electronics

Article

# Direct-Virtio: A New Direct Virtualized I/O Framework for NVMe SSDs

Sewoog Kim [1], Heekwon Park [2] and Jongmoo Choi [1,*]

1 Department of Software, Dankook University, Yongin 16890, Korea; ksu1110@dankook.ac.kr
2 Memory Solutions Lab., Samsung Semiconductor Inc., San Jose, CA 95134, USA; heekwon.p@samsung.com
* Correspondence: choijm@dankook.ac.kr; Tel.: +82-31-8005-3242

**Abstract:** Virtualization is a core technology for cloud computing, server consolidation and multi-platform support. However, there is a concern regarding performance degradation due to the duplicated I/O stacks virtualization environments. In this paper, we propose a new I/O framework, we refer to it as Direct-Virtio, that manipulates storage directly, which makes it feasible to avoid the duplicated overhead. In addition, we devise two novel mechanisms, called vectored I/O and adaptive polling, to process multiple I/O requests collectively and to check I/O completion efficiently. Real implementation-based evaluation shows that our proposal can enhance performance for both micro and macro benchmarks.

**Keywords:** I/O virtualization; NVMe SSD; direct I/O; implementation; evaluation

**Citation:** Kim, S.; Park, H.; Choi, J. Direct-Virtio: A New Direct Virtualized I/O Framework for NVMe SSDs. *Electronics* **2021**, *10*, 2058. https://doi.org/10.3390/electronics10172058

Academic Editor: Manuel E. Acacio

Received: 31 July 2021
Accepted: 24 August 2021
Published: 26 August 2021

## 1. Introduction

Virtualization is a fundamental technology for cloud computing and multi-platform support [1–4]. By creating multiple virtual machines from a physical machine, it can enhance the server utilization and can allow pay-as-you-go service. In addition, by utilizing various virtualization solutions such as Virtualbox [5] and KVM [6], it can provide multiple operating systems or multiple security domains on a laptop and mobile device [7].

There are a variety of virtualization techniques, which can be classified into three categories: CPU virtualization [8–10], memory virtualization [11–13] and I/O virtualization [14–18]. For example, Borrowed Virtual Time (BVT) is a scheduling algorithm for allocating CPUs among virtual machines in a fair-share manner while providing fast dispatch [8]. Ballooning is a memory management scheme that adjusts the memory size of a virtual machine dynamically according to its usage [11].

Virtio is a well-known framework for I/O virtualization [14]. It consists of two components, frontend and backend, which are located in a virtual machine and a host machine (or hypervisor), respectively. The advantage of this approach is that it can make use of the existing I/O stack, including file system, block layer and driver, without any special hardware support (e.g., SR-IOV [16] or Intel VT-d [19]). However, the downside is that it causes performance drop since I/Os need to go through the I/O stacks twice, one in a virtual machine and the other in a host machine, which will be further discussed in Section 2.

To overcome the performance drop, this paper proposes a novel I/O framework, called Direct-Virtio, that accesses NVMe SSDs directly so that it can avoid the host machine I/O stack overhead. Specifically, it maps hardware queues supported by NVMe SSDs into user space, enabling to access storage without intervention of the host machine I/O stack. Furthermore, it provides new interfaces to create a new queue and assign it to a virtual machine at the user level.

In addition, we devise two mechanisms, called vectored I/O and adaptive polling, for improving performance further. The vectored I/O mechanism is designed to decrease the NVMe command processing overhead by processing multiple I/Os collectively. The adaptive polling mechanism separates a polling thread from a main thread and schedules it in a timely manner to reduce the polling overhead.

We implement our proposed Direct-Virtio in QEMU [20] on KVM environment [6]. Then, we evaluate its performance using two benchmarks, Fio [21] and Filebench [22]. Evaluation results with a real NVMe SSD device have shown that Direct-Virtio can improve performance by up to 20% with an average of 12%, compared with the existing Virtio framework.

The contributions of this paper can be summarized as follows.

- We observe that the existing I/O virtualization framework suffers from performance degradation due to the duplicated I/O stacks.
- We design a new I/O virtualization framework with three concepts; user-level direct access, vectored I/O and adaptive polling.
- We present several quantitative evaluation results with diverse configurations such as different number of jobs, queue depths and reference patterns.

The rest of this paper is organized as follows. Section 2 introduces background and motivation of this study. We explain our design in Section 3. Evaluation results are discussed in Section 4. Related works are surveyed in Section 5. Finally, we present the conclusion and future work in Section 6.

## 2. Background

In this section, we first describe the features of NVMe SSDs and the approaches for I/O virtualization. Then, we discuss our observation that motivates this study.

### *2.1. NVMe SSDs and I/O Virtualization*

NVMe SSDs (Non-Volatile Memory Express Solid State Drives) are a new type of SSDs that makes use of the NVMe standard interface for supporting high performance storage on the PCI Express interconnect [23]. Since they can handle I/Os with a high throughput and can provide large capacity, they become popular storage choices for data center and cloud computing [24,25].

NVMe SSDs support multiple hardware queues and multiple commands per each queue (e.g., 64 K queues and 64 K commands per queue). Queues are further divided into submission queues and completion queues for sending commands to a device and for receiving results, which can be mapped flexibly into memory address space. Besides, they support MSI-X (Message Signaled Interrupt eXtended) for efficient interrupt handling and aggregation. The multiple queue support and flexibility make NVMe SSDs as an attractive candidate for I/O virtualization.

There are two distinct approaches for I/O virtualization, software-based [14,25–28] and hardware-based [15–17]. Hardware-based approaches rely on hardware supports such as IOMMU (I/O Memory Management Unit) for mediating DMAs by I/O devices and SR-IOV (Single Root I/O Virtualization) for abstracting a physical device into multiple virtual devices. On the contrary, software-based approaches do not require any hardware supports. They employ a special virtualization software called hypervisor [1] or make use of virtualization functionalities provided by an operating system in a host machine [6].

