Improving Performance by Bridging the Semantic Gap between Multi-queue SSD and I/O Virtualization Framework

Tae Yong Kim*†, Dong Hyun Kang*, Dongwoo Lee*, Young Ik Eom*
*College of Information and Communication Engineering, Sungkyunkwan University, Suwon, South Korea
†SSD Development Team, Samsung Electronics, Hwasung, South Korea
{taeyongkim, kkangsu, dwlee, yieom}@skku.edu

***Abstract*—Virtualization has become one of the most helpful techniques, and today it is prevalent in several computing environments including desktops, data-centers, and enterprises. However, an I/O scalability issue in virtualized environments still needs to be addressed because I/O layers are implemented to be oblivious to the I/O behaviors on virtual machines (VM). In particular, when a multi-queue solid state drive (SSD) is used as a secondary storage, each VM reveals *semantic gap* that degrades the overall performance of the VM by up to 74%. This is due to two key problems. First, the multi-queue SSD accelerates the possibility of *lock contentions*. Second, even though both the host machine and the multi-queue SSD provide multiple I/O queues for I/O parallelism, existing Virtio-Blk-Data-Plane supports only one I/O queue by an I/O thread for submitting all I/O requests. In this paper, we propose a novel approach, including the design of virtual CPU (vCPU)-dedicated queues and I/O threads, which efficiently distributes the lock contentions and addresses the *parallelism issue* of Virtio-Blk-Data-Plane in virtualized environments. We design our approach based on the above principle, which allocates a dedicated queue and an I/O thread for each vCPU to reduce the semantic gap. We also implement our approach based on Linux 3.17, and modify both the Virtio-Blk frontend driver of guest OS and the Virtio-Blk backend driver of Quick Emulator (QEMU) 2.1.2. Our experimental results with various I/O traces clearly show that our design improves the I/O operations per second (IOPS) in virtualized environments by up to 167% over existing QEMU.**

Keywords—multi-queue; solid state drive; non-volatile memory express; lock contention; parallelism; virtualization; quick emulator

I. INTRODUCTION

Today, virtualization is one of the most helpful techniques, which has been stabilized through several optimization techniques. Therefore, it is now prevalent in several computing environments including desktops, data-centers, and enterprises. However, in the area of virtualization, I/O performance issues still need to be addressed because I/O layers are implemented to be oblivious to the I/O behaviors. Thus, both industry and academia have focused on optimizing the I/O layers in virtualization. Their approaches can be classified into two categories: hardware approach and software approach. Today, a number of chip manufacturers provide a variety of technologies and interfaces, such as Virtualization Technology (VT) [1], Virtualization Extensions (VEs) [2], Single Root I/O Virtualization (SR-IOV) [3], and I/O Memory Management Unit (IOMMU) [4], for supporting virtualization at the hardware level. However, these approaches can be inappropriate in some virtualized environments since they require specialized hardware, even though their performance has practically reached that of bare-metal systems. In addition, SR-IOV and IOMMU limit the benefits of virtualization (e.g., VM migration [5], [6]) by dedicating a virtual machine to a physical device of the host machine. Therefore, many efforts have been made to overcome the limitations of the hardware approaches. Some researchers have reported the problems of I/O overheads incurred during communication between the guest and the host machine (e.g., *exit* and duplicated I/O layers) and have proposed software-based approaches such as Virtio [7], Kernel-based Virtual Machine (KVM) [8], and Efficient and Scalable Para-virtual I/O System (ELVIS) [9] in order to minimize the I/O overheads.

A multi-queue SSD with Peripheral Component Interconnect Express (PCIe) [10] dramatically accelerates the I/O performance of a secondary storage by exploiting parallelism in SSD (e.g., Non-Volatile Memory Express (NVMe) SSD). Unfortunately, it does not directly result in overall system performance. This is because the block layer of the operating system waits to hold a lock whenever an I/O request is sent from a single request queue to the multi-queue SSD. To address this lock contention problem, in previous work, a block layer was proposed that efficiently improves the system performance by using two levels of queues: software staging queues and hardware dispatch queues [11]. However, in virtualized environments based on QEMU [12], the proposed mechanism cannot take advantage of the multi-queue SSD because QEMU has another lock contention problem, which is incurred by using the Virtio-Blk-Data-Plane technique [17]. This technique was introduced to optimize the performance of VMs. It provides a dedicated I/O thread for each I/O device in the host, and skips some of the duplicated I/O layers between the host and the guest (e.g., the I/O scheduler and the block layer). The dedicated I/O thread directly issues I/O requests of its vCPU to a shared circular queue in QEMU after holding the lock on the single queue. In addition, the dedicated I/O thread, which is responsible for submitting I/O requests in the single queue to the host kernel, periodically attempts to hold the lock on the single queue. As a result, QEMU significantly suffers from frequent lock

978-1-4673-7619-8/15/$31.00 ©2015 IEEE

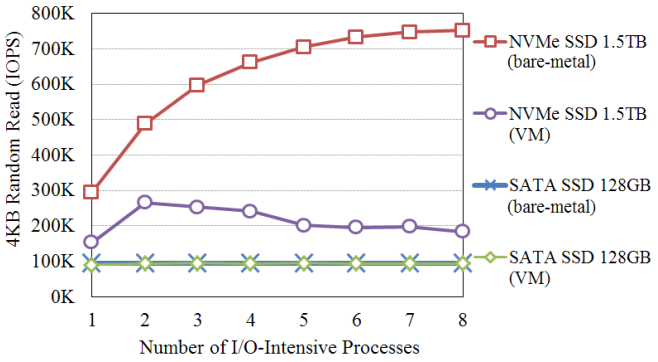

Fig. 1. Performance comparison in a virtualized and a non-virtualized environment with a SATA SSD and an NVMe SSD through FIO.

contentions. To relieve the lock contentions in virtualized environments, Ming Lei proposed the design of Virtio-Blk multiple queues with two optimization schemes [13]. However, the proposed approach cannot fully take advantage of the multi-queue SSD since the I/O scalability problem remains due to using the single I/O thread. Oh et al. also proposed a scheme involving a pipelined polling I/O thread to optimize the performance of the I/O thread. However, they did not consider the I/O scalability of a VM [14].

To understand the impact on the performance of the VM when using the multi-queue SSD and the SSD based on Serial ATA (SATA) interface, we first measured random read performance by varying the number of I/O-intensive processes with a Flexible I/O tester (FIO) benchmark [15], and compared the performance to that of bare-metal systems (Figure 1). Our experimental results show two important implications. First, the VM with the multi-queue SSD reveals poor performance, with a decrease of up to 74% as the number of I/O processes increases. Second, the results of the SATA-based SSD are similar to those obtained on bare-metal systems. These results imply that the performance of the VM decreases as the I/O latency increases because the high I/O latency accelerates the possibility of the lock contentions that degrade the overall performance of the VM. In addition, the existing Virtio-Blk-Data-Plane in QEMU utilizes only one I/O request queue by an I/O thread for submitting all I/O requests, even though both the host machine and the multi-queue SSD provide multiple I/O request queues for parallelism. As a result, the *lock contention* problem and the *parallelism issue* incur the *semantic gap* between the host and the guest.

