
# MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through

**Bo Peng,** ***Shanghai Jiao Tong University, Intel;*** **Haozhong Zhang,** ***Intel;***
**Jianguo Yao,** ***Shanghai Jiao Tong University;*** **Yaozu Dong,** ***Intel;***
**Yu Xu and Haibing Guan,** ***Shanghai Jiao Tong University***

https://www.usenix.org/conference/atc18/presentation/peng

**This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).**

**July 11–13, 2018 • Boston, MA, USA**

ISBN 978-1-939133-02-1

**Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.**

# MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through

Bo Peng
*Shanghai Jiao Tong University*
*Intel Corporation*

Haozhong Zhang
*Intel Corporation*

Jianguo Yao
*Shanghai Jiao Tong University*

Yaozu Dong
*Intel Corporation*

Yu Xu
*Shanghai Jiao Tong University*

Haibing Guan
*Shanghai Jiao Tong University*

## Abstract

The fast access to data and high parallel processing in high-performance computing instigates an urgent demand on the I/O improvement of the NVMe storage within datacenters. However, unsatisfactory performance of the former NVMe virtualization demonstrates that NVMe storage devices are often underutilized within cloud computing platforms. NVMe virtualization with high performance and device sharing has captured the attention of researchers. This paper introduces MDev-NVMe, a new virtualization implementation for NVMe storage device with: (1) full NVMe storage virtualization running native NVMe driver in guest, and (2) a mediated pass-through mechanism with an active polling mode which can achieve both high throughput, low latency performance and a good device scalability. This paper subsequently evaluates MDev-NVMe on Intel OPTANE and P3600 NVMe SSD by comparison with the mainstream virtualization mechanisms using application-level I/O benchmarks. With polling, MDev-NVMe can demonstrate a 142% improvement over native (interrupt-driven) throughput and over 2.5 $\times$ the *Virtio* throughput with only 70% native average latency and 31% *Virtio* average latency with a reliable scalability. Finally, the advantages of MDev-NVMe and the importance of polling are discussed, offering evidence that MDev-NVMe is a superior virtualization choice with high performance and promising levels of maintenance.

## 1 Introduction

NVM Express (NVMe) [21] is an inherently parallel, high-performing interface and command set designed for non-volatile memory based storage. NVMe devices have been designed from a logical device interface where storage media is attached via a PCIe bus [26]. NVMe SSDs can deliver I/O operations with a very high throughput performance on the low latency, making them the superior storage choices for data-intensive datacenters to obtain high performance computing in cloud services [2, 5]. Currently, NVMe SSDs are also used to accelerate I/O missions between system storage and other PCIe devices, such as GPUs on graphics cards [7]. However, because of imbalanced requirements of applications in high performance platforms, NVMe devices are often underutilized in terms of I/O performance [19, 22, 17]. NVMe devices in datacenters require solutions to achieve high performance and sharing capabilities [31].

I/O virtualization [13, 24] is a key component of cloud computing, which not only optimizes utilization of physical resources, but also simplifies storage management, providing a simple and consistent interface to complex functions. Typical cases of NVMe virtualization include VM boot disk and server side caching of VM data. With respect to NVMe devices, limited PCIe slots make NVMe virtualization essential in datacenters with high density so that the benefits of NVMe devices can be shared among the VMs.

Recently, NVMe virtualization [33, 28, 3] has become widely studied by computer scientists and researchers. Keeriyadath [16] summarizes three of the NVMe virtualization approaches; implementing SCSI to NVMe translation layer on the hypervisor (blind mode), pure virtual NVMe stack by distributing I/O queues amongst hosted VMs (Virtual Mode), and SR-IOV [11] based NVMe controllers per virtual functions (physical mode). Modern virtualization solutions for NVMe Storage such as *Virtio* [23], Userspace NVMe driver in QEMU [34] and Storage Performance Development Kit (SPDK) [15] are implemented in the userspace of the Linux system. However, these mechanisms have their respective shortcomings: the performance of VMs with *Virtio* and Userspace NVMe driver in QEMU are considered poor compared with native drivers. At the same time SPDK must collaborate with "Hugepage" memory which adds further pressure to the memory resources of the host server.

