a single device. The performance sum of three devices having exitless polling applied polling with a single thread with 3 devices in Fig. 16 (b) is better than that of a single device having exitless polling applied with three queues for fully parallel processing with parallel batching in Fig. 15. From a logical point of view, it is expected that the cases described above should show similar results, but their results are different in practice. Second, parallel batch processing for a single device leads to a result of performance improvement which is not completely linear.

As means of analyzing this problem, a NULL test is performed in order to confirm the performance in the single device case. In the NULL test, I/O is not performed to a real device but I/O is completed by returning TRUE on a specific layer. Therefore, the application which issued an I/O request immediately recognizes that I/O is completed successfully. The reason why the NULL test is performed is that some overhead can be found as our methodology makes the current I/O structure in guest OS and QEMU parallel but does not make any other components for I/O better, such as VFS, file system and host OS. Thus the goal is to find out the source of the overhead in the test. First, the test is performed to the request function in the Linux block layer in order to confirm the maximum performance of guest OS. The result of the test shows that 900K IOPS can be achieved. It may be concluded that 900K IOPS is the maximum performance which the guest OS can achieve. We thought that it is possible to find out whether there is a problem in the upper layer on the Linux block layer or in the lower layer below the Linux block layer by comparing to the maximum performance, those for a single and multiple devices with the methodology applied. But the test results for a single device and multiple devices are almost the same, which means that there is no difference from the application layer to the block layer. This is because the multiple-queue block layer on the Linux is performed in parallel being well optimized. It is found that the performance problem is caused by the bottom layer in the block layer, which indicates that our methodology missed something important.

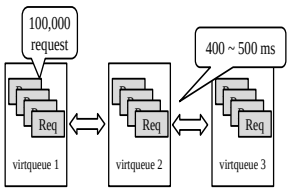
This problem was investigated focusing on the use of multiple queues for parallel batching. In parallel batching with multiple queues, I/O requests are issued by Round-Robin (RR) scheduling as shown in Fig. 7 (b); for example, the first I/O request is issued onto the first queue, the second one is issued onto the second queue, and so on. But we thought that RR scheduling conflicts with the upper layer such as the multiple queue



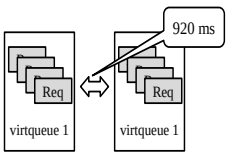
(a) Completion time on each of queues with RR policy



(b) Completion time on a single queue with RR policy



(c) Completion time on each of queues with queue pinning policy



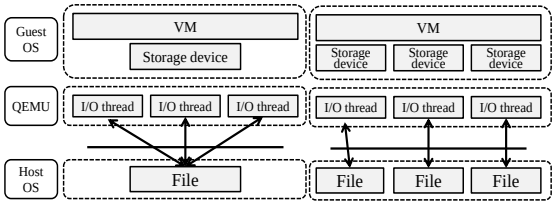
(d) Completion time on a single queue with queue pinning policy

Fig. 8. Comparison between RR policy and pinning one.

block layer for the following reasons: as shown in Fig. 7 (a), the multiple queue block layer has software queues in order to fully utilize the multiple cores. For instance, each software queue is assigned to each of the CPU cores. Hardware queues are assigned in order to support the device driver which has many internal queues. This implementation is beneficial for parallel operations because each of the cores has a software queue with locality due to pinning. I/O requests are issued by taking such advantage from the upper layer, and thus the device driver which has internal queues can be fully utilized.

In the NULL test, it was found that the overhead of block layer is incurred by the lower layer. We first thought that omitting the existing layers would be helpful. However, after checking the multi-queue results, we finally decided to fully utilize the existing layers.

In parallel batching, the method of utilizing software/hardware queues in the multi-queue patch is employed when issuing I/O requests. As mentioned above, in parallel batching, multiple queues are used by RR scheduling because this permits using multiple queues in parallel. This works in the case of using NVM-based SSDs because such SSDs permit processing multiple I/O requests at a time without any seek operations. Fig. 8 shows the total processing time for 100,000 requests in a queue. Because it takes less than 3ms in each queue, the utilization of each queue is relatively fair. But the RR scheme can cause the problem that the I/O request can be issued to the irrelevant device queue of another core whereas it leads to utilization of multiple queues. To avoid having the problem, a software queue is pinned to each hardware queue in the upper layer as shown in Fig. 7 (c). This figure illustrates the modified design of software queues in the case of using our extended methodology. In this design, the request processing time in each queue, is increased up to 500ms, but it takes only less time to process 100,000 requests in one queue, that is, a decrease from 980ms to 920ms. This result implies that increasing the utilization of each queue by reducing the cost of context switch and improving the locality via pinning software queues can achieve performance enhancement even though it may lead to unfair utilization of queues. We conducted the same random read tests with