Hardware-based approaches and software-based approaches have their own advantages and disadvantages in terms of cost, flexibility, and performance. Specifically, software-based approaches are cost-effective since they are based on commodity devices, not requiring any special hardware support [28]. In addition, they can share and/or isolate a device among multiple virtual machines flexibly [25]. However, there is a concern about performance degradation. This study investigates how to solve such performance degradation, so we focus on software-based approaches. Note that our proposed techniques, vectored I/O and adaptive polling, can be used for not only software-based approaches, but also hardware-based ones since they have a potential to reduce the NVMe command processing overhead for both environments.

Figure 1 shows the structure of Virtio, a well-known software-based I/O virtualization framework [14]. It consists of two layers, host machine and virtual machine. An operating system in the host machine, called HostOS or Host-level OS, takes charge of the virtual machine management such as creation and resource allocation. An operating system in the virtual machine, also known as GuestOS or VM-level OS, provides computing environment using the virtual resources created by HostOS.

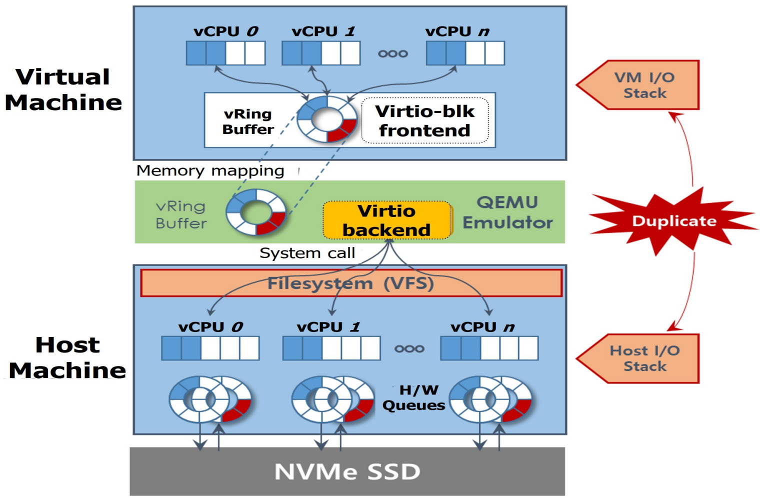

**Figure 1.** Virtio I/O Virtualization framework: It consists of two layers, virtual machine and host machine, which are connected via QEMU.

A storage device driver is implemented in two components, frontend and backend. The frontend component, called Virtio-blk, is located in the virtual machine while the backend component, Virtio-backend, is located in QEMU, as a user process in the host machine. The frontend and backend are connected via a ring buffer, called vRing, used for transferring or receiving I/O requests or results [29]. The ring buffer is a shared region, not requiring memory copy for communication between the frontend and backend.

When a user sends an I/O request, it goes through the I/O stack in the virtual machine including file system and block layer, and finally arrives at the frontend component. Then, it is transferred to the backend via vRing. The backend invokes system calls to send the request into the Host-level OS. Then, the request goes through the I/O stack in the host machine and finally arrives NVMe hardware queues. Note that the queues in the current Linux Kernel are managed per core to avoid locking and cache collision [30].

## 2.2. Motivation

Figure 2 presents the I/O virtualization overhead. In this experiment, we execute the Fio benchmark [21] with various number of queue depths and reference patterns (e.g., random/sequential and read/write). Details of our experimental environment and workloads are discussed in Section 4.

To assess the I/O virtualization overhead, we measure performance at two levels, virtual machine and host machine. At the host machine level (hereafter, host-level), I/Os go through the host I/O stack only. However, at the virtual machine level (VM-level), I/Os undergo not only the host I/O stack but also the VM I/O stack, as explained in Figure 1. Therefore, the differences between two levels reveal the overhead of the duplicated I/O stacks.

Figure 2 exhibits that VM-level I/Os perform worse than host-level I/Os. There exist several cases where the performance degradation is more than 4 times (e.g., for the random read reference pattern with the queue depth of 4). Our sensitivity analysis reveals that there are three reasons for this drop. First, since VM-level I/Os need to go through not only the VM I/O stack, but also the host I/O stack, they suffer from the overhead to pass the I/O stacks twice.

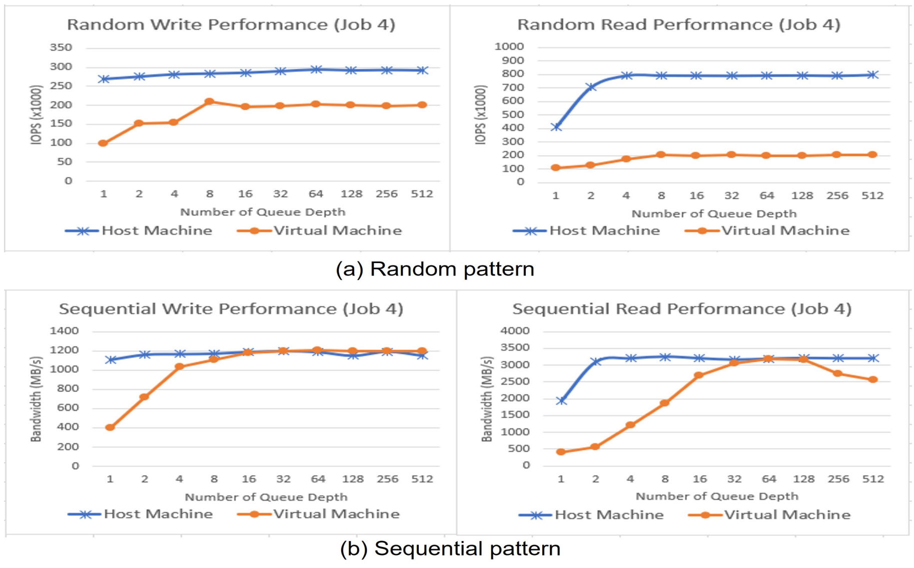

**Figure 2.** I/O virtualization overhead: Performance degradation is due to the duplicated I/O stacks.

Second, we collect I/O traces and observe that there is a delay between an I/O completion in an NVMe device and a notification of the I/O completion to a virtual machine. This is because the I/O completion monitored by a host NVMe driver is informed to the Virtio backend via a file system event notification mechanism asynchronously [31], which creates a problematic situation where the backend does not identify or notify the I/O completion to the virtual machine via vRing [29], even though the request has already finished. This observation leads us to design our adaptive polling mechanism.