Observing that mediated pass-through [30] has been gradually recognized as an effective solution for I/O virtualization [12, 27], concurrently the Linux kernel has

introduced a mediated device framework which supports such usage since 4.10 [25]. Therefore, it is proposed that implementing an effective NVMe virtualization using mediated pass-through is an appropriate methodology. The detail of MDev-NVMe design is that a full virtualization is proposed which enables VMs running with a native driver, and a MDev-NVMe module with device emulation, admin queue emulation, and also with I/O queue shadow. To improve the VM performance, an active polling mode for queue handling is proposed. As a result, this design is then built for the experiments on Fio [4] benchmarks by undertaking comparison experiments between mainstream NVMe virtualization solutions. The proposed design can achieve up to 142% improvement of througerhput with 70% average latency over native performance, and also performs well regarding scalability and flexibility. Also, the experiments are performed on different I/O blocksizes and provide suggestions on selecting appropriate blocksizes for NVMe virtualization. At last, the discussion of optimization on active polling is undertaken in the final section, which also provides suggestions on the future design of virtualization and high-performance storage.

In summary, this paper makes the following contributions:

(1) We introduce a full NVMe virtualization solution MDev-NVMe with mediated pass-through that runs the native NVMe driver in each guest.

(2) We demonstrate details of emulation of Admin queues and shadow of I/O queues in the mediated pass-through which can pass-through performance-critical resources accesses in NVMe storage.

(3) We design an active polling for shadow SQ and CQs and host CQs in NVMe to achieve better performance over native devices.

(4) We do further comparison evaluations of overall performance on throughput, average latency, QoS between MDev-NVMe and other virtualization mechanisms. We analyse the influence of blocksizes on performance and give suggestions for NVMe virtualization to choose blocksizes in different I/O scenarios.

(5) We discuss both MDev-NVMe's performance and maintenance advantages over former virtualization And we discuss the active polling optimization in virtualization and give suggestions on polling support in the design of virtualization and storage hardware.

The rest of this paper is organized as follows. A background of NVMe storage protocol and the motivation on NVMe virtualization are provided §2. The design is demonstrated and implementation details of MDev-NVMe are detailed in §3. The results of the comparison evaluations between MDev-NVMe and other mechanisms are located within §4. There is a discussion on polling in §5. §6 details related developments, and finally the conclusion is located in §7.

# 2 Background and Motivation

## 2.1 NVMe Storage Protocol

For a long time, SATA (Serial ATA) has been used as the interface for traditional storage devices such as Hard Disk Drives (HDD) and a number of Solid State Drives(SSD). Despite the fact that SATA is enough for a rotational storage, the SATA interface cannot meet the requirement of modern storage devices. This is because of the requirement to provide a much higher I/O throughput due to the limitation of Advance Host Controller Interface (AHCI) architecture design in SATA interface. To resolve the I/O performance bottleneck, the NVMe protocol was designed and developed with a PCIe interface instead of SATA [21]. Generally, to accelerate the I/O speed of SSD, NVMe protocol optimized command issues between the host system and the storage devices, compared with the traditional ATA AHCI protocol. Concurrently, it includes support for parallel operations by supporting up to 64K commands within a single I/O queue to the device [21].

Here are some basic definitions in NVMe protocols. NVMe defines two main types of commands: **Admin Commands** and **I/O Commands**. In I/O operations, commands are placed by the host software into the **Submission Queue (SQ)**, and completion information received from SSD hardware is then placed into an associated **Completion Queue (CQ)** by the controller. NVMe separately designs SQ and CQ pairs for any Admin and I/O commands respectively. The host system maintains only one Admin SQ and its associated Admin CQ for the purpose of storage management and command control, while the host can maintain a maximum of 64K I/O SQs or CQs. The depth of the Admin SQ or CQ is 4K, where the Admin Queue can store at most 4096 entries, while the depth of I/O Queues is 64K. SQ and CQ should work in pairs, and normally one SQ utilizes on one CQ or multiple SQs utilize the same CQ to meet the requirements of high performances in multithread I/O processing. A SQ or CQ is a ring buffer and it is a memory area which is shared with the device that can be accessed by Direct Memory Access (DMA). Moreover, a doorbell is a register of the NVMe device controller to record the head or tail pointer of the ring buffer (SQ or CQ).

A specific command in a NVMe IO request contains concrete read/write messages and an address pointing to the DMA buffer if the IO request is a DMA operation. Once the request is stored in a SQ, the host writes the doorbell and kicks (transfers) the request into the NVMe device so that the device can fetch I/O operations. After an IO request has been completed, the device will subsequently write the success or failure status of the

request into a CQ and the device then generates an interrupt request into the host. After the host receives the interrupt and processes the completion entries, it writes to the doorbell to release the completion entries.