(a) Single device case (b) Multiple device case

Fig. 9. Comparison between a single device and multiple devices with file-based storage.

pinned software queues, and in the experiment illustrated by Fig. 7 (c), we obtained 300K IOPS, which corresponds to a 10% improvement.

The performance of the single device case was improved, but the ideal performance was not yet achieved, and the performance of the multiple device case could be higher than that of the single device case. The performance of single device optimization with three queues and three threads is the same as that of multiple device optimization with a single queue and a thread. But the performance result for a single device is 300K IOPS and that for multiple devices is 350K IOPS. The result of the ideal case is 380K IOPS (in the iodepth test with three processes in host OS). The single device environment differs from that using multiple devices in guest OS, and the difference is illustrated in Fig. 9. As can be seen in the figure, the use of a single device means that the host file system has several I/O threads make requests to a single file. In contrast to this case of using a single device, the use of multiple devices means that the host file system has several I/O threads and files, and each thread makes requests to a different file. The performance is expected to be degraded when I/O threads make multiple requests to a single device due to a scalability problem. We thus conducted an experiment with the fio micro benchmark to confirm the existence of this problem. To simulate the single device, we conducted random read tests with a single file, three processes and 32 iodepth resulting in 310K IOPS. Compared with the case of using a single device, we ran the benchmark with three files, three processes and 32 iodepth, leading to 380K IOPS. This result tells us that there is a scalability problem in the ext4 file system, because the performance with three devices to a single file is less than the performance with three devices to each with its own files. If this scalability problem can be eliminated, the better performance can be achieved.

6. EVALUATION

To demonstrate the effectiveness of applying our methodology to an NVM-based paravirtual I/O system, we conducted experiments to measure the I/O performance of a KVM system using a flash memory-based SSD, Ramdisk [10] and NVMe-based SSD as such an exemplar system in the following environment: an Inter Xeon(R) E5- 2690 with 2.90GHz 2 CPUs (with 16 cores each) with 128GB RAM was used. The host OS was ubuntu version 12.04, the Linux kernel version was 3.2.0, the guest OS kernel versions were 3.5.0 rc7 & 3.15.0, the QEMU version was 1.6.2, the number of vCPUs was six with a single VM based on each, and the ext4 file system was used in both the host and guest OSes.

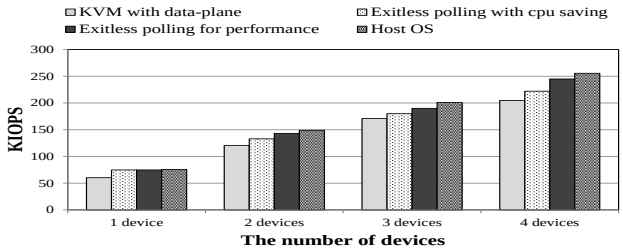


Fig. 10. IOPS results on multiple SSD devices in direct-access tests (with 32 fio processes per device, performing 4KB random reads).

6.1 Flash Memory-based SSDs

A flash memory-based SSD, a Samsung 845DC evo, the capacity of which is 960GB with SATA3 was used in a test with LSI megaraid 9361-8i.

6.1.1 Performance and CPU usage for multiple devices

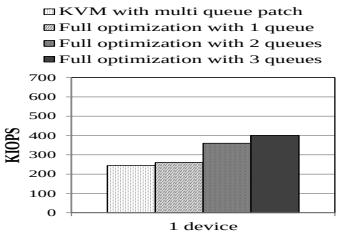
Fig. 10 shows four results of enhancing the data plane which leads to the best performance among the existing solutions; exitless polling, a high performance version without saving CPU cycles, exitless polling with CPU saving, and host OS which corresponds to the ideal case.