The third reason is that I/O patterns, sequential or random, affect the performance drop considerably. Operating systems try to aggregate multiple I/O requests into a single NVMe command in order to reduce the I/O processing overhead. For the sequential pattern and when the queue depth is large enough to hold multiple requests (e.g., 16 or larger), the VM I/O stack can aggregate I/Os and transfer them to the host I/O stack as a single request, which mitigates the host I/O stack overhead. On the contrary, for the random pattern or for the sequential pattern with a small queue depth, each I/O is transferred and processed individually in the host I/O stack, resulting in considerable overhead, as shown in Figure 2. It implies that, when we design Direct-Virtio, we need to carefully consider how to retain the benefit of the I/O aggregation, which inspires us to develop the vectored I/O mechanism.

## 3. Design

In this section, we first explain how we design Direct-Virtio to provide user-level direct access for avoiding the host I/O stack overhead. Then, we discuss two challenges and how we address them using vectored I/O and adaptive polling.

Figure 3 illustrates the conceptual differences between the existing Virtio and our Direct-Virtio framework. In Virtio, when an I/O request arrives, the Virtio backend invokes system calls, either using traditional I/O or asynchronous I/O (AIO) [31], and sends the request into the host I/O stack. Then, the host I/O stack processes the request including converting it to an NVMe command, issuing it into a submission queue and checking a completion queue. Finally, after completing the request, the result is informed to the backend via a file system event notification mechanism.

As we have discussed with Figure 2, Virtio suffers from the performance degradation due to the duplicated I/O stack overhead. To overcome this problem, our proposed Direct-Virtio accesses NVMe devices directly at the user level. For this user-level direct access capability, Direct-Virtio equips with new interfaces to create a queue and to map the created queue into a user address space.

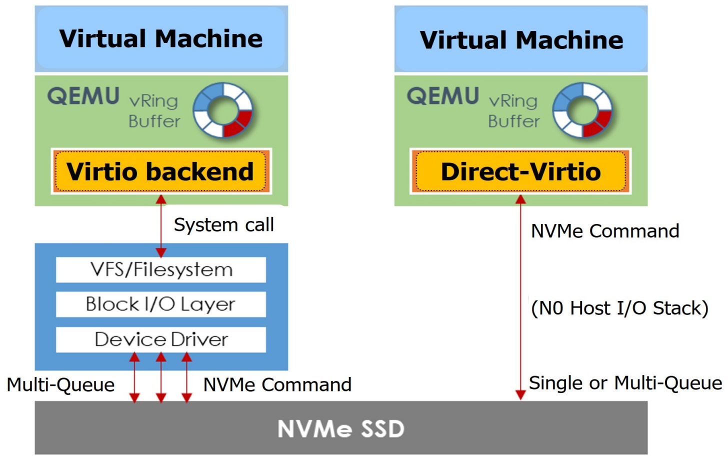

**Figure 3.** Direct-Virtio concept: Accessing NVMe SSDs directly allows to avoid the duplicated I/O stack overhead.

NVMe devices support several admin commands such as create/delete a queue, set/get features, and asynchronous event request [32]. The NVMe driver in the host I/O stack has procedures that utilize these commands to manage submission and completion queues. We add a new system call using *ioctl()* so that Direct-Virtio can utilize them at the user level for creating a queue. In addition, we map the memory address of the created queue into user space using *dma_common_mmap()* so that Direct-Virtio can send an I/O command into a submission queue and check a result from a completion queue. As the result, Direct-Virtio can handle devices directly instead of invoking the *read()* and *write()* system calls to serve I/O requests, which implies that it bypasses the host I/O stack.

In this study, we create a single pair of queues, one submission queue and one completion queue, per each virtual machine. This is because the goal of this study is investigating the duplicated I/O stack overhead, not the differences between single and multiple queues. We expect that utilizing multiple queue pairs in a virtual machine is feasible by distributing I/O requests across them, like Kim's study [24]. Current version of Direct-Virtio supports simultaneous execution of multiple virtual machines where each virtual machine uses different queues.

User-level direct accessing raises two challenges. The first is a security challenge. Malicious users can write an incorrect or harmful command, which may cause device malfunctions and errors. To alleviate this risk, Direct-Virtio checks a permission and allows only root user to create and map a queue into user space. However, there may exist a security vulnerability in current design and we leave this challenge as a future work.

The second challenge is limited functionalities. The host I/O stack has several useful functionalities such as I/O scheduling, caching and I/O aggregation. However, when we bypass the stack, we cannot utilize these functions. As discussed with Figure 2, I/O aggregation is critical to reduce the I/O processing overhead. In addition, polling is indispensable for fast storage devices [33]. Therefore, we decide to develop these two mechanisms in Direct-Virtio.

Figure 4 presents how we design our user-level I/O aggregation mechanism. An I/O request transferred from the Virtio-blk frontend is in the form of a scatter-gather list (denoted as SG list in the figure) [29]. When an I/O is not aggregated in the VM stack and is transferred individually, the list has only one entry for the I/O. Otherwise, when I/Os are aggregated, the list has multiple entries where each entry corresponds to each I/O. In the traditional Virtio, the list is passed as an argument to a system call. However, in our Direct-Virtio, it needs to be converted to an NVMe command.