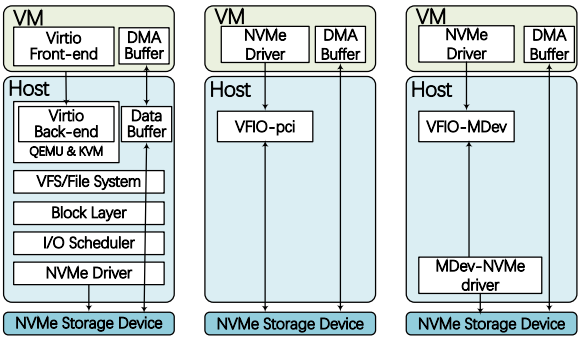
## 2.2 Motivation

High-performance cloud computing applications have raised great demands on the I/O performance of modern datacenters. The I/O virtualization is widely deployed in datacenter servers to support the heavy data transmission and storage tasks within cloud computing workloads. The virtualization for the new high-performance storage devices NVMe is a concentration point of I/O virtualization in cloud computing. This is because the NVMe devices can demonstrate great performance advantages over the traditional devices on the native host servers. There have been a number of previous successful NVMe virtualization solutions including VM boot disk and server side caching of VM data, both of which prove that NVMe is well suited for exploitation in various virtualization. To ensure all the VMs in cloud computing servers share the benefits of NVMe devices from the basic I/O virtualization has become essential in virtualization research for NVMe. The NVMe devices in VMs provides some basic requirements for the virtualization mechanisms:

**High Throughput:** The throughput performance of NVMe storage is the most important performance feature due to the requirements for both quick data access and parallel processing of cloud computing services. The virtualization mechanism needs to ensure the high performance of the NVMe devices in VMs, so the throughput-intensive services can work increasingly efficiently.

**Low Latency:** From previous research, the unloaded read latency of NVMe SSD storage is 20-100$\mu$s [9]. Storage virtualization mechanisms usually cause latency overhead in VMs because the high frequency of I/O operations will bring large numbers of context switches. To satisfy latency-sensitive applications in cloud computing services, NVMe virtualization should suffer a low latency overhead.

**Device Sharing:** Modern high-performance datecenter servers often support large numbers of VMs for separate cloud computing services. The limited NVMe storage devices should be shared by different VMs which can show great scalability. With device sharing, different VMs can work with high-performing NVMe without interference between each other, and the statuses of different VMs need to be managed by the hypervisor. Also, supporting VM live migration [32] is also an important requirement on the virtualization mechanism, which greatly relies on device sharing features. Also, the tail latency in VMs shows the QoS of device sharing among different VMs.




(a) Virtio (b) VFIO (c) MDev-NVMe

Figure 1: NVMe virtualization comparisons.

The comparisons between *Virtio* and *VFIO* and our design are displayed in Fig. 1. The previous NVMe storage virtualization *Virtio* and *Virtual Function I/O (VFIO)* [29] both have their shortcomings: *Virtio* uses the original *virtio-blk* [20] driver and it has no specific optimization for NVMe. Consequently, *Virtio* suffers readily apparent overhead from software layers which originate from guest OS to the NVMe devices, including the Virtual File System (VFS), block layer and I/O scheduler. Whereas in *Virtio*, the throughput and latency in VMs can only meet 50% of the native performance.

On the other hand, *VFIO* method can use direct pass-through [1, 10] to assign the entire NVMe device to a single VM, therefore *VFIO* can not meet the requirements of device sharing despite its near-native performance on both latency and throughput. Also, with direct pass-through, the information of virtual NVMe can not be managed by the hypervisor so it cannot support migration.

As shown in Fig. 1(c), the mediated pass-through virtualization is a full virtualization mechanism which combines both the advantages of the two former mechanisms, and it requires no modification to guest OS kernel and these guests can use the raw NVMe drivers. A "VFIO-MDev" interface is implemented for each individual VM to cooperate with QEMU [6] to support device sharing between multi VMs, which "VFIO-MDev" can emulate all the PCIe device features for each guest NVMe devices. In order to guarantee a near-native I/O performance in mediated pass-through, the direct concept of the proposed MDev-NVMe is to optimize *VFIO* by modifying the raw NVMe driver into the MDev-NVMe driver in Linux kernel, so that it can pass-through basic units of I/O operations as many as possible. As a result, all the statuses of physical and virtual queues can be managed by the MDev-NVMe module, and any file system on the host kernel is unnecessary, which helps reduce the software stack overhead. In general, a mediate pass-through is a superior choice which can meet all the essential goals of NVMe virtualization.

## 3 Design and Implementation

Cloud applications demand on high performance NVMe storage devices in modern datacenters. To meet both low latency and high throughput performance goals and support a sharing policy on NVMe devices, we design a mediated pass through virtualization mechanism: MDev-NVMe.