In this paper, we propose a novel approach with the design of vCPU-dedicated queues and I/O threads, to address the semantic gap. vCPU-dedicated request queues efficiently distribute the lock contentions by allocating a dedicated queue for each vCPU. vCPU-dedicated I/O threads improve the parallelism by exploiting the characteristics of the multi-queue SSD (e.g., multiple queue-pairs and multiple interrupts).

Our main contributions can be summarized as follows:

- **Motivation and Design**. We analyze the performance of the virtualized systems and find significant performance degradations in the virtualized system with multi-queue SSD. This observation motivates the design of our approach. Based on this observation, we design a novel approach that distributes the lock contentions and improves the parallelism by extending the Virtio-Blk-Data-Plane with vCPU-dedicated queues and I/O threads. We also implement the prototype of the proposed approach by modifying both the frontend driver of the guest OS and the backend driver of QEMU 2.1.2.
- **Three optimizations**. In order to further optimize the I/O performance, we introduce three optimization techniques as follows. (1) We set the CPU affinity for callback functions of the request queues to prevent unnecessary CPU scheduling. (2) We eliminate the inter-process interrupt (IPI) mechanism of the guest to simplify I/O path. (3) We dynamically adjust the number of I/O requests for I/O batch submission according to incoming workload volumes.
- **Evaluations on both a null block device and a real SSD**. We evaluate our approach on a null block device [19], which simulates a multi-queue virtual device by receiving I/O requests and acknowledging I/O completions immediately. Also, we evaluate a real NVMe SSD to increase the scope of our evaluation. The experimental results clearly show that the proposed approach improves the I/O performance in a virtualized environment by up to 167% over the state-of-the-art virtualization approach, known as Virtio-Blk-Data-Plane.

The remainder of this paper is organized as follows. We explain the multi-queue SSD and describe the architecture of QEMU and its behaviors in Section II. We present the design of our approach and three optimization techniques in Section III. In Section IV, we analyze the results of our evaluation. Finally, we discuss related works in Section V and conclude our findings in Section VI.

II. Background

A. Multi-queue SSD

While SSDs have been improved due to internal parallelism of Non-Volatile Memory (NVM) and fast response time, their potential could not be further exploited due to the physical limit on hardware. For example, SATA (which is a typical storage interface in desktop environments) has 6 Gbps in link speed, and the link speed of SAS, which is a standard in enterprises, is up to 12 Gbps. Unfortunately, recently-released SSDs have already reached the full speed provided by hardware interfaces. In short, the performance bottleneck has shifted from SSDs to host interfaces.

A multi-queue SSD is a PCIe-based SSD with novel protocols such as NVMe and Small Computer System Interface (SCSI) express, which contribute to higher performance when compared to a SATA SSD and a Serial Attached SCSI (SAS) SSD. Thus, this has become a general trend, replacing conventional SSDs. Meanwhile, PCIe was previously utilized as a graphic device interface due to its powerful link speed, reaching 128 Gbps. However, it has recently been used as a storage interface because of its low latency.

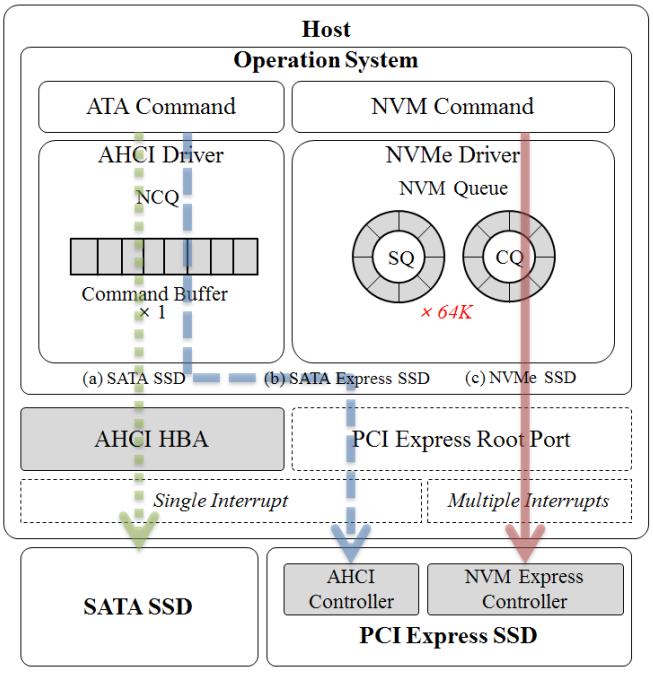

Fig. 2. Comparison of I/O paths and main components between SATA SSD and PCIe-based SSD.

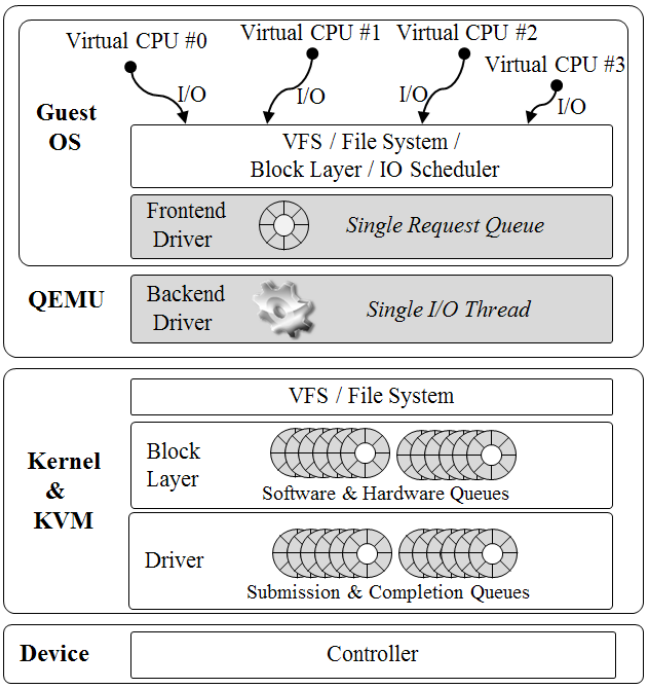

Fig. 3. Primary software layers and data structures for I/O requests in a single VM with four vCPUs.

PCIe has many advantages as a storage interface. First, PCIe 3.0 supports 8 Gbps link speed per lane and 128 Gbps with 16 lanes. This is ten times faster than typical storage interfaces. Second, PCIe reduces hardware overheads. A PCIe-based SSD can be directly connected to the PCIe interface, while conventional SSDs are attached to the host through an additional host chipset or host bus adaptor. This difference creates a bus overhead of 1 microsecond or more per command. The overhead is not a major issue for HDDs which transfer 4KB of data in 10 microseconds, but SSDs can transfer the same size data in 2 microseconds or less. Thus, reducing hardware overheads is essential for developing high performance SSDs. For these reasons, PCIe interface qualifies as a storage interface [16].

The PCIe-based SSD needs a software interface, namely protocol, as well as a hardware interface to communicate with a computer system; NVMe is a typical example, and Advanced Host Controller Interface (AHCI) and SCSI express can also be used. However, AHCI is not suitable for multi-core systems with high performance storage as it is an old standard for HDDs, and the specification of SCSI express is still ongoing, and no product has yet been released. NVMe has been originally designed to address the needs of desktop, data-center, and enterprise systems by maximizing parallelism and supporting future NVM technologies. To achieve its goals, NVMe has two primary characteristics: it has up to *64K I/O queues* with 64K commands per queue and is an aggregation of *2K MSI-X interrupts* [10]. These features mean that NVMe can transfer data concurrently on multi-core systems without synchronizations among the CPU cores. Figure 2 shows multiple layers with three data paths of two software protocols for the multi-queue SSD. Unlike AHCI, NVMe has separate submission and completion queues for handling I/O requests. Furthermore, NVMe applies 2,048 interrupts with a steering scheme while AHCI supports only a single interrupt.