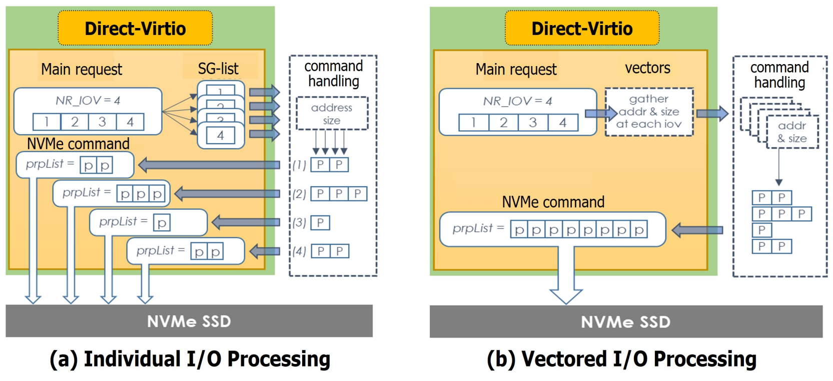

**Figure 4.** VectoredI/O mechanism: It processes multiple I/Os using a single NVMe command.

The initial version of Direct-Virtio handles each entry in the list separately, converting each entry into an NVMe command as shown in Figure 4a. This version is easy to implement. However, it incurs substantial NVMe command processing overhead, which makes it impossible to reap the benefit of user-level direct access. Therefore, we implement a new version, called vectored I/O processing mechanism as shown in Figure 4b, that converts all entries in the list into a single NVMe command. It first maps the list into an I/O vector, a structure used for handling I/Os in QEMU. Then, we construct a Physical Region Pages (PRP) list, which is used as an argument when we insert an NVMe command into a submission queue. This new implementation can keep the advantage of I/O aggregation, enabling us to fully obtain the benefit of user-level direct access.

The second mechanism that we develop for Direct-Virtio is adaptive polling. In Virtio, there are two synchronization points. One is between the frontend and backend to transfer I/O requests and notifications via vRing. The other is between the backend and the NVMe driver in the host stack to invoke system calls and to be informed for completed commands. Our analysis reveals that these two separate points often cause a problematic situation where the notification of a request to the frontend is delayed even though the request has already completed in NVMe devices.

To overcome this problem, we integrate the two points by employing user-level polling. Specifically, Direct-Virtio polls its completion queue directly and replies to a notification as soon as it detects a completed request. The original Virtio backend code utilizes a main loop for processing I/O requests sent from the frontend and for checking I/O completions by polling events that are informed by the host I/O stack. For Direct-Virtio, we modify the polling part, checking the completion queue directly instead of the file system events. However, this modification increases CPU utilization considerably (e.g., up to 70% in our experiment) due to the continuous checking. To alleviate this overhead, we introduce adaptability in Direct-Virtio.

Figure 5 shows our adaptive polling mechanism. It creates a separated thread for polling purpose, while utilizing the main thread for request processing purpose only. When the main thread sends a request to NVMe SSDs, it also sends a signal to the polling thread so that it can be woken up after a predefined sleep time. After this time, it resumes and conducts polling to check completion. When completed, the polling thread notifies it to the main thread using the *eventfd* facility and goes back to sleep again. Finally, the main thread does post-processing of the completed request.

Now, the question is how to determine the sleep time. By introducing the time, Direct-Virtio can reduce the polling overhead by checking the completion queue in a timely manner. However, if the time is too long, it may deteriorate the latency of a request. On the contrary, if the time is too short, it may cause unnecessary monitoring even though the completion does not occur. In this study, we use a history based prediction that adjusts the time dynamically. If the polling thread has to do unnecessary checking, it increases the time. Otherwise, the completion has already occurred when the thread is woken up, it decreases the time. Note that our adaptive polling mechanism can be utilized effectively not only for Direct-Virtio, but also for the traditional Virtio and other polling schemes for fast storage devices.

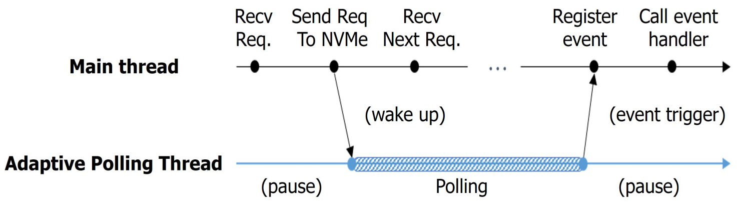

**Figure 5.** Adaptive polling mechanism: A separated polling thread is woken up with the consideration of the expected completion time.

## 4. Evaluation

This section first explains our experimental environment. Then, it presents performance evaluation results for both micro and macro benchmarks.

Table 1 summarizes our experimental environment. Hardware platform consists of 8 CPUs (16 CPUs when hyperthread is enabled), 64 GB DRAM and 1.2 TB NVMe SSD device. On this platform, we install the Linux kernel version 4.4.0 for the host OS. We implement Direct-Virtio in the QEMU [20] version 2.8.0. Then, we create a virtual machine with 4 CPUs, 32GB DRAM and two NVMe hardware queues, one for submission and the other for completion. Finally, we utilize two benchmarks, Fio [21] and Filebench [22], for performance evaluation.