### 3.1 Architecture

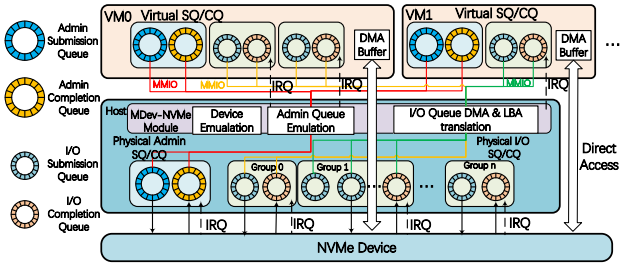



Figure 2: Architecture of MDev-NVMe.

The mediated pass-through virtualization is a full virtualization mechanism which combines both advantages of the two former mechanisms *Virtio* and *VFIO*. At the same time, it requires no modification to the guest OS kernel so the guest kernel can use the raw NVMe drivers. From the aspect of the architecture, *Virtio* provides a full device emulation with the *virtio-blk* front end driver which suffers an apparent performance overhead from software layers. And *VFIO* passes through the entire device to one VM with no sharing features. Therefore, the mediated pass-through selectively emulates or pass-throughs basic I/O units of the NVMe I/O operations. To accelerate the I/O operations and reduce the unnecessary frequent "vm-exit", we pass through I/O SQs and CQs with a queue shadow instead of I/O queue emulation. The access of DMA buffers in NVMe devices can be dealt similar to other PCIe devices.

The brief introduction of the architecture is shown in Fig. 2. From the architecture, we offer a full virtualization method, so all guests require no modification to raw NVMe drivers and can utilize all the basic NVMe commands. The MDev-NVMe is a kernel module within the Linux host system which offers three important functions: the basic PCI device emulation, Admin queue emulation and I/O queue shadow with DMA & LBA translation. Also, in MDev-NVMe, the DMA buffer in NVMe devices can be accessed by VMs as normal PCIe devices, which is similar to direct pass-through. All the guests cooperate with these three important functions.

**Device Emulation:** MDev-NVMe allows that all the guests need no modification to the native NVMe driver in kernels, so each guest can access a virtual NVMe device with the PCIe bus. So the MDev-NVMe module should emulate all the features of the NVMe devices for each VMs. The details of PCIe device emulation based on"VFIO-MDev" which is similar with the original "VFIO-pci": we emulated PCI registers such as BAR and other PCI configurations, and emulated NVMe registers and logic of guest device. The emulated interrupts contain INTx, MSI, and MSI-X.

**Admin Queue Emulation:** In NVMe protocol, there is only one pair of Admin SQ and CQ, while there can be up to 65535 pairs of I/O SQs and CQs. When different VMs generates I/O SQ and CQ at the same time, the only pair of Admin SQ and CQ in the host kernel should be able to handle the sharing and scheduling between the VMs. So we need device emulation for the Admin Queue so that all the guests can manage their NVMe I/O operations in the emulated pair of Admin Queue, and MDev-NVMe will finally parse the virtual Admin commands into physical Admin commands.

**I/O Queue Shadow:** Since there can be up to 64K I/O queues in a NVMe device instead of only one pair, I/O queues do not require device emulations. In our architecture, all the guest I/O commands in virtual queues can be passed through into the shadowed physical queues with the DMA & LBA translation. The guest I/O commands are directly executed by the device as the same as host commands after command translations, which can apparently reduce the overhead through emulation.

**DMA Buffer Access:** Similar to other PCIe devices, the host DMA engine can directly manage the memory addresses of the DMA buffer in guest NVMe storage devices, just like the *VFIO* pass-through. The DMA feature is necessary at any time that the CPU cannot maintain the rate of data transfer required, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer. The frequent I/O operations on NVMe storage devices cause a high rate of data transfer, and MDev-NVMe can overcome the CPU overhead of frequent "vm-exit" when accessing device memory.

### 3.2 Queue Handling

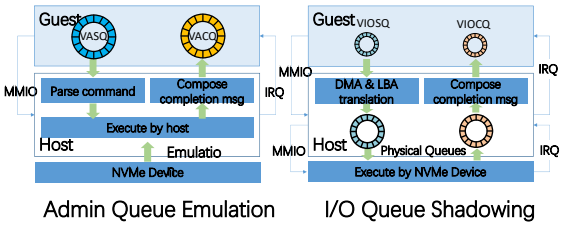



Figure 3: A brief introduction to queue handling.

We separately design queue handling mechanism for Admin Queues and I/O Queues, as shown in Fig. 3.