Generally, the development of storage technologies requires structural changes in software and computer systems. Multi-queue SSDs would be used broadly from desktops, data-centers, and enterprises due to their advantages [16]. Several manufacturers have been releasing PCIe-based NVMe SSDs, and their performance reaches up to 750K IOPS of 4KB random read. Moreover, their throughput achieves up to 3 GB/s in 32KB sequential read. These I/O performance improvements challenge the processing ability of software such as the OS and applications. In particular, it is probable that PCIe-based NVMe SSDs largely affect virtualized systems because of complicated architectures on software.

B. QEMU and Virtio-Blk-Data-Plane

In our research, we use QEMU to create a VM with Virtio-Blk-Data-Plane (which is an outstanding I/O virtualization technique) and propose a new architecture based on Virtio-Blk-Data-Plane through specific structural analyses. QEMU is a typical open source machine emulator and hypervisor. Especially, QEMU achieves near native performance by executing the guest code directly on the host CPU using KVM or Xen in Linux on x86 machines [7], [8].

Virtio-Blk-Data-Plane rapidly accelerates I/O operations through a para-virtualized I/O technique called Virtio-Blk. The main feature of Virtio-Blk-Data-Plane is a dedicated per-device thread for processing I/O requests. When several storage devices are attached on the same VM, this approach can process I/O requests of each device in parallel by avoiding synchronizations that requires acquiring a global mutex. As a

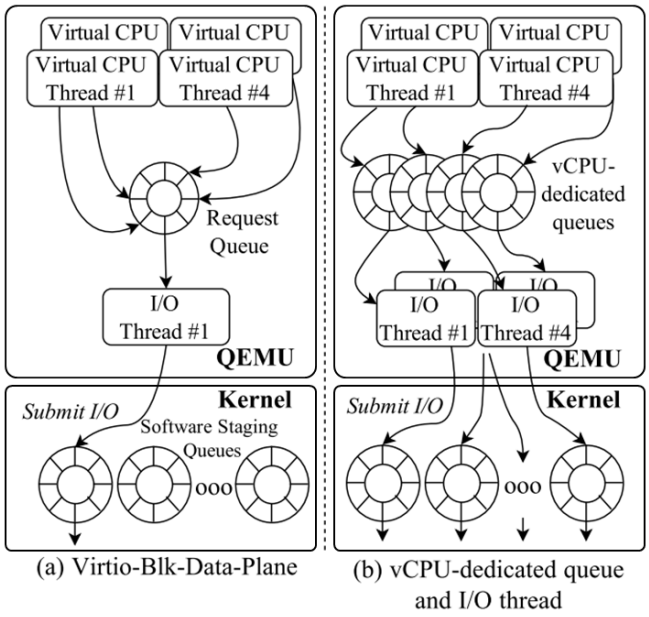

Fig. 4. Comparison of the architecture between (a) virtio-blk-data-plane and (b) vCPU-dedicated queue and I/O thread.

result, the Virtio-Blk-Data-Plane technique achieved up to 200K IOPS per SCSI target server, or a total of 1.5M IOPS with 7 SCSI target servers at 4KB I/O requests in contrast to the Virtio-Blk reaching a total of 147K IOPS [17].

Actually, the process of I/O request transfer from a vCPU in a VM to the host device is quite complicated. Figure 3 illustrates a typical architecture when a single VM is created by QEMU with four vCPUs, and shows an I/O path from the vCPUs to the device. As shown in Figure 3, the number of software layers and data structures through which an I/O request should pass is excessive as compared to non-virtualized systems. The software layers prevent a VM from improving I/O performance. Therefore, we focus on the frontend driver in the guest OS and the backend driver in QEMU, and attempt to enhance I/O virtualization mechanism in our research.

III. Architecture and Implementation

In this section, we propose the design of *vCPU-dedicated queues and I/O threads* that addresses I/O scalability problem. Various optimizations are possible to improve performance. However, the significant *semantic gap* between multi-queue SSD and I/O virtualization framework should be realized and resolved above all. This not only improves performance, but also motivates other optimizations.

A. Design of vCPU-dedicated queues and I/O threads

vCPU-dedicated queue: First, single request queue in QEMU should be modified to solve the *lock contention* problem when QEMU uses the Virtio-Blk-Data-Plane technique. According to our analysis, one vCPU basically acquires a single global mutex when accessing the shared request queue atomically. In this situation, the other vCPUs which try to access the queue should wait the mutex continuously. This operation poses severe lock contentions among the vCPUs, which result in unnecessary CPU scheduling.

Therefore, we provide QEMU with a *vCPU-dedicated request queue* per vCPU. The considerable advantage of our approach is the advanced vCPU parallelism achieved by minimizing the lock contentions. Figure 4 shows our abstract architecture, which mainly describes major threads and data structures, before (a) and after (b) applying the vCPU-dedicated queues when a VM that consists of 4 vCPUs is created by QEMU. The number of request queues located between vCPUs and I/O threads is significantly different between the two architectures. In previous studies [13], [14], a small number of request queues were utilized regardless of the number of vCPUs in a VM. When there are only a few request queues and the VM has more vCPUs than request queues in order to enhance overall performance on multi-core system, the performance is again limited due to lock contentions. Thus, to sustainably diminish the lock contentions, the number of request queues should be identical to the number of vCPUs. As a result, if the number of I/O-intensive processes is equal to the number of vCPUs, the waiting time to acquire the lock decreases by 80%, from 50 microseconds to 10 microseconds. Also, in addition to the vCPU-dedicated queue, the mechanism of the vCPU-dedicated I/O thread is essential to enhance I/O parallelism.

vCPU-dedicated I/O thread: The main issues on the I/O parallelism is closely related to the single I/O thread, even though the vCPU-dedicated queue is able to partially contribute on the performance. The Linux kernel generally has per-core software queues and multiple hardware queues for I/O parallelism. Moreover, the position of the software queue, where an I/O request will be inserted, is determined by the index of the host CPU that submits the I/O requests. However, even if each vCPU requests I/Os, the host OS cannot recognize which vCPU submits the I/O request because the single I/O thread actually submits all I/O requests. For this reason, when a VM has a single I/O thread, all I/O requests are inserted into one queue in the host block layer. Consequently, the single I/O thread inefficiently uses the host software queues. In such a case, the performance on a multi-queue SSD is limited due to the utilization rate of the software queues. This is because the multi-queue SSD is able to maximize its performance when the software queues are fully utilized. In addition, the single I/O thread inevitably becomes a serious bottleneck due to the significant number of I/O completions that the single I/O thread should handle.