**Table 1.** Experimental environment.

| Item | Specification |
|---|---|
| Host machine | Intel i7-6700 (8 core), 64 GB DRAM, 1.2 TB NVMe SSD |
| Linux kernel | 4.4.0 (Ubuntu-14.04.5 LTS), KVM virtualization |
| QEMU | 2.8.0 |
| Virtual machine | 4 CPUs, 32GB DRAM, NVMe hardware queues |
| Benchmarks | Fio, Filebench |

Figures 6 and 7 present the performance evaluation results for the random write and read pattern, respectively. In this experiment, we use the Fio benchmark with 4 KB block size and asynchronous I/O mode. We change the number of queue depths ranging from 1 to 512 and the number of jobs from 1 to 8.

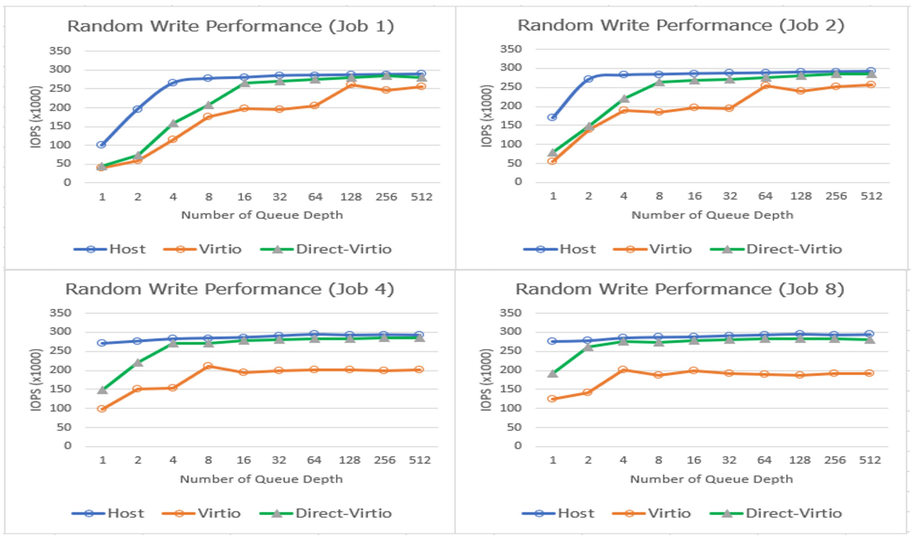

**Figure 6.** I/O performance comparison between Virtio and Direct-Virtio for random write pattern.

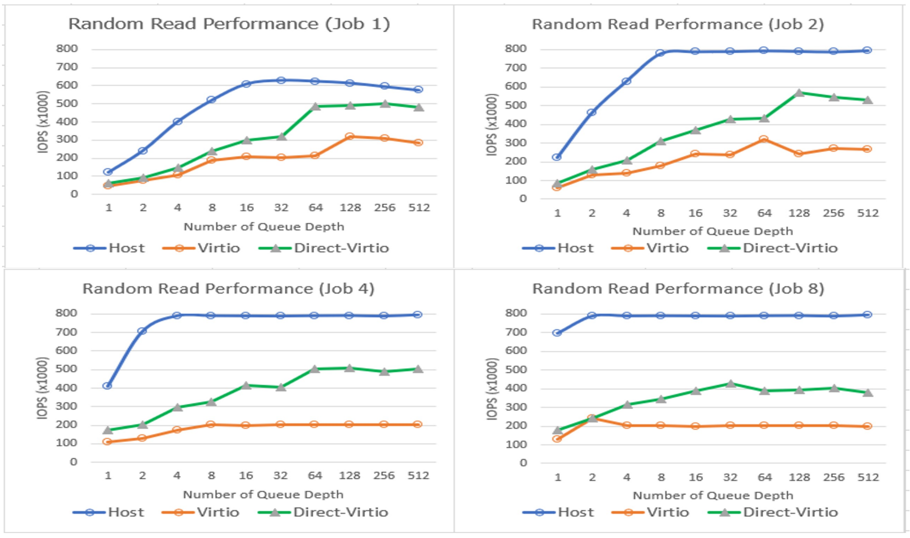

**Figure 7.** I/O performance comparison between Virtio and Direct-Virtio for random read pattern.

From these results, we can observe that our proposed Direct-Virtio indeed gives a positive impact on performance. It improves IOPS (I/O operation Per Second) by up to 1.85 times (8 jobs and 2 queue depths) with an average of 1.35 times for the random write pattern. For the random read pattern, it enhances by up to 2.52 times (4 jobs and 128 queue depths) and 1.75 times on average. We also notice that Direct-Virtio exhibits performance comparable to the host machine for the random write pattern. However, the differences between two for the random read pattern are around 300 K, meaning that there is still room for further optimization of this pattern.

The performance evaluation results for the sequential write and read pattern are presented in Figures 8 and 9, respectively. Direct-Virtio enhances throughput by up to 1.6 times (2 jobs and 1 queue depth) with an average of 1.09 times for the sequential write pattern. It also increases by up to 1.72 times (4 jobs and 2 queue depths) and 1.17 times on average for the sequential read pattern.