Performance for multiple devices The performance in the CPU saving case was measured using a request polling thread in QEMU used for two devices, which means that a single request polling thread runs for two devices. In the case of using the high performance version, a request polling thread and a reply polling thread were run for each of the devices. As a result, this version performed almost the same (240K IOPS) as host OS (250K IOPS). Also, the performance of the CPU saving case was improved nearly by 10% compared with the data-plane case.

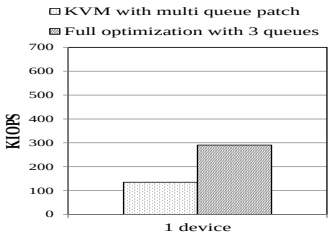
CPU usage for multiple devices As previously mentioned, CPU usage of 1,000% was measured for random I/O in the data-plane case. The usage was reduced to 750% in the CPU saving case, which means CPU usage of nearly 150% was needed for I/O processing. Since the number of vCPUs was six, the maximum CPU usage of a single VM was 600%. Also CPU usage of 850% was shown in the case of using the high performance version. This indicates that the percentages in CPU usage decreased w.r.t. the maximum CPU usage were 25% and 15% for the cases as shown in Table 1. It is expected that the

Table 1. CPU usage for SSDs (4 deivces case shown in Fig. 10.)

	Max CPU utilization	Min CPU utilization
Data-plane	1,000%	950%
Pipelined polling (Performance)	850%	820%
Pipelined polling (CPU saving)	750%	710%



(a) IOPS in the direct-access, aio test (with 4 fio processes in total, performing 4KB random reads, when iodepth is 32 per process)



(b) IOPS in the direct-access test (with 128 fio processes in total, performing 4KB random reads)

Fig. 11. Performance of a single device on ramdisk. (KVM with multiqueue patch: default KVM performance with multiqueue patch in guest OS (data-plane + multiqueue patch). Full optimization with 1 to 3 queues and a dedicated thread: our approach is taken.)

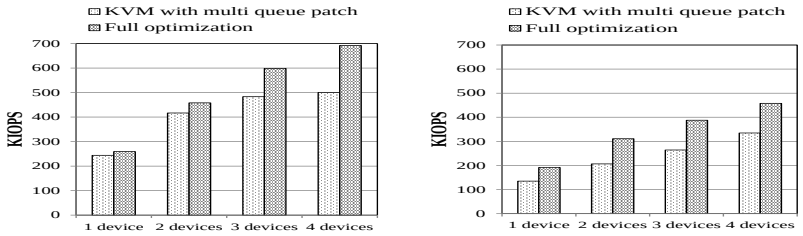
performance will be further enhanced if many devices are connected to the system, causing congestion because the lock contention in KVM is significantly reduced. Also, CPU consumption regarding polling is reduced because the response waiting thread in QEMU consumes CPU cycles only when computation is needed in that it is blocked waiting for an I/O event.

The request polling thread periodically checks I/O requests from the guest OS. Thus it might need to consume CPU cycles more than any exiting solution. However it uses busy polling only if there are many I/O requests in the I/O queue in the request thread; otherwise, it comes to sleep. For this reason, it does not incur high overhead. The completion polling thread in guest OS periodically checks I/O completion using the peek function, but in our methodology, the completion polling thread becomes active only when receiving an I/ O request in order to avoid incurring high overhead.

6.2 Ramdisk

6.2.1 Maximum performance with a single device

A test with a ramdisk was performed to check the maximum performance in the case of using a single fast NVM-based storage device. First, the use of our methodology led to much higher performance than any other existing solution mainly by applying exitless polling. The performance for a single device previously mentioned (65K IOPS) was improved up to 140K IOPS. But this was still far lower than that in the RAID-0 case (about 250K) where four SSD devices were connected or in the case of using a single NVMe SSD. Lack of scalability in the block layer in guest OS might cause this problem. The use of our methodology led to almost the same result compared with the case where only exitless polling was used. The I/O layer in guest OS except for the device driver cannot currently issue I/O requests as fast as possible even if fast processing is needed in order to achieve the maximum hardware performance. The problem of scalability in the Linux block layer is solved by taking the Linux multi-queue approach [19]. This was tested in guest OS version 3.15 with our methodology applied, in order to confirm performance improvement without incurring the overhead of the guest block layer. As a result, the base performance was improved achieving up to 150K IOPS. When exitless polling was applied, it reached up to 250K IOPS. When parallel batching in addition to exitless polling was employed, it reached up to 280K IOPS. However this result was not a desirable linear