We propose a mechanism of *vCPU-dedicated I/O threads*, the number of which is the same as the number of vCPUs, and each I/O thread shares the dedicated request queue with a vCPU. This mechanism of vCPU-dedicated I/O thread can improve parallelization by fully exploiting multiple hardware queues in a multi-queue SSD because each I/O thread is commonly executed by a non-overlapping CPU core. The differences between the single-threaded I/O design (a) and the vCPU-dedicated I/O thread design (b) are illustrated in Figure 4. While some researchers asserted that small number of I/O threads can utilize the entire hardware performance [13], [14], this may be incorrect. Of course, four I/O-intensive processes can utilize the overall performance of an NVMe SSD,

achieving up to 750K IOPS, but this is shortsighted because the performance of SSD can be increased constantly. Eventually, to solve the fundamental performance limitation due to the single I/O thread, the mechanism of vCPU-dedicated I/O thread should be applied to QEMU, and we verify our design in Section IV.

Moreover, the mechanism of vCPU-dedicated I/O thread offers another significant benefit since our approach can consistently maintain I/O patterns of applications in the guest to a storage device in the host. Mostly, the performance of sequential access to SSDs is faster than that of random access. Therefore, maintaining the characteristic of the sequential access is essential to improve the performance. However, sequential access from the guest is polluted by the single I/O thread in virtualized environments, because every I/O workload is mixed in the single I/O thread. The characteristic of the combined I/O workload is very similar to that of random access from the viewpoint of the host. On the contrary, the vCPU-dedicated I/O thread mechanism can avoid pollution of the I/O workloads due to the separation of vCPU-dedicated queues and I/O threads. Unfortunately, this advantage may not result in performance improvements immediately without the technical support of SSDs because the SSDs should manage the I/O workloads internally to get the advantage. Nevertheless, this merit is essential for upcoming higher performance SSDs. In our evaluation, we verify that the performance is gradually improved as the number of I/O threads increase by up to the number of vCPUs. Moreover, we demonstrate the specific result of the experiment and the test environment in Section IV.

Although the design of vCPU-dedicated queues and I/O threads contributes to higher performance improvements, the VM still has the issue of unrevealed overheads. Therefore, to optimize parallelism and to get better performance, (1) we configure CPU affinity for callbacks of I/O completions, (2) remove useless inter-processor interrupts, and (3) enhance the technique of the I/O batch submission through awareness of the I/O workloads.

B. *Configuring CPU Affinity for I/O Completions*

The design of the vCPU-dedicated queue and the I/O thread motivates QEMU to enhance the I/O completion process in addition to the performance improvements by their mechanism. QEMU usually uses Message Signaled Interrupts Extended (MSI-X) interrupts of PCIe for I/O completions when I/O requests are performed by the host. Unlike the line-based interrupt, MSI-X is able to designate the number of CPUs for interrupt handling to deal with the interrupts effectively on multi-core systems. For example, if the 4th CPU is designated for interrupt handling and the interrupt is triggered, all I/O requests would be completed on the 4th CPU. This feature reduces scheduling overheads and enhances cache hit rates by processing the I/O submission and the completion on the same CPU [16], [18]. In practice, NVMe has adopted this technique to simplify its I/O path and complicated interrupt handlings. Even though QEMU has also adopted MSI-X, this is actually useless due to the single I/O thread which is responsible for submissions and completions. This is largely because a single designated CPU for the I/O thread can become a bottleneck. To be specific, a considerable number of I/O completions converge simultaneously on the same CPU if a single CPU is assigned for all I/O completions. In addition, in the case of multiple designated CPUs for the I/O thread, the number of cache miss rates will significantly increase because submissions and completions can be handled on different CPUs.

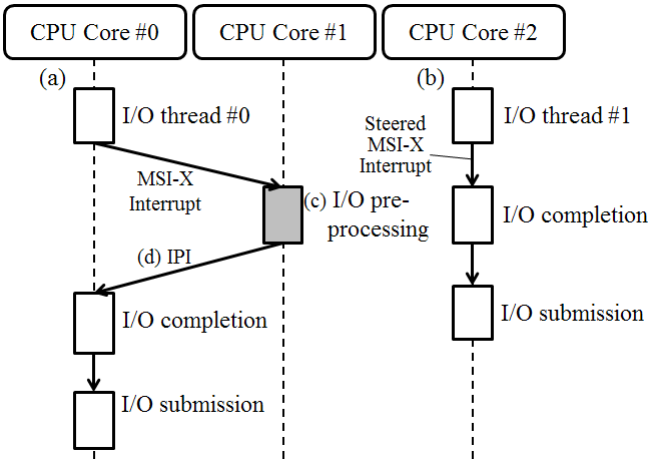

Fig. 5. Comparison between non-optimized and optimized I/O completion paths.

In this respect, the design of the vCPU-dedicated queue and I/O thread necessitates an appropriate configuration on the CPU affinity. Thus, we assign a single non-overlapping CPU per vCPU-dedicated I/O thread. This improves the cache hit rates and reduces the scheduling overheads caused by unnecessary context switches [18]. As a result, the performance was improved by 10% via this configuration on CPU affinity. We demonstrate the specific result of the experiment in Section IV.

C. *Eliminating Inter-process Interrupts*

Configuring CPU affinity for I/O completions not only improves performance but also provides another optimization point which simplifies an I/O path by removing useless IPIs. When a hardware interrupt that notifies an I/O completion is triggered, an IPI generally occurs to steer the interrupt to the particular CPU which issued the I/O request. Similarly, QEMU also utilizes IPIs for steered I/O completions. An IPI, of course, increases the cache hit rates; however, this induces scheduling delays due to the additional interrupt handlings. However, through the configuration on CPU affinity, useless IPIs can be entirely eliminated. In our experiment, we measured in detail the latencies among software layers such as a guest kernel, a guest frontend driver, a QEMU backend driver, and a host kernel. The result shows that the latency of about 150 microseconds between the frontend driver and the guest kernel completely disappeared when configuring the CPU affinity.

Figure 5 illustrates a conventional I/O path (a) and an improved I/O path by configuring CPU affinity and eliminating IPIs (b). While MSI-X interrupts are mostly handled by different CPU in the case of I/O thread No. 1, an I/O completion of I/O thread No. 2 is directly processed on the

same CPU without scheduling delays (c). Moreover, the I/O completion path has been shortened by removing IPIs (d).

D. Workload-aware I/O batch submission

To improve performance, we investigated an I/O batch submission technique, and finally discovered a significant source where optimization is required. The I/O batch submission is an effective technique used to improve performance by batching all I/O requests in the request queue and submitting them to the host at the same time through io_submit system call. This technique can diminish the frequency of mode switches between user mode and kernel mode. In a previous study [13], the effectiveness of the I/O batch submission was presented, showing that the number of I/O requests per system call is increased by 26 times and the performance was improved by up to 54%. However, this technique may degrade the performance by the feature of the vCPU-dedicated I/O thread because I/O requests are distributed by allocated I/O threads even during I/O-intensive workloads unlike the single-threaded architecture.

We implemented an advanced I/O batch submission technique that accumulates I/O requests efficiently by recognizing whether the I/O workload is heavy and issuing the batched requests all at once when necessary. This is largely because of the relative efficiency whereby the I/O requests are handled at the same time via only one submission system call in the case of I/O-intensive workloads. Therefore, our approach, *the workload-aware I/O batch submission*, records the number of I/O requests per system call and estimates intensiveness of the I/O workload through the batched I/O history. Moreover, during executing the I/O-intensive workloads, our technique additionally waits for more time to batch more I/O requests before submitting. To verify this technique, we present the comparative experiment in Section IV.