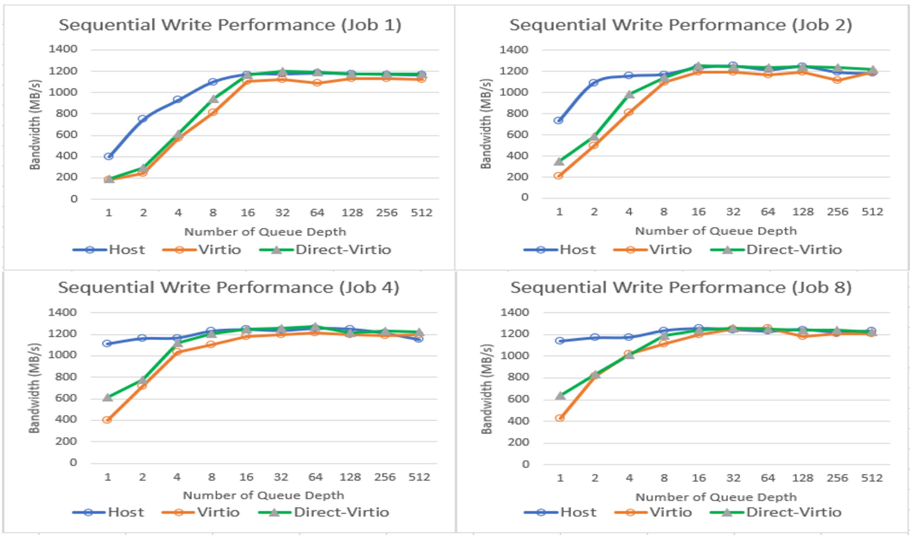

**Figure 8.** I/O performance comparison between Virtio and Direct-Virtio for sequential write pattern.

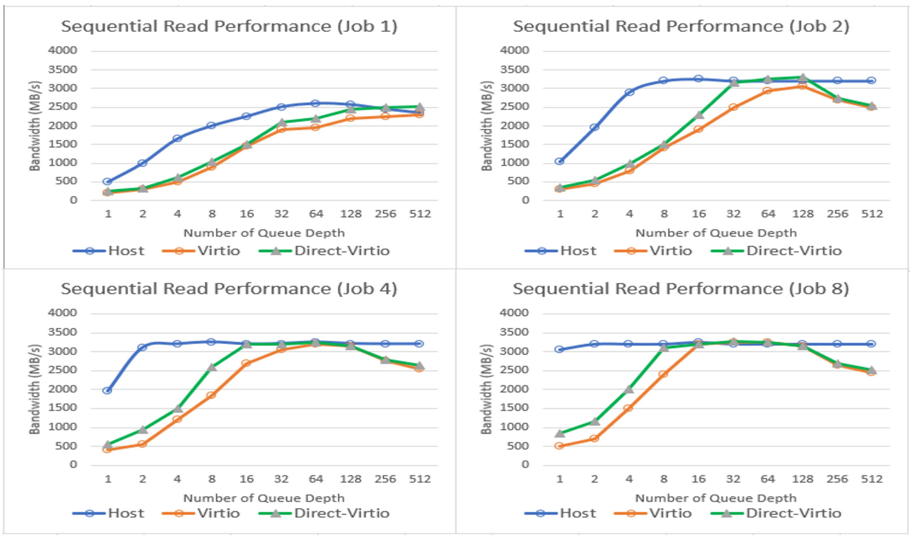

**Figure 9.** I/O performance comparison between Virtio and Direct-Virtio for sequential read pattern.

Evaluation results show that Direct-Virtio reaps more performance achievement for the random pattern, compared to the sequential pattern. One reason is that there are big performance gaps in the random pattern, as shown in Figure 2, making it easier to boost performance. Our detailed analysis reveals that most I/Os transferred from a virtual machine to the backend are 64 KB or larger for the sequential pattern while they are mostly 4 KB for the random pattern. It implies that most I/Os of the sequential pattern are already grouped within the I/O stack of a virtual machine, making the number of I/O requests relatively smaller. The large number of I/O requests in the random pattern makes it possible for Direct-Virtio to obtain higher performance gain by employing user-level direct access. In addition, the adaptive polling mechanism is more effective for the random pattern due to the large number of I/O requests. Finally, the vectored I/O mechanism can maintain the benefit of the I/O aggregation, which allows Direct-Virtio to show performance comparable to the host machine when the queue depth is larger than 8.

The results also uncover that Direct-Virtio obtains more performance gain for the read workload than the write one. This is because writes are slower than reads in NVMe SSDs, meaning that the software processing time takes more portions in the latency of the read request. Therefore, reducing the software overhead can generate more benefit for the read workload.

Figure 10 presents the performance evaluation results for diverse workloads in Filebench, summarized in Table 2. It demonstrates that Direct-Virtio can improve the performance of real applications by up to 20% and 12% on average. The varmail workload, which has the random pattern, shows the best enhancement improvement while the video server workload, that has the sequential pattern, shows the least.

**Table 2.** Characteristics of Filebench workloads.

| Workload | Block Size Rate | Random Rate |
|---|---|---|
| File server | 4 KB(26%) 60 KB (16%) 64 K (26%) | Mix |
| Oltp | 4 KB(98%) | Random |
| Varmail | 4 KB(60%) 8 KB, 12 LB, 16 KB (each 7%) | Random |
| Video server | 60 KB(47%) 64 KB (47%) | Sequential |
| Web server | 4 KB(11%) 60 KB (22%) 64 K (28%) | Mix |
| Mongo | 4 KB(48%) 64 K (18%) | Mix |

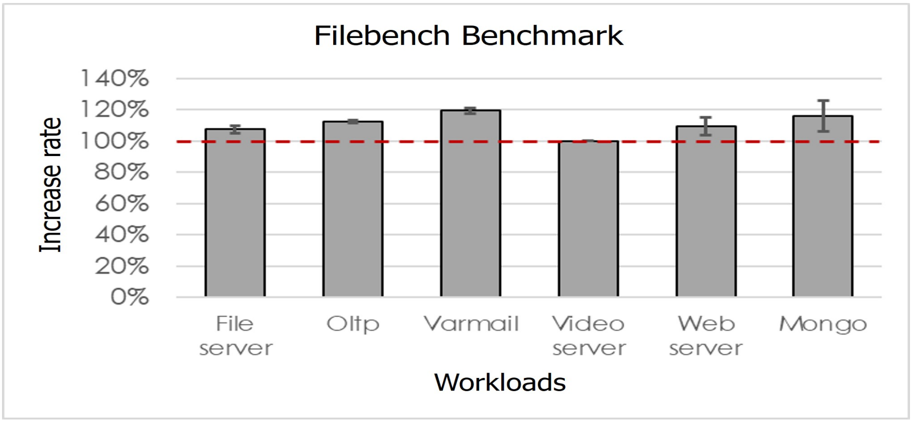

**Figure 10.** I/O performance comparison between Virtio and Direct-Virtio for macro workloads (100% baseline is Virtio).