When more than one virtual machine runs on the host, they all maintain one pair of virtual Admin SQ and CQ

in their guest OS as native. Therefore, the only physical pair of Admin SQ and CQ must be shared among the virtual machines. Concretely, MDev-NVMe usually examines the virtual Admin Queue to check the commands. If the guest Admin commands wants to create guest I/O queues, the MDev-NVMe module can directly create corresponding physical I/O queues in the host kernel. If guest Admin commands only want to check and manage the information of the I/O status and device information without attaining the concrete features and parameters of the NVMe devices. The MDev-NVMe module can directly write or update information from the NVMe doorbell register with Memory-Mapped I/O (MMIO) operation. The majority of the Admin queue operations are accomplished in the initialization process of virtual machines, so the Admin queue emulation will not become the bottleneck location of the NVMe I/O performance.

Compared with Admin queues, we use a more convenient pass-through mechanism for guest I/O SQs and CQs, where we make a queue shadow from the guest queues to the host queues. NVMe devices have more than one pair I/O SQs and CQs, so I/O queue shadow is feasible in MDev-NVMe virtualization. In host kernel, I/O queues are separately bound with physical cores of the server and interrupts of the submission and completion of each I/O commands from queues are trapped by the corresponding cores to accelerate I/O operation. To improve performance in MDev-NVMe VMs, we can also bind the guest I/O queues with the host CPU resources by shadowing guest I/O queues and host I/O queues by providing a simple DMA & LBA translation, which is introduced in the §3.3. Once the guest writes or updates an I/O command, the MDev-NVMe module will directly writes translated commands into shadowed physical queues and updates the device doorbell register with MMIO. When a guest I/O operation completes in the corresponding physical queue, the device will generate an interrupt into the host kernel, and also our MDev-NVMe module generates an interrupt into the guest kernel after checking the DMA & LBA translator. With queue shadowing, the handling of interrupts of the guest virtual queues is actually the handling of interrupts of shadowed physical queues, where we make full use of the bound CPUs (by physical queues) to accelerate the interrupt handling.

## 3.3 DMA & LBA translation

As we demonstrated in the architecture figure in §3.1, a guest I/O queue are shadowed with a corresponding physical I/O queue in a DMA & LBA translator to achieve resource partitioning and better I/O performance. After translation, a virtual I/O queue id are bound with a physical I/O queue id, and the translation result will be stored in a translation table which is maintained in the host kernel. The translation process is based on the Intel Extension Page Table (EPT) support for memory translation, so that we can maintain a translation table cache when multiple VMs create large numbers of virtual I/O queues and the limited physical queues must be shared.

The DMA & LBA translation are based on the static partition strategy, where all VMs are assigned with part of the continuous space of the NVMe device at the initialization process. Specifically, the translation unit in an I/O command is the data pointer which points to a DMA buffer. The address is a Guest Physical Address (GPA). The MDev-NVMe module use *vfio_pin_pages* to translate the GPA to HPA (Host Physical Address) and pin the DMA buffer in host memory. Specifically, the *vfio_pin_pages* can ensure the isolation of the guest memory between different virtual machines. Since all the translations are under control of the translator in MDev-NVMe module, a malicious VM cannot access the physical I/O queues which is not assigned to itself and can not access the I/O queues in other VMs either. The command buffer is also protected by the EPT, which helps to ensure the security of DMA buffers of different VMs. Another important unit of the I/O operation is the Start Logical Block Address (SLBA). Since our MDev-NVMe virtualization takes a static partition mechanism, the SLBA in guest I/O command can be modified and subsequently copied into the host I/O queue by applying a space area offset, which are also managed by MDev-NVMe module in host kernel.

## 3.4 2-Way Polling Mode

With the mediated pass-through mechanism, the NVMe devices can achieve a performance better than *Virtio*. However, the performance in VMs still show an obvious gap compared with the native platform in experiment observation. The I/O bottleneck results from frequent submissions and completions of guest I/O commands, so we discuss the detailed origin of overhead in Fig. 4.

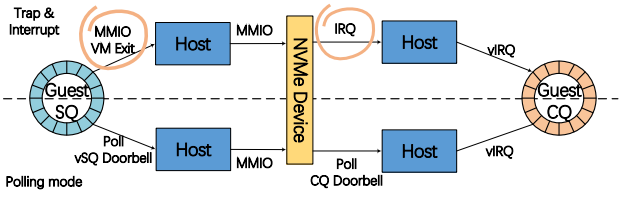



Figure 4: A guest I/O operation process in two modes.

Fig. 4 shows entire processes of a guest NVMe I/O command operation in "non-polling" or "polling" mode. Generally, the submission latency from the host kernel to the physical NVMe device is inappreciable, which is less than 10 $\mu$s in our observations from experiments. However, the guest I/O commands suffers seriously from two part overhead: (1) when the guest I/O commands are translated and submitted to the corresponding phys-

ical queues through an MMIO operation it will result in a "vm-exit", (2) when handling interrupts generated by physical device when updating a doorbell register with a completion of physical I/O queue.