(a) IOPS in the direct-access, aio test (with 4 fio processes in total, performing 4KB random reads, when iodepth is 32 per process)

(b) IOPS in the direct-access test (with 32 fio processes per device, performing 4KB random reads

Fig. 12. Performance of multiple devices on ramdisk.

one. We performed experiments after changing the options of the fio benchmark including iodepth from 1 to 32 and the number of processes from 4 to 128 because a large number of threads would incur context switching overhead in process scheduling. As shown in Fig. 11, the use of parallel batching led to improvements in the I/O performance by more than 50% compared with the baseline case after changing the fio benchmark options. But there was some performance limitation as shown in Fig. 11 (a); although more queues and completion threads were added, it would not be possible to achieve over 400K IOPS, which is not the maximum performance for a single VM because the use of multiple devices can lead to more than 600K IOPS and the ideal performance is 900K according to the pervious simulation study.

6.2.2 Performance and CPU utilization for multiple devices

Fig. 12 shows the result of applying our methodology compared with the data-plane case for multiple queues; in our methodology, only the threads for parallel batching were not used but the function for parallel batching was used. The figure shows that the performance could be improved when an additional device was connected. Also the performance was enhanced in the iodepth test where less "exits" occurred. On the contrary, in the case of using the existing method, the more exits and lock contention occurred, the more devices were connected, because of accessing APIC and performing internal KVM operations in KVM compared with our methodology. However our methodology does not cause this problem by performing exitless polling. The more devices are attached, the higher performance is obtained. 50 to 100% more CPU usage is required than any existing approach because busy polling is used for fully utilizing the disk bandwidth.

6.3 NVMe Device

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

6.3.1 Performance of random read

We tested the NVMe device in host OS for checking the maximum performance of random read. As shown in Figs. 13 and 14, the maximum NVMe performance was not achieved in the case of a single thread or a small number of iodepth. In the case of direct access with 128 threads, the maximum performance of device was reached, 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 maximum NVMe device performance could not be achieved due to the serialization of I/O operations. In order to reach the maximum

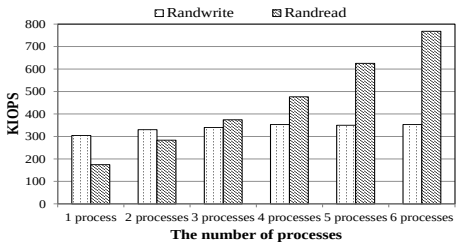


Fig. 13. IOPS for an NVMe SSD in the direct-access, aio test (with fio process(es), processing 4KB requests when iodepth is 32 per process).

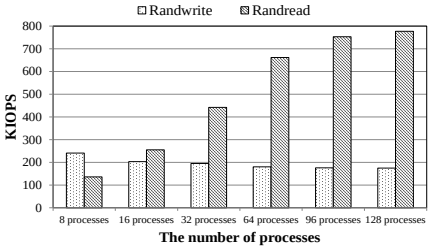


Fig. 14. IOPS for an NVMe SSD in the direct-access test (with fio processes, processing 4KB requests).

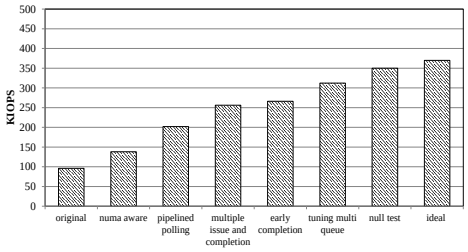
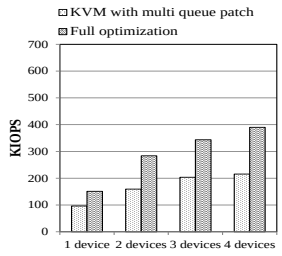


Fig. 15. IOPS for an NVMe SSD in the direct-access test (with 128 fio processes, performing 4KB random reads) on a single virtual storage device.