Overall, the performance improvement of Direct-Virtio comes from three sources. First is the user-level direct access scheme, which enables us to eliminate the host I/O stack overhead. The second source is the adaptive polling mechanism that can reduce the unnecessary delay by identifying completed requests and notifying them to a virtual machine in a timely manner. The third source is the vectored I/O mechanism. By keeping the advantage of the I/O aggregation, it can provide performance comparable to the host machine for the sequential pattern.

## 5. Related Work

There are various proposals that try to access NVMe SSDs at the user level [24,34,35]. Kim et al. designed a user-level I/O framework, called NVMeDirect, that allows user applications to access commercial NVMe SSDs directly without any hardware modification [24]. SPDK (Storage Performance Development Kit) is a set of tools and libraries that are popularly used for implementing high-performance user-mode storage applications [34]. Kourtis et al. presented a novel key-value store, uDepot, that manipulates fast NVMe block devices at the user level [35].

Enhancing I/O performance virtualized environments are also studied actively [14–18,25–28]. Tian et al. propose a new architecture, called coIOMMU, that integrates the vIOMMU (virtual I/O Memory Management Unit) and cooperative DMA buffer to support direct I/O efficiently [15]. Xue et al. design a new NVMe virtualization system for enhancing reliability by introducing three techniques, namely cross-process journal, isolation-based failure recovery, and fast restart [25]. Kim et al. identify the problems in I/O virtualization, lock contention and limited parallelism, and devise vCPU-dedicate queues and I/O threads to overcome these problems [27].

Two studies are closely related to our work. Peng et al. design MDev-NVMe that makes use of a mediated pass-through mechanism to achieve high throughput and low latency based on native NVMe driver in a virtual machine [26]. They discover that a virtual machine can achieve 50% of the host-level performance due to the duplicated I/O stack overhead, which is also observed in our study.

Yang et al. propose a novel I/O service, called SPDK-vhost-NVMe, to reduce the I/O virtualization overhead [18]. They identify that the virtualization overhead comes from three parts, namely in guest OS, host OS and context switch (e.g., VM_Exit) between them. To shrink this overhead, they make use of user space NVMe drivers and design collaboration techniques. Their motivation and approach are similar to ours. However, they utilize SPDK for supporting user-level NVMe management whereas we implement new interfaces for this management and devise two new mechanisms.

## 6. Conclusions

This study proposes a new I/O virtualization framework, called Direct-Virtio. It makes use of the user-level storage direct access to avoid the duplicated I/O stack overhead. In addition, it introduces the vectored I/O mechanism to process I/Os in a batch style and the adaptive polling mechanism to check completion queues at an appropriate time. Real implementation based evaluation results show that Direct-Virtio outperforms the existing Virtio framework.

There are two research directions as future work. The first one is developing an I/O analysis tool for virtualized environments. Blktrace is a well-known tool for investigating I/O characteristics in non-virtualized environments [36]. However, while performing this study, we realize that there is no such tool virtualized environments. We are currently developing a tool that can uncover the virtual machine and host machine I/O behaviors in an integrated manner. The second direction is extending Direct-Virtio for supporting various QoS (Quality of Service) among virtual machines.

**Author Contributions:** Conceptualization, J.C. and H.P.; methodology, S.K.; software, S.K.; validation, S.K.; writing—original draft preparation, J.C. and H.P.; writing—review and editing, J.C.; supervision, J.C.; project administration, J.C. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01475, (SW StarLab) Development of Novel Key-Value DB for Unstructured Bigdata) and by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2020-0-00985, Development of Lightweight System Software for Edge Server System Resource Management and Control).

**Acknowledgments:** The authors appreciate all the reviewers and editors for their precious comments and work on this article.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I.; Warfield, A. Xen and the Art of Virtualization. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing (Lake George), NY, USA, 19–22 October 2003.
2. Younge, A.; Henschel, R.; Brown, J.; Laszewski, G.; Qiu, J.; Fox, G. Analysis of Virtualization Technologies for High Performance Computing Environments. In Proceedings of the IEEE 4th International Conference on Cloud Computing (Cloud), Washington, DC, USA, 4–9 July 2011.
3. Muller, T.; Knoll, A. Virtualization Techniques for Cross Platform Automated Software Builds, Tests and Deployment. In Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA), Porto, Portugal, 20–25 September 2009.
4. Lee, S.-H.; Kim, J.-S.; Seok, J.-S.; Jin, H.-W. Virtualization of Industrial Real-Time Networks for Containerized Controllers. *Sensors* **2019**, *19*, 4405. [CrossRef] [PubMed]
5. An Oracle White Paper. Oracle VM VirtualBox Overview. Available online: https://www.oracle.com/assets/oracle-vm-virtualbox-overview-2981353.pdf (accessed on 27 July 2021).
6. Kivity, A.; Kamay, Y.; Laor, D.; Lublin, U.; Liguori, A. KVM: The Linux Virtual Machine Monitor. In Proceedings of the Linux Symposium, Ottawa, ON, Canada, 27–30 July 2007.
7. Shuja, J.; Gani, A.; Bilal, K.; Khan, S. A Survey of Mobile Device Virtualization: Taxonomy and State-of-the-Art. *ACM Comput. Surv.* **2016**, *49*, 1. [CrossRef]
8. Cherkasova, L.; Gupta, D.; Vahdat, A. Comparison of the Three CPU Schedulers in Xen. Available online: https://www.hpl.hp.com/personal/Lucy_Cherkasova/papers/per-3sched-xen.pdf (accessed on 27 July 2021).
9. A VMware White Paper. VMware® vSphere™: The CPU Scheduler in VMware ESX® 4.1. Available online: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmw_vsphere41_cpu_schedule_esx-white-paper.pdf (accessed on 27 July 2021).
10. Wang, H.; Isci, C.; Subramanian, L.; Choi, J.; Qian, D.; Mutlu, O. A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters. In Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Istanbul, Turkey, 14–15 March 2015.
11. Waldspurger, C. Memory Resource Management in VMware ESX Server. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, USA, 9–11 December 2002.