IV. Experiments and Evaluation

A. Experimental Group

In our evaluation, we compared our architecture with two other solutions as follows.

Baseline: This is an unmodified QEMU 2.1.2 with the Virtio-Blk-Data-Plane technique, which is the latest version now. Baseline consists of a single request queue and a single I/O thread, which leads to the I/O scalability problem.

MQ: This was presented in a previous research conducted by Ming Lei [13]. MQ supports a feature of multiple request queues unlike Baseline, and it has a single I/O thread like Baseline. In our experiment, we directly compiled the MQ source code without any modifications, and allocated 8 request queues which were equal to the number of created vCPUs.

MIOT: This is our approach based on QEMU 2.1.2 (which is identical to Baseline) that applies the design of vCPU-dedicated queues and I/O threads. Moreover, all the proposed optimizations are adopted, including configuring CPU affinity for I/O completions, eliminating IPIs, and workload-aware I/O batch completion.

B. Experiment Setup

All experiments were performed on a system equipped with an Intel i7-2600 hyper-threaded quad-core CPU running at 3.40GHz, 16GB RAM, and two types of storage devices. The first storage type is a null block device which is a virtual device for simulation of storage. The second storage type is a Samsung XS1715 1.5TB NVMe SSD that can deliver 750K IOPS on random reads, and 350K IPS on random writes. In addition, it also offers a speed of 2780MB/s in sequential reads and a speed of 1330 MB/s in sequential writes.

The null block device [19], used widely in our evaluation, is highly suitable for simulating a multi-queue SSD. It is mostly similar with the Linux null device regarding internal I/O operations, but it particularly has a characteristic of multiple queues. Therefore, much like the NVMe SSD, the null block device is able to simulate multi-queue operations without a real device. Furthermore, it can validate a higher range of performance where the NVMe SSD is unable to reach. For example, the null block device is capable of reaching 1200K IOPS with 8 I/O-intensive processes.

The experimental system creates one VM which consists of 8 vCPUs and 14GB RAM. Although the system is equipped with a quad-core CPU, it can achieve its maximum performance using 8 threads due to hyperthreading. This is the reason why we allocated 8 vCPUs in one VM. In addition, the VM directly attaches storage devices to minimize the duplicated software layers between the host and the guest.

In general, normal experiments on scalability were previously performed on datacenter or enterprise systems equipped with many-core CPUs, unlike our experimental setup. However, we evaluate all experiments on a desktop PC because NVMe SSDs have been developed not only for datacenter and enterprise systems but also for desktop environments, and the maximum performance of the NVMe SSD is easily achieved with a quad-core CPU for desktops. Thus, we can fully verify the I/O scalability in our experimental environment. For these reasons, high performance systems are not practically vital for our research.

C. FIO benchmark for I/O workloads

All I/O workloads are generated by FIO microbenchmark to clarify the experimental results through detailed configurations such as request type, address, data size, queue depth, I/O engine, cache mode, and even the number of I/O processes. Furthermore, the benchmark demonstrates internal information such as latency and the number of context switches as well as IOPS and throughput (MB/s). These outputs help to clarify our experimental results.

Basically, we use four types of I/O workloads: 4KB random read, 4KB random write, 32KB sequential read, and 32KB sequential write. We also set the I/O engine to libaio, the I/O depth to 32, and the non-cache mode. To verify I/O scalability, we vary the number of I/O processes from 1 to 8, and each I/O process transfers a total of 1GB data.

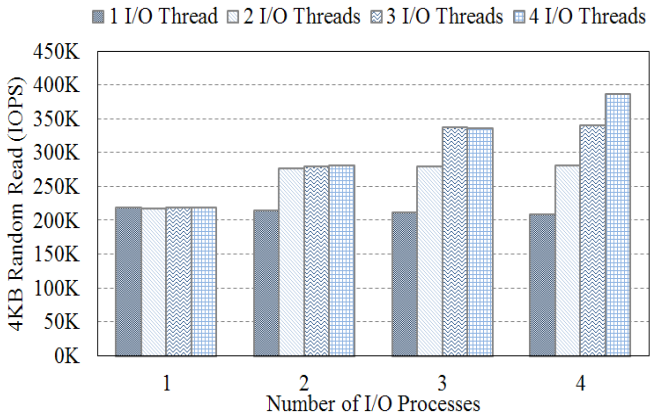

Fig. 6. IOPS with FIO when scaling the number of I/O threads.

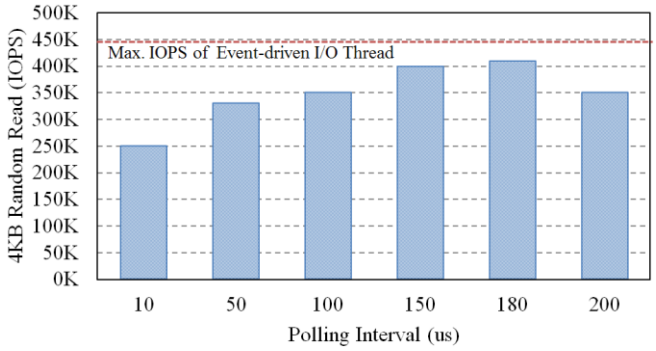

Fig. 7. Performance with varying polling interval on polling-based vCPU-dedicated I/O thread design.

D. *Impact of the Number of I/O Threads*

In order to verify the effectiveness of the number of I/O threads, we explore the performance of 4KB random read varying the number of I/O threads because the small size random read is beneficial to make the I/O threads read data in parallel. To improve experimental accuracy, we specify a non-overlapping CPU affinity for each I/O thread, which helps to prevent unintended CPU scheduling. This is because an I/O thread can be processed by a number of CPUs through the host CPU scheduling in a short period of time. This results in I/O requests being distributed to the different software queues in the host, as in the multiple I/O thread techniques. Figure 6 shows that the number of I/O threads has a distinct effect on performance. In conclusion, more I/O threads obviously contributed to higher performance and the design of the vCPU-dedicated I/O thread was motivated by this result.

E. *Event-driven I/O Thread vs. Polling I/O Thread*

The existing QEMU basically uses an event-driven I/O thread that reacts to events such as I/O submission and completion by dispatching to event handlers. However, we attempted a new I/O thread based on the polling mechanism to achieve better performance. Numerous previous studies [20], [21], [22] claimed that overheads in I/O virtualization were mainly caused by *exits* between the guest and the host. Thus, most studies proposed *exitless* methods based on the polling mechanism. Likewise, the existing QEMU generates *exits* when a vCPU notifies the I/O thread of the I/O submissions and a device notifies the I/O completions to the I/O thread, because the event-driven I/O thread is also an *exit-based* architecture. Furthermore, it is likely that many of the *exits* pose performance degradations due to unnecessary context switches.