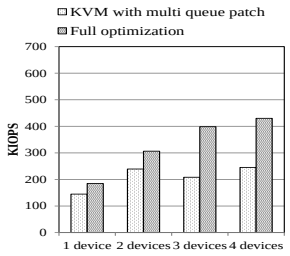
NVMe performance, I/O needs to be performed in parallel. If we want to fully utilize an NVMe SSD, we should simultaneously perform more than six I/Os with 32 as iodepth or 96 I/O threads.

6.3.2 Performance for a single device

A test was conducted in order to measure the performance for a single virtual storage device in guest OS using the NVMe device. Fig. 15 shows the result of applying our methodology and using the current technologies in the case of running fio. Each result in the graph includes that next to in the left; for example, the result of exitless polling means that of affinity awareness plus exitless 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 Fig. 15 indicates that about 95K IOPS was obtained without any components of the methodology applied. The affinity awareness test was performed with pinning vCPUs and batching memory with consideration of the processor affinity in the NUMA architecture; as a result, 135K IOPS was obtained. The exitless polling result was 200K IOPS which was obtained while affinity awareness was applied. Parallel batching with three queues led to 250K IOPS, but did not achieve a linear improvement that we had expected. The result of applying early completion and other components of the methodology shows improvements but does not show a big difference because I/O was performed faster when the NULL test was conducted, without causing congestion, and thus early completion made no big impact. In the test, QEMU immediately returned with in the I/O process. This means that the other I/O processing in host was eliminated. There was no big performance improvement even though parallel batching can lead to processing three I/O requests concurrently, because there are three queues and a completion thread.



(a) IOPS in the direct-access test (with 32 fio processes per device, performing 4KB random reads)



(b) IOPS in the direct-access, aio test (with 4 fio processes, performing 4KB random reads when iodepth is 32 per process)

Fig. 16. IOPS for an NVMe SSD in the direct-access, aio test on multiple virtual storage devices; our approach is taken except for the use of multiple queues and a dedicated thread to increase the cpu utilization.

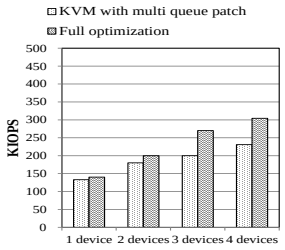
But the result of tuning multiple queues via pinning a queue per vCPU as described in Section 5 can achieve good performance. The ideal result is the same as that of using three threads and 32 as iodepth with the NVMe device because we implemented parallel batching based on the aio interface.

6.3.3 Performance for multiple devices

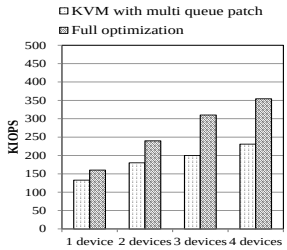
In this test, we used only one partition in an NVMe SSD. Then we created four files in the raw format, and each of the files is connected to the guest as a virtual storage node in order to provide four storage devices in the VM. We set the numbers of queues and dedicated threads to ones. We assumed that the use of four processes leads to 500K IOPS which is the ideal result as shown in Fig. 13 since the numbers of devices and I/O threads are fours. Fig. 16 shows the result. The performance in the original case is improved when more devices are attached and used. But the degree of performance improvement becomes lower in the data plane. On the contrary, compared with our methodology, the performance gain is small in the case where the number of attached devices is small, but the more devices are connected to the VM, the more the performance is improved; when four devices are connected, the performance is improved by about 80% compared with the original case. Note that in the case of making and using multiple file-based storage devices from a single physical device, in a VM, the performance in this case is higher compared with that for a single storage device with multiple thread connections. This means that a VM has a large amount of I/O processing capability but that single VM storage will suffer from a scalability problem when high IOPS is required.

6.3.4 Random write

Normally, the performance of random write to an NVMe SSD depends on the state of the SSD. For instance, the clean state which means that the SSD is not dirty can lead to the maximum performance of the SSD because there is no need for erasure and garbage collection. The state of sustain means that data is written to most of the flash memory of the SSD, and thus there is need for erasure before write or garbage collection. The following three states of SSD are defined: Clean, Partially-clean and Sustain. To a manufacturer such as Samsung, the performance in the sustain state is the base of write performance. But the random write performance in the sustain state is approximately 100 to 120K IOPS.