12. Gandhi, J.; Basu, A.; Hill, M.; Swift, M. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Cambridge, UK, 13–17 December 2014.
13. Jia, G.; Han, G.; Wang, H.; Yang, X. Static Memory Deduplication for Performance Optimization in Cloud Computing. *Sensors* **2017**, *17*, 968. [CrossRef] [PubMed]
14. Russell, R. Virtio: Towards a de facto standard for virtual I/O devices. *ACM SIGOPS Oper. Syst. Rev.* **2008**, *42*, 5. [CrossRef]
15. Tian, K.; Zhang, Y.; Kang, L.; Zhao, Y.; Dong, Y. coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O. In Proceedings of the 2020 USENIX Annual Technical Conference (ATC), Virtual Event (online), 15–17 July 2020.
16. Muench, D.; Isfort, O.; Mueller, K.; Paulitsch, M.; Herkersdorf, A. Hardware-Based I/O Virtualization for Mixed Criticality Real-Time Systems Using PCIe SR-IOV. In Proceeding of the IEEE 16th International Conference on Computational Science and Engineering, Sydney, Australia, 3–5 December 2013.
17. Williamson, A. Vfio: A User's Perspective. In KVM Forum. 2012. Available online: https://www.linux-kvm.org/images/b/b4/2012-forum-VFIO.pdf (accessed on 27 July 2021).
18. Yang, Z.; Liu, C.; Zhou, Y.; Liu, X.; Cao, G. SPDK Vhost-NVMe: Accelerating I/Os in Virtual Machines on NVMe SSDs via User Space Vhost Target. In Proceedings of the IEEE 8th International Symposium on Cloud and Service Computing (SC2), Paris, France, 18–21 November 2018.
19. An Intel White Paper. Enabling Intel® Virtualization Technology Features and Benefits. Available online: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf (accessed on 27 July 2021).
20. Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the 2005 USENIX Annual Technical Conference (ATC), Anaheim, CA, USA, 10–15 April 2005.
21. Flexible I/O. Available online: https://fio.readthedocs.io/en/latest/fio_doc.html (accessed on 27 July 2021).
22. Tarasov, V.; Zadok, E.; Shepler, S. Filebench: A Flexible Framework for File System Benchmarking. *Login Usenix Mag.* **2016**, *41*, 1.
23. NVMe Specification. Available online: https://nvmexpress.org/specifications/ (accessed on 27 July 2021).
24. Kim, H.; Lee, Y.; Kim, J. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems, Santa Clara, CA, USA, 12–14 July 2017.
25. Xue, S.; Zhao, S.; Chen, Q.; Deng, G.; Liu, Z.; Zhang, J.; Song, Z.; Ma, T.; Yang, Y.; Zhou, Y.; et al. Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure. In Proceedings of the 2020 USENIX Annual Technical Conference (ATC), Virtual Event, 15–17 July 2020.
26. Peng, B.; Zhang, H.; Yao, J.; Dong, Y.; Xu, Y.; Guan, H. MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through. In Proceedings of the 2020 USENIX Annual Technical Conference (ATC), Boston, MA, USA, 11–13 July 2018.
27. Kim, T.; Kang, D.; Lee, D.; Eom, Y. Improving performance by bridging the semantic gap between multi-queue SSD and I/O virtualization framework. In Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST), Santa Clara, CA, USA, 30 May–5 June 2015.
28. Xia, L.; Lange, J.; Dinda, P.; Bae, C. Investigating virtual pass-through I/O on commodity devices. *ACM SIGOPS Oper. Syst. Rev.* **2009**, *43*, 3. [CrossRef]
29. Jones, M. Virtio: An I/O Virtualization Framework for Linux. Available online: https://developer.ibm.com/articles/l-virtio/ (accessed on 27 July 2021).
30. Bjørling, M.; Axboe, J.; Nellans, D.; Bonnet, P. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In Proceedings of the 6th International Systems and Storage Conference (SYSTOR), Haifa, Israel, 30 June–7 July 2013.
31. Kivity, A. Different I/O Access Methods for Linux, What We Chose for Scylla, and Why. Available online: https://www.scylladb.com/2017/10/05/io-access-methods-scylla/ (accessed on 27 July 2021).
32. NVMe Command Set Specifications. Available online: https://nvmexpress.org/developers/nvme-command-set-specifications/ (accessed on 27 July 2021).
33. Yang, J.; Minturn, D.; Hady, F. When Poll Is Better than Interrupt. In Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST), San Jose, CA, USA, 14–17 February 2012.
34. Yang, Z.; Harris, J.; Walker, B.; Verkamp, D.; Liu, C.; Chang, C.; Cao, G.; Stern, J.; Verma, V.; Paul, L. SPDK: A Development Kit to Build High Performance Storage Applications. In Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, China, 11–14 December 2017.
35. Kourtis, K.; Ioannou, N.; Koltsidas, I. Reaping the performance of fast NVM storage with uDepot. In Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST), Boston, MA, USA, 25–28 February 2019.
36. Lingenfelter, D. Measuring the Workload on Drives on a Linux System Using the Blktrace Tool. Available online: https://www.seagate.com/files/www-content/ti-dm/_shared/images/measure-workload-on-drives-tp662-1409us.pdf (accessed on 27 July 2021).