To overcome the two-part overhead, a direct idea is to change the trap of interrupts into an active polling mechanism in the 2-way overhead. First, the shadowed doorbell of guest SQ or CQ can be stored in host memory area so that MDev-NVMe can directly manage all the guest I/O operations and update the NVMe queue status. With polling, we polls guests to write into the shadow doorbells instead of generating an MMIO into the host. Also, we disables interrupt integration of host CQs into guests and adopts polling new completion entries from the host CQs. As a result, the shadow SQ tail doorbell, the shadow CQ head doorbell, and the host CQ are all stored in the host memory, and they can be directly fetched and immediately updated by the polling threads.

Now in our MDev-NVMe, we use 3 threads for individually polling the shadow SQ tail doorbells, the shadow CQ head doorbells, and the host CQs so that we can get even better performance of I/O queue handling than native platform in the host kernel, which is shown in §4, while using 1 thread with round-robin polling of multiple queues cannot provide corresponding performance. The threads will bring 100% usage of 3 cores on the host server. And the polling threads can be shared between VMs.

# 4 Experiments

To demonstrate the I/O performance of virtual machines of our mediated pass-through virtualization mechanism, we run fio [4] I/O benchmark experiments in Linux virtual machines based on the comparison between different virtualization mechanisms, including *Virtio*, famz userspace driver, SPDK *vhost-scsi*, and SPDK *vhost-blk*. The performance of MDev-NVMe and the other 4 mechanisms are compared and normalized with the native performance on the physical devices. For Throughput performance, MDev-NVMe shows the best bandwidth performance among all the virtualization mechanisms and can provide up to 1.5× native performance. For Latency performance, MDev-NVMe presents the lowest average latency, and can give a bounded maximum latency with a reliable QoS performance. Also, the tail latency performance of MDev-NVMe is also outstanding among all the comparison experiments. In the meanwhile, our MDev-NVMe scales well without a visible performance drop.

## 4.1 Configuration

The evaluation concentrates on achieving the I/O performance of NVMe Storage devices and demonstrating the advantages and disadvantages of different virtualization methods, including *Virtio*, Famz userspace driver, SPDK *vhost-scsi*, and SPDK *vhost-blk*. Experiments on different concrete NVMe SSD products show different results and give us more insights. Nowadays, the most advanced NVMe Storage Device is based on Intel OPTANE Technology with 3D XPoint Memory media. The OPTANE SSD has an amazing performance which can support up to 550K IOPS in 4K random read and 500k IOPS in 4K random write with a 10$\mu$s latency. So we first build our experiment environment on the OPTANE SSD DC P4800X 375G NVMe SSD, and the server hardware platform includes 2 Intel Xeon CPU E5-2699 v3 with 18 CPU cores (2.3GHz), 64GB system memory. Besides, we also do the same experiments on a more commonly used NVMe SSD: INTEL SSD DC P3600 400G, running on the server which includes 2 Intel Xeon CPU E5-2680 v4 with 14 CPU cores (28 threads) (2.4GHz), 64GB system memory.

We run 64bit Ubuntu 16.04 with a 4.10.0 kernel with our MDev-NVMe kernel module in the host server, and 64bit Debian 9.0 with a 4.10.0 kernel in the guests. Each virtual machine is allocated with 4 VCPUs and 8GB system memory. To get the best I/O performance in comparison experiments, we set up 1-VM case which the single virtual machine can get the entire volume. To discuss the scalability of MDev-NVMe sharing, we partition the NVMe SSD with 4 60G area into 4 individual VMs.

We use the flexible I/O tester (FIO) [4] as our application evaluation benchmark. FIO is a typical I/O benchmark with different parameters to demonstrate the IO performance, and it is widely used in research and industries. Specifically, we take **libaio** as the fio engine and we run 5 groups of test scripts in Table 1, including the random read or write with "iodepth=1", the random read or write with "iodepth=32", and the 70% random read and 30% random write. The FIO test doesn't use any file system and chooses "O_DIRECT" parameter for Direct I/O experiments.