(a) IOPS in the direct-access test (with 32 fio processes per device, performing 4KB random reads)



(b) IOPS in the direct-access, aio test (with 4 fio processes, performing 4KB random reads when iodepth is 32 per process)

Fig. 17. Performance for multiple devices on an NVMe.

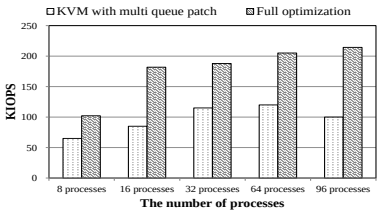


Fig. 18. IOPS for an NVMe SSD in the direct-access test (with fio processes performing 4KB random writes) on a single virtual storage device.

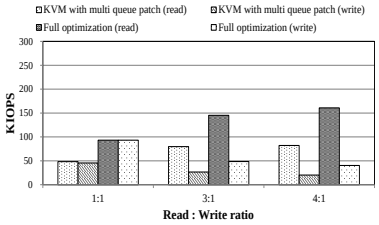


Fig. 19. IOPS in the direct-access test for mixed workload of random read/write.

This does not reflect only the effect of our methodology, and thus we focus on the clean or partially clean state.

The maximum performance in the clean state is 350K IOPS with Fig. 13 and Fig. 14. As shown as Fig. 17, the use of our methodology can lead to the performance for multiple devices up to 350K IOPS, and 210K IOPS can be achieved in the case of a single device as shown in Fig. 18 because the write performance for a single file in host OS is 260K IOPS. Compared with random read, random write can be more affected by the virtual environment due to journaling of the filesystem and synchronization of metadata.

6.3.5 Mixed Workload

Fig. 19 shows the performance of random read/write in running fio. This test has great importance because the real-world workloads have mixed-read/write I/O patterns and process I/O requests in parallel. The result shows that the performance is improved up to twice in terms of IOPS by applying our methodology.

6.3.6 Macrobenchmark

We ran the TPC-C workload on MySQL in a test. As shown in Fig. 20, the use of our methodology leads to further performance improvement. The I/O pattern of TPC-C workload is mixed-read/write. But the performance of the workload cannot reach the maximum IOPS as described on the mixed workload. For example, the use of our methodology can lead to the performance of reading 250,000 blocks per sec, which is shown as a result of running vmstat in host OS. But the execution of TPC-C workload can reach

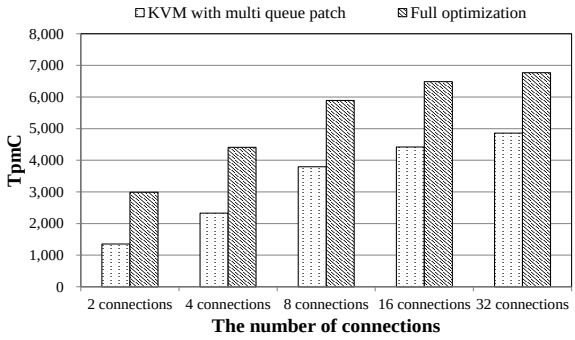


Fig. 20. IOPS in the direct-access test for TPC-C benchmark.

the performance of reading 180,000 blocks per sec even though the CPU is not fully utilized on the VM; which means if the execution of MySQL or TPC-C workload is further optimized, the use of our methodology can lead to higher performance.

6.3.7 CPU usage

We show that the high performance of random I/O requires high CPU usage as described in the SATA SSD case. CPU usage can be reduced by exitless polling. But fast storage such as NVMe SSDs which is much faster than SATA SSDs consumes slightly more CPU cycles by approximately 70 to 80% in the cases based on four devices. This is because: 1) high IOPS requires busy polling; and 2) there is an additional OS in the virtual environment, and thus the same operations should be performed twice; for example, a read in guest OS will be performed again in host OS. Pass-through can be a solution in order to reduce CPU usage, because pass-through permits accessing the device directly in guest OS. However we cannot use file-based storage if pass-through is used. To address this problem, I/O can be split by pass-through or non-pass-through, which was proposed in a recent study [21]; reads can be processed by pass-through while writes are performed in the existing I/O stack.