In our approach, we observed that the event-driven design of vCPU-dedicated queues and I/O threads generates more *exits* while improving overall parallelism. Thus, we implemented and tried to use a polling-based vCPU-dedicated I/O thread to diminish the number of increased *exits*. As shown in Figure 7, we measured performance with various polling intervals on 4KB random read, and the polling interval was varied from 10 microseconds to 200 microseconds in order to find the appropriate interval. Of course, we validated the effect of the polling I/O thread using perf [23]; using the polling I/O thread, the number of *exits* was decreased by 76%. However, the performance of optimized polling I/O threads was not impressive, because it was not superior to that of event-driven I/O threads. This is largely because (1) one vCPU can no longer have a negative effect on the other vCPUs because of the minimized lock contentions through the feature of the vCPU-dedicated queue. (2) QEMU does not always produce an *exit* for every I/O request and response. Through the I/O batch techniques provided by Linux block I/O [24], the performance degradation caused by *exits* is not as severe as expected, and (3) all polling-based techniques commonly have an inherent problem in terms of CPU utilization rates regardless of the optimized polling interval. For these reasons, we finally adopted the event-driven I/O thread which is highly appropriate for our design.

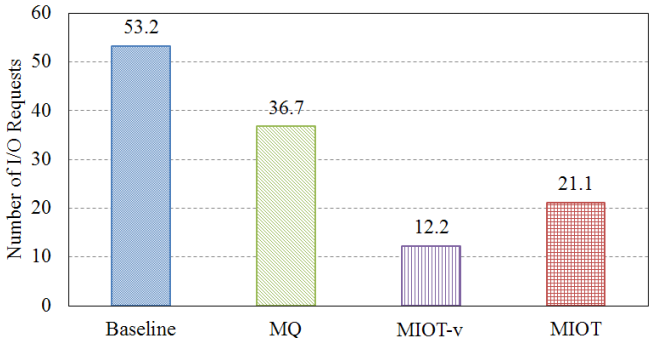

Fig. 8. Measuring the number of I/O requests in a single I/O batch submission for Baseline, MQ, MIOT, and MIOT without the workload-aware I/O batch submission technique.

F. *Effect of Workload-aware I/O batch submission*

To verify the effect of the workload-aware I/O batch submission, we measured the number of I/O requests handled at the instant of each I/O batch submission, and compared the MIOT with prior works such as Baseline and MQ. In addition, MIOT excluding workload-aware I/O batch submission (denoted as MIOT-v) was also subject to this experiment. As shown in Figure 8, the measured value of MIOT is 21.1 on

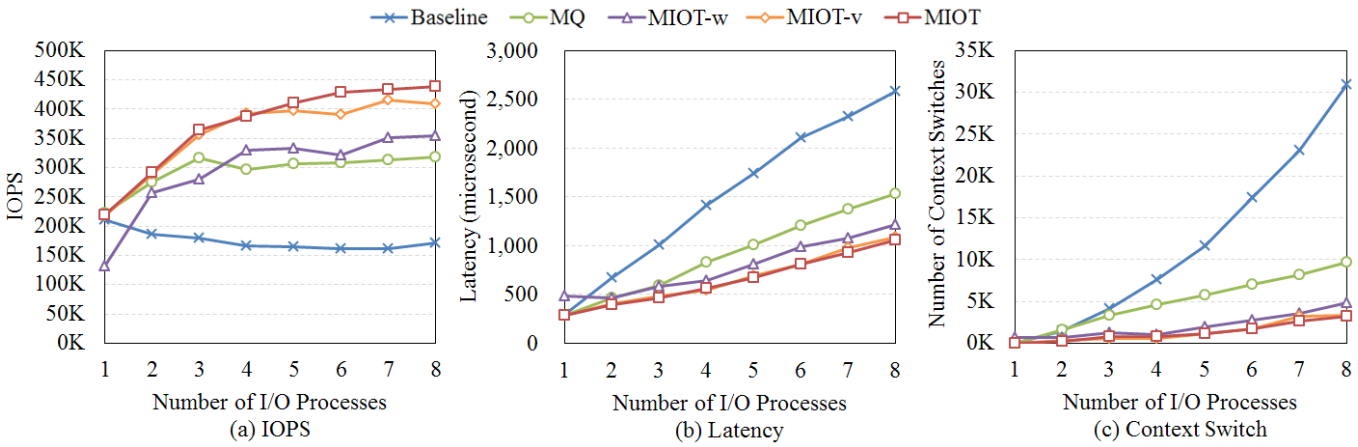

Fig. 9. Comparison of IOPS, latency, and the number of context switches among Baseline, MQ, MIOT-w, MIOT-v, and MIOT.

average, while that of Baseline is 53.2. The major reason for this result is that I/O requests were distributed by numerous I/O threads. However, in terms of the number of I/O requests per unit time, the result of MIOT is possibly higher than that of Baseline, because unlike the other threads vCPU-dedicated I/O threads performs their operations simultaneously. Nevertheless, the measured value of MIOT is increased by up to 72% compared to MIOT-v. As a result, the overall performance was improved by 10% through workload-aware I/O batch submission.

G. *Analysis of the Effect of Three Optimizations*

To analyze the effectiveness of the three optimizations, we measured the IOPS, latency, and the number of context switches through 4KB random read of FIO, and the measurement was performed with five targets: Baseline, MQ, MIOT without the three optimizations (denoted as MIOT-w), MIOT without only the workload-aware I/O batch submission (denoted as MIOT-v), and MIOT.

First, the result of IOPS demonstrates that the three optimizations have a positive impact on performance as shown in Figure 9(a). Especially, MIOT improves IOPS by up to 167% compared with Baseline because it reduces the overhead of context switches by using multiple I/O threads. Figure 9(b) shows that all latencies are proportional to the number of I/O processes, but there is no particular difference among MIOT-w, MIOT-v, and MIOT. As we expected, MIOT significantly reduces the latency by up to 59% over Baseline.

Note that, as shown in Figure 9(c), the number of context switches of Baseline increases fairly steeply compared to the other solutions as the number of I/O processes increases. On the other hand, the measured the values of MIOT-w, MIOT-v, and MIOT are approximately 10% of that of Baseline. This demonstrates that excessive context switches, caused by the single request queue and the single I/O thread, are the major cause of performance degradations.

Most previous works in virtualization environments focused on the *exit* overhead because it significantly degrades performance of virtual machine. Therefore, we measured the number of *exits* and the total time that is used for handling the

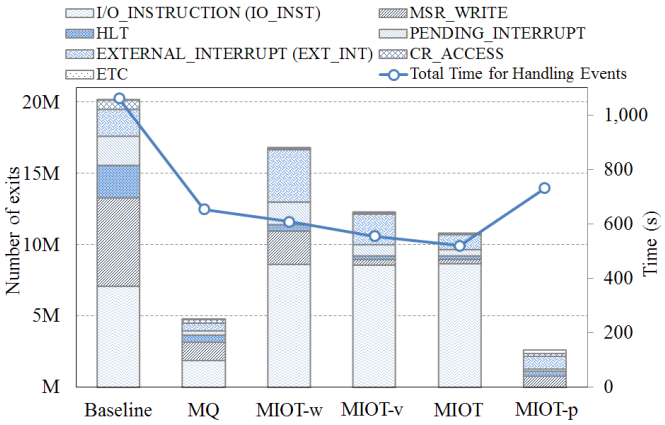

Fig. 10. Measurement of the number of exits and the cause of the exit to verify the effectiveness of the various optimizations.