Table 1: Fio test cases

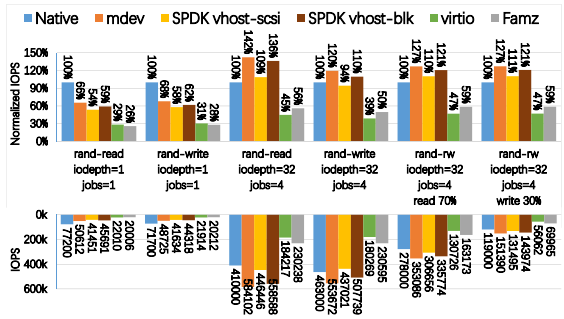
| Test case | Description |
|---|---|
| rand-read-qd1 | 4K random read, iodepth=1, numjobs=1 |
| rand-write-qd1 | 4K random write, iodepth=1, numjobs=1 |
| rand-read-qd32 | 4K random write, iodepth=32, numjobs=4 |
| rand-write-qd32 | 4K random write, iodepth=32, numjobs=4 |
| rand-rw-qd32 read 70% | 4K random read, iodepth=32, numjobs=4 |
| rand-rw-qd32 write 30% | 4K random write, iodepth=32, numjobs=4 |

## 4.2 Throughput Performance

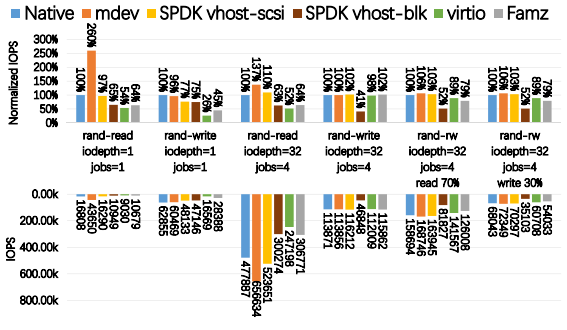
We first concentrate on the throughput performance of I/O operations on OPTANE and P3600. The basic throughput benchmark concentrates on the 4K page random read and write. Fig. 5(a) and Fig. 5(b) are the corresponding IOPS results on the OPTANE and P3600 NVMe SSDs. The results in these figures contain two

parts: the upper part is the normalized performance results based on the native performance baseline, and the bottom part is the original benchmark IOPS results. On OPTANE, the MDev-NVMe mechanism shows the best throughput performance with or without I/O multi-queue optimization("iodepth=1" or "iodepth=32") over all the other virtualization mechanisms. With the active polling for I/O queues, MDev-NVMe can make full use of multi-queue features in 4K random read and write I/O benchmarks with "iodepth=32", where MDev-NVMe shows up to 142% performance over native results. Since the SPDK *vhost-scsi* and SPDK *vhost-blk* mechanisms utilize similar idea of polling, they can also achieve better performance than native results. On P3600 NVMe SSDs, MDev-NVMe show the best performance although the advantages are not as obvious as the OPTANE experiments. Moreover, the throughput performance of 4K random write with multi-queue optimization presents an apparent gap compared with the 4K random read with multi-queue optimization, as we shown in the bottom part of Fig. 5(b).

In this group of experiments, different virtualization mechanisms show well-matched performance expect SPDK *vhost-blk*, which shows that SPDK *vhost-blk* optimization does not works efficiently on P3600.




(a) OPTANE




(b) P3600

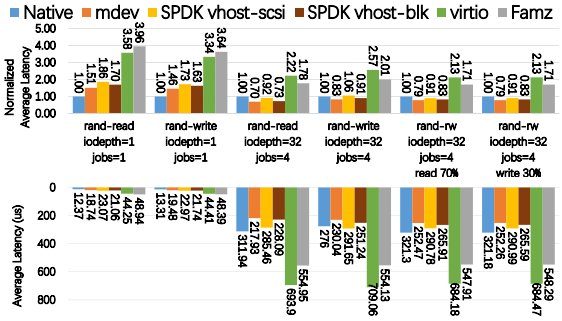
Figure 5: Throughput performance of IOPS.

## 4.3 Latency Performance

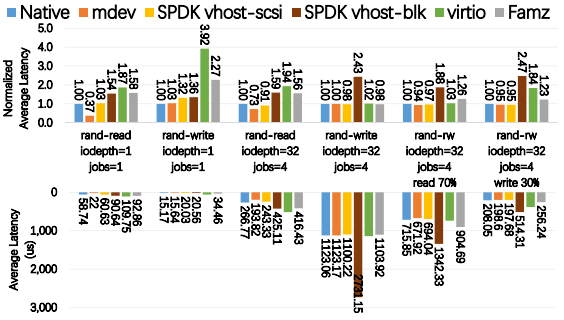
We also focus on the latency of I/O operations in virtual machines based on different NVMe virtualization mechanisms. We particularly discussed the average latency, max latency and the tail latency on different virtualization mechanisms in our latency experiments. Specifically, the latency measured in our fio experiments contains both submission latency and completion latency.

### 4.3.1 Average Latency

Firstly, the average of latency performance in benchmarks are demonstrated in Fig. 6(a) and Fig. 6(b). Also the upper part is the normalized performance results based on the native performance baseline, and the bottom part is the original latency results. The average latency performance can show the average operation overhead of each I/O commands in NVMe devices through the software stack of virtualization.