7. RELATED WORK

There was a study to optimize the host block layer to fully utilize SSDs [22]. In this work, the authors introduced a request queue incurring the block layer overhead, and attempted to delete it. The authors of the paper [19] proposed a new request queue for the multi-core architecture by splitting the queue and reducing the lock contention. We agree on the fact that the current request queue incurs overhead for SSDs. On the other hand, the authors of the papers [6, 19, 22] took host OS-based approaches. It is necessary to minimize the work for queueing and scheduling in the guest context. The authors of the paper [23] proposed an enhanced request queue by temporal merging. We believe that this approach can also be effective in the guest block layer; the guest block layer should be optimized by using a queue which is used for communication between the guest and host.

The authors of the paper [24] explained the problem that scheduling delay in a VMM (VM Monitor) leads to delay in the I/O process in the VM. This problem is serious in an environment where low latency is required. 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 a VM is thus different from the I/O process in host. Scheduling delay in the VM is more important than that in host OS because the I/O process in QEMU decides whether to send I/O requests to the host OS or to send them back to the VM after it receives them from a VM for the first time. This problem is addressed by our methodology because the process that in charge of I/O with the methodology applied checks I/O requests by polling, which should be more effective by considering the processor affinity.

The authors of the paper [25] attempted to optimize networking I/O in VMs by taking a polling-based approach. This work however focused on networking I/O that is performed in a stack which is 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 while the storage stack was improved by using the existing methods, which might minimize changes in QEMU and the guest device driver.

8. CONCLUSIONS

We have devised and developed an architecture independent methodology to improve the performance of paravirtual I/O systems based on NVM devices, via exitless polling and parallel batching ensuring the processor affinity. We have demonstrated the effectiveness of this methodology by applying it to a KVM system using different NVM-based storage devices. Our experiments study suggests that the use of the methodology can lead to enhancements in throughput by up to 100% for workloads in mixed read/write patterns being widely observed in practice. We believe that the methodology will be effective for such paravirtual I/O systems using not only the current but also future enhanced fast storage devices.

We are investigating the following issues to further extend methodology towards optimizing the performance of an NVM-based paravirtual I/O system: taking a hybrid approach that leads to low latency based on a combination of polling and interrupt for the VM block layer, dynamically changing s of I/O issue and completion threads depending on the varying need for the resources to improve the performance, designing and using low delay scheduling for I/O threads in guest, developing a comprehensive solution for issuing I/O requests as fast as possible, removing the guest I/O scheduler for fast storage considering the existence of the duplicated I/O scheduler in the host side, thus incurring overhead, and finally performing specific I/O by pass-through in order to reduce CPU usage.

ACKNOWLEDGMENT

This research was supported by 1) Institute for Information & communications Technology Promotion (IITP) Grant funded by the Korea government (MSIP) (R0190-16-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development). It was also partly supported by 2) the National Research Foundation (NRF) grant (NRF-2016M3C4A7952587, PF Class Heterogeneous High Performance Computer Development). In addition, this work was partly supported by 3)BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea(NRF) (21A20151113068).

REFERENCES

1. R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, Vol. 42, 2008, pp. 95-103.
2. VMWare Inc., "Esx server 2-architecture and performance implications," Technical Report, VMWare, 2005.
3. K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM Sigplan Notices*, Vol. 41, 2006, pp. 2–13.
4. K. Lee, D. Lee, and Y. I. Eom, "A paravirtualized file system for accelerating file i/o," in *Proceedings of IEEE International Conference on Big Data and Smart Computing*, 2014, pp. 309–313.
5. A. Huffman, "Nvm express specification 1.1, http://www.nvmexpress.org/specifications," 2013.
6. W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "Os i/o path optimizations for flash solid-state drives," in *Proceedings of USENIX Annual Technical Conference*, 2014, pp. 483–488.
7. A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: bare-metal performance for i/o virtualization," *ACM SIGPLAN Notices*, Vol. 47, 2012, pp. 411–422.
8. N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual i/o system." in *Proceedings of USENIX Annual Technical Conference*, 2013, pp. 231–242.
9. A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, Vol. 1, 2007, pp. 225–230.
10. M. Oh, H. Eom, and H. Y. Yeom, "Enhancing the i/o system for virtual machines using high performance ssds," in *Proceedings of IEEE International Performance Computing and Communications Conference*, 2014, pp. 1–8.
11. IBM Redhat, "Kvm virtualized i/o performance," Technical Report, 2013.
12. A. Landau, M. Ben-Yehuda, and A. Gordon, "Splitx: Split guest/hypervisor execution on multi-core," in *Proceedings of the 3rd Workshop on I/O Virtualization*, 2011.
13. H. Lv, Y. Dong, J. Duan, and K. Tian, "Virtualization challenges: a view from server consolidation perspective," in *ACM SIGPLAN Notices*, Vol. 47, 2012, pp. 15–26.
14. P. Primer, "An introduction to sr-iov technology," *Intel LAN Access Division, Revision*, Vol. 2, 2008.
15. Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, "Optimizing the block i/o subsystem for fast storage devices," *ACM Transactions on Computer Systems*, Vol. 32, 2014, p. 6.
16. J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt,," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012, p. 3.
17. J. Nakajima, "Enabling optimized interrupt/apic virtualization in kvm," in *KVM Forum*, 2012.
18. Intel, "Intel 64 and ia-32 architectures software developer manual," Technical Report, 2011.
19. M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th ACM International Systems and Storage Conference*, 2013, p. 22.