events during the same amount of time for executing the I/O workload, and analyzed the relation between the outcomes and performances. Figure 10 shows the number of *exits* while running the perf [23] that is a standard tool to profile performance counters. As shown in Figure 10, the experiment was performed with six targets: Baseline, MQ, MIOT without the three optimizations (denoted as MIOT-w), MIOT without the workload-aware I/O batch submission (denoted as MIOT-v), MIOT, and MIOT with the polling technique (denoted as MIOT-p). First, MIOT-w decreased the number of *exits* by 17% compared to Baseline, mainly due to the drop of MSR_WRITEs and HLTs. In contrast, IO_INSTs and EXT_INTs increased because of the improved performance due to I/O parallelism. Moreover, the number of *exits* decreased with MIOT-v and MIOT because of the reduction in EXT_INTs. Particularly, due to MIOT-v, the MSR_WRITEs were dramatically reduced by 95%. In addition, the workload-aware I/O batch submission decreased the EXT_INTs by 49%. In these results, the number of *exits* is mostly proportional to the total time for handling events (Figure 10); however it is invalid in the case of MQ and MIOT-p. Although the number of *exits* in MQ and MIOT-p is much lower than that of the other solutions, not only the total time for handling events but

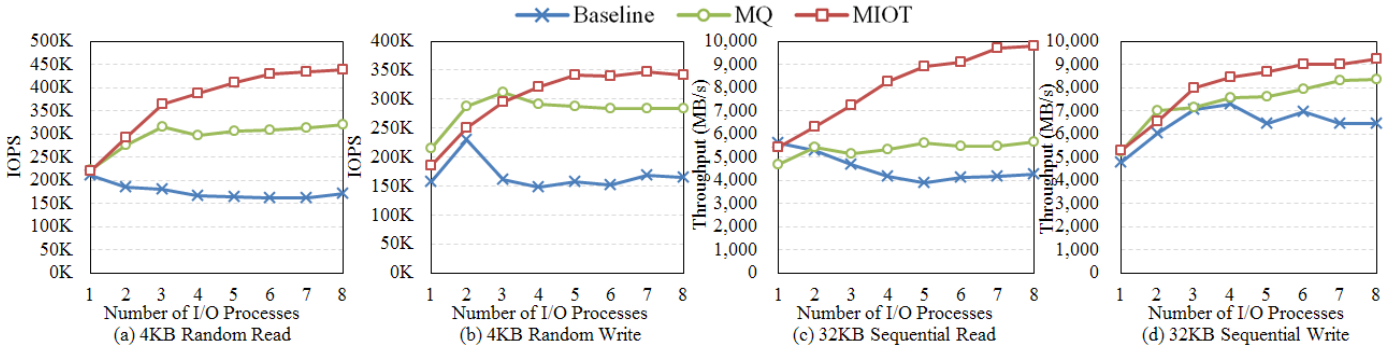

Fig. 11. Performance comparison on four types of I/O workloads among Baseline, MQ, and MIOT using a null block device.

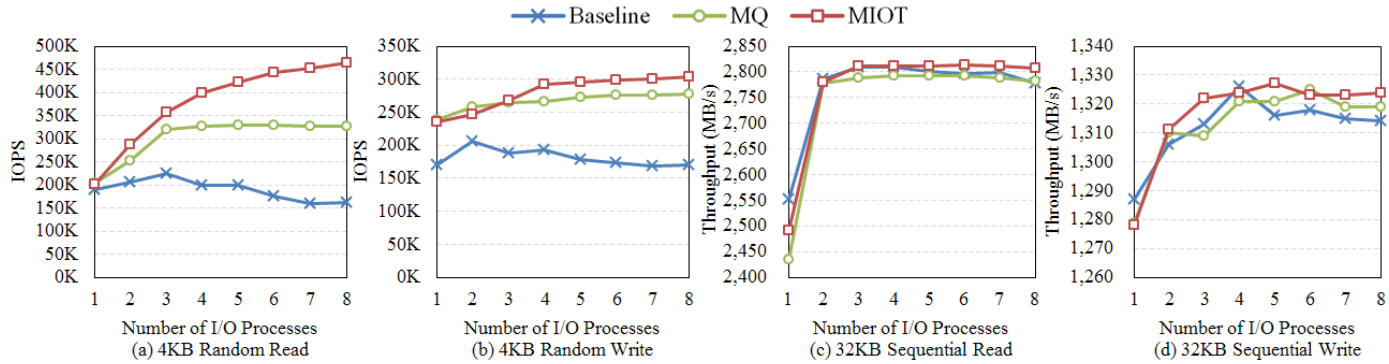

Fig. 12. Performance comparison on four types of I/O workloads among Baseline, MQ, and MIOT using an NVMe SSD.

also IOPS were worse than those of MIOT-w, MIOT-v, and MIOT. Furthermore, the MIOT-p method even has a few IO_INSTs. Thus, while it is true that the increased number of *exits* leads to performance degradations, this is not absolute in all cases.

H. Performance on Null Block Device

We evaluated and compared the performances of the three different designs including Baseline, MQ, and MIOT with a null block device, while varying the number of I/O-intensive processes from 1 to 8. The performance was measured in IOPS and throughput (MB/s) using the four types of I/O workloads illustrated in Section IV.C. First, as shown in Figure 11(a), IOPS gradually decreased in Baseline even though the number of I/O processes increased, while MQ and MIOT increased it. In particular, MIOT improved IOPS by up to 2.67x compared to Baseline and by up to 38% compared to MQ, and the performance on the other I/O workloads also increased significantly as shown in Figure 11(b), (c), (d). It shows that the I/O scalability issue is completely disappeared.

Actually, we expected that the performance of MQ would not improve due to the single I/O thread allocated in MQ; however, its performance was acceptable. This was because the I/O thread was assigned a number of CPUs by the CPU scheduling in the host. This unexpected scheduling gave a positive impact on performance, similar to the effect of the multiple I/O thread technique, but it also had a limitation as shown in the experimental result.

While we achieved excellent performance improvements, a performance limit was observed when the number of I/O processes went over four. This is not caused by the number of CPU cores, but largely due to basic overheads incurred by the virtualization layers. We attempted the same experiment on another system equipped with an octa-core CPU, and the result was very similar. Unfortunately, completely eliminating fundamental overheads was impossible in software-based I/O virtualization.

In conclusion, our approach sustains 440K IOPS in random read, 350K IOPS in random write, 9800 MB/s throughput of sequential read, and 9200 MB/s of sequential write. These results show the performance improvement of up to 167% compared to Baseline.

I. Performance on NVMe SSD

Finally, we conducted the same experiment with an NVMe SSD because the null block device is not perfectly identical to the NVMe SSD, even if the null block device can similarly simulate a multi-queue SSD.

Unlike the concern regarding the differences between the two devices, noteworthy variation was not observed in the experimental result, while some differences were observed. First, the IOPS of random read on the NVMe SSD was slightly higher than that on the null block device. To be specific, as shown in Figure 12(a), MIOT achieved approximately 460K IOPS in random read, and was improved by up to 187% compared to Baseline. Figure 12(b) also shows that the IOPS of MIOT was better than that of Baseline and MQ. On the other hand, surprisingly, the throughputs of sequential read (c) and sequential write (d) had comparatively not improved. According to our analysis, the NVMe SSD has relatively poor

throughput (3000 MB/s in sequential read and 1400 MB/s of sequential write), while high random read IOPS has about 750K. For this reason, the performance improvements are now concealed on the NVMe SSD. However, our design will be more advantageous because it is obvious that higher performance SSDs will be developed in the near future.