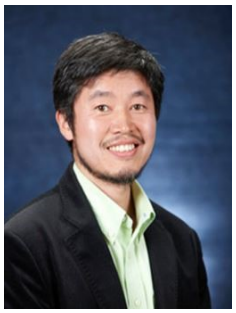
20. D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system iops on low-cost, commodity hardware," in *Proceedings of ACM International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, p. 69.
21. K. Tian, Y. Dong, and D. Cowperthwaite, "A full gpu virtualization solution with mediated pass-through," in *Proceedings of USENIX ATC*, 2014.
22. E. Seppanen, M. T. O'Keefe, and D. J. Lilja, "High performance solid state storage under linux," in *Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–12.
23. Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, H. Eom, and H. Y. Yeom, "Exploiting peak device throughput from random access workload," in *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, 2012, p. 7.
24. C. Xu, S. Gamage, H. Lu, R. R. Kompella, and D. Xu, "vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core." in *Proceedings of USENIX Annual Technical Conference*, 2013, pp. 243–254.
25. J. Liu and B. Abali, "Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization," in *Proceedings of the 23rd ACM International Conference on Supercomputing*, 2009, pp. 225–234.

Myoungwon Oh is a Ph.D. student and received MS degree in Computer Science and Engineering at Seoul National University, Seoul, Korea. He received BS degree in Computer Software from Gwangwoon University, Seoul, Korea, in 2010. His main research interests are distributed file system, operating system and high performance storage system.

Hanul Sung is a Ph.D. candidate in Computer Science and Engineering at Seoul National University, Seoul, Korea. She received BS degree in Computer Science from Sangmyung University, Seoul, Korea, in 2012. Her main research interests are distributed systems, operating systems, high performance storage systems and cloud computing.

Sanggeol Lee received the BS degree in Electronics from Sungkyunkwan University, Suwon, Korea, in 2006. He is currently a senior engineer in the Department of Semiconductor at Samsung Electronics, where he has been working for since 2005. He is an application engineer for enterprise SSDs. His research interests include enterprise SSDs, all flash arrays, distributed file systems, virtualizations, operating systems, cloud computing and datacenters.

Hyeonsang Eom received the BS degree in Computer Science and statistics from Seoul National University, Seoul, Korea, in 1992, and the MS and Ph.D. degrees in Computer Science from the University of Maryland at College Park, Maryland, USA, in 1996 and 2003, respectively. He is currently an Associate Professor in the Department of Computer Science and Engineering at SNU, where he has been a faculty member since 2005. He was an intern in the data engineering group at Sun Microsystems, California, USA, in 1997, and a senior engineer in the Telecommunication R&D Center at Samsung Electronics, Korea, from 2003 to 2004. His research interests include distributed systems, cloud computing, operating systems, high performance storage systems, energy efficient systems, fault-tolerant systems, security, and information dynamics.



Heon Young Yeom is a Professor with the School of Computer Science and Engineering, Seoul National University. He received B.S. degree in Computer Science from Seoul National University in 1984 and his M.S. and Ph.D. degrees in Computer Science from Texas A&M University in 1986 and 1992 respectively. From 1986 to 1990, he worked with Texas Transportation Institute as a Systems Analyst, and from 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University in 1993, where he currently teaches and researches on distributed systems and transaction processing.