V. Related Work

Numerous studies have attempted to reduce the I/O performance gap between virtualized and non-virtualized systems. However, although high performance SSDs such as a multi-queue SSD were announced, studies on how the multi-queue SSD affects the virtualization framework are uncommon. Fortunately, some studies relevant to the multi-queue SSD have recently been published.

A recent study posed an I/O scalability problem in Linux block I/O layer caused by serious lock contentions, and proposed a new Linux block I/O layer [11] to solve the problem. The researchers designed two levels of queues which consist of software staging queues and hardware dispatch queues to diminish lock contentions and improve I/O parallelism. Even though the research was conducted in non-virtualized environments, this significantly motivated our desire to study the impact of a multi-queue SSD on virtualization.

Ming Lei revealed performance degradations with a multi-queue SSD in a virtualized environment for the first time [13]. He raised a lock contention problem produced by a single shared request queue in Virtio-Blk-Data-Plane, and proposed extended multiple request queues and two optimization schemes. However, unlike our approach, the study focused only on the single request queue, and ignored another problem caused by the single I/O thread. Moreover, the evaluation was conducted with a small number of request queues and I/O processes. Thus, it was not sufficient to verify I/O scalability with the scheme. We evaluated the output of the research in Section IV. The results of our experiments showed that our approach improved the performance by 38% compared to the work.

A recent work posed another performance degradation issue using four SSDs combined by RAID0 [14]. The main contribution was reducing the number of *exits* between the host and the guest via a pipelined polling I/O thread in Virtio-Blk-Data-Plane. Furthermore, they proposed a technique of multiple issues and multiple completions through multiple I/O threads for an NVMe SSD. Our approach has several key differences to this research. First, this work uses a polling mechanism as an operating mode of the I/O thread, and it was not impressive with multiple I/O threads as demonstrated in Section IV.E. Second, this work utilized only three I/O threads while our approach applies the per-vCPU I/O thread scheme. Lastly, we experimented not only with the NVMe SSD, which are evaluated in this work, but also with a null block device because it can validate the higher range of performance where the NVMe SSD is unable to reach.

VI. Conclusion

In this paper, we observed that existing virtualization technologies cannot guarantee the performance of guest machines when a multi-queue SSD is used as its secondary storage. This is because the guest machines suffer from lock contentions when issuing their I/O requests from the I/O virtualization framework to the multi-queue SSD. Furthermore, the guest machines cannot fully exploit multiple I/O queues in the host, which is caused by the semantic gap between the guest and the host machines. In order to reduce this semantic gap, we proposed a novel approach that efficiently distributes the lock contentions and improves the I/O parallelism by developing a new architecture including vCPU-dedicated queues and I/O threads with three optimization schemes.

We evaluated our approach on a null block device, which simulates a multi-queue virtual device by receiving I/O requests and acknowledging I/O completions immediately, as well as on a real NVMe SSD. Our experimental results with various I/O traces clearly show that IOPS performance was significantly improved by up to 2.67x, and that the throughput was enhanced by up to 132% compared to a state-of-the-art I/O virtualization technique, the Virtio-Blk-Data-Plane.

Acknowledgment

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2010-0020730) and this work was supported by ICT R&D program of MSIP/IITP. [10041244, SmartTV 2.0 Software Platform]. Young Ik Eom is the corresponding author of this paper.

References

[1] D. Abramson *et al*., "Intel virtualization technology for directed I/O," *Intel Technology J.*, vol. 10, no. 3, pp. 179-191, Aug. 2006.

[2] R. Mijat and A. Nightingale, "Virtualization is coming to a platform near you," ARM White Paper, 2011.

[3] Y. Dong *et al*., "High performance network virtualization with SR-IOV," *J. Parallel Distributed. Computing*, vol. 72, no. 1, pp. 1471-1480, Nov. 2012.

[4] *AMD I/O virtualization technology (IOMMU) specification* [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/488821.pdf

[5] *KVM live migration* [Online]. Available: http://www.linux-kvm.org/images/5/5a/KvmForum2007$Kvm_Live_Migration_Forum_2007.pdf

[6] C. Clark *et al*., "Live migration of virtual machines," in *Proc. 2nd Symp. on Networked Syst. Design & Implementation*, Boston, MA, 2005, pp. 273–286.

[7] R. Russell, "VIRTIO: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Syst. Review*, vol. 42, no. 5, pp. 95-103, Jul. 2008.

[8] A. Kivity *et al*., "KVM: the linux virtual machine monitor," in *Proc. Linux Symp.*, Ottawa, ON, 2007, pp. 225–230.

[9] N. Har'El *et al*., "Efficient and scalable paravirtual I/O system," in *Proc. USENIX Annu. Tech. Conf.*, San Jose, CA, 2013, pp. 231-242.

[10] *NVM Express specification 1.2* [Online]. Available: http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_2-Gold-20141209.pdf

[11] M. Bjørling *et al*., "Linux block io: introducing multi-queue ssd access on multi-core systems," in *Proc. 6th Int. Syst. and Storage Conf.*, Haifa, 2013, pp. 22:1-22:10.

[12] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, Anaheim, CA, 2005, pp. 41-46.

[13] *Virtio Blk multi-queue conversion* [Online]. Available: http://www.linux-kvm.org/images/6/63/02x06a-VirtioBlk.pdf

[14] M. Oh *et al*., "Enhancing the I/O system for virtual machines using high performance SSDs," in *Proc. IEEE int. Performance Computing and Commun. Conf.*, Austin, TX, 2014, pp. 1-8

[15] *FIO: flexible IO tester* [Online]. Available: http://freecode.com/projects/fio

[16] K. Eshghi and R. Micheloni, "SSD architecture and PCI Express interface," in *Inside Solid State Drives (SSDs)*, Dordrecht, Netherlands: Springer Netherlands, 2013, pp. 19-45.

[17] *KVM virtualized I/O performance* [Online]. Available FTP: public.dhe.ibm.com Directory: /linux/pdfs File: KVM_Virtualized_IO_Performance_Paper_v2.pdf

[18] W. Shin *et al*., "OS I/O path optimizations for flash solid-state drives," in *Proc. USENIX Annu. Tech. Conf.,* Philadelphia, PA, 2014, pp. 483-488

[19] *Null block device driver* [Online]. Available: https://www.kernel.org/doc/Documentation/block/null_blk.txt

[20] A. Gordon *et al*., "Towards exitless and efficient paravirtual I/O," in *Proc. 5th Int. Syst. and Storage Conf.*, Haifa, 2012, pp. 10:1-10:6.

[21] A. Landau *et al*., "Splitx: split guest/hypervisor execution on multi-core," in *Proc. 3rd Conf. on I/O Virtualization*, Portland, OR, 2011, pp. 1.

[22] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proc. 12th Int. Conf. on Archtectural Support for Programming Languages and Operation System*, San Jose, CA, 2006, pp. 2-13.

[23] *Perf: linux profiling with performance counters* [Online]. Available: https://perf.wiki.kernel.org

[24] J. Axboe, "Linux block IO—present and future," in *Proc. Linux Symp.*, Ottawa, ON, 2004, pp. 51–61.