(a) OPTANE




(b) P3600

Figure 6: Average latency performance.

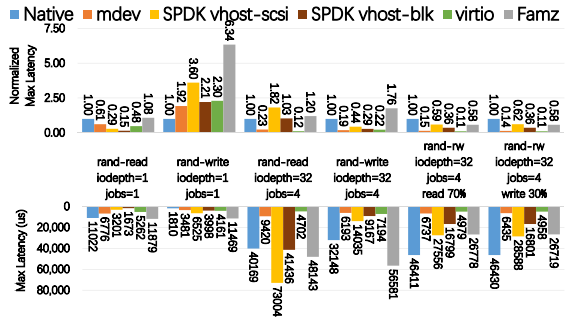
As we talked in the former section, the mediated pass-through takes advantage of the *VFIO* pass-through mechanisms and it overcomes the block layer and Backend software stack. When there is no multi-queue optimization [8, 18], all the virtualization mechanism shows no apparent advantage over the native latency performance. The MDev-NVMe, SPDK *vhost-scsi* and SPDK *vhost-blk* can show 2× latency performance over the *Virtio* and Famz userspace driver, and the original latency intervals between native performance and MDev-NVMe is nearly 10 $\mu$s, which is at the level of the latency from the host to physical NVMe devices. In multi-queue experiments where "iodepth=32", the native latency increase to 322 $\mu$s which is about 30× the result with "iodepth=1". However, in multi-queue benchmarks, Mdev-NVMe show the best performance and can

achieve only 81% of the native performance. MDev-NVMe shows over 2.5× advantages over *Virtio* and Famz drivers, and the advantages over SPDK *vhost-scsi* and SPDK *vhost-blk* are not far but obvious. On P3600, MDev-NVMe can show great results on all the test cases, except the random write with "iodepth=32" benchmark where all the virtualization performance results are similar. Similar to the throughput benchmarks, SPDK *vhost-blk* does not show the optimization results as those on OPTANE.

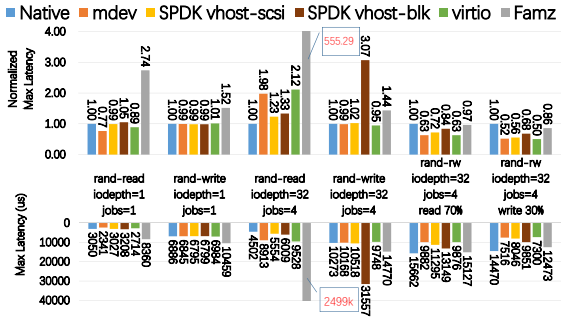
In general, MDev-NVMe can achieve the best average latency performance, so our mediated pass-through mechanism has overcome utmost software layer I/O overhead.

### 4.3.2 QoS

To talk about the QoS performance of the NVMe devices in virtual machines, we concentrate on the maximum latency and tail latency performance in different benchmark test cases.



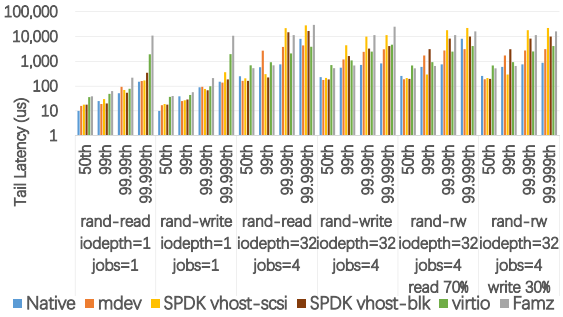

(a) OPTANE



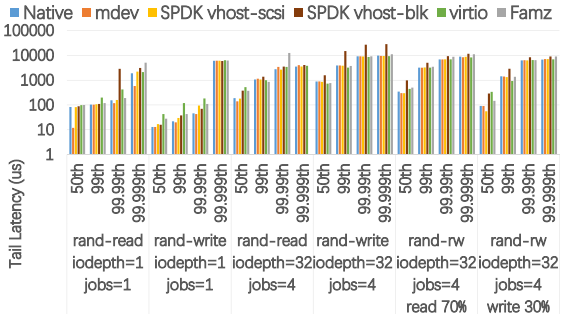

(b) P3600

Figure 7: Maximum Latency performance.

Firstly, the max of latency performance in benchmarks is demonstrated in Fig. 7(a) and Fig. 7(b). For Qos performance, our MDev-NVMe shows well-matched performance of the maximum latency with *Virtio* on both OPTANE and P3600. Also, the normalized maximum latency performance of MDev-NVMe can be less than 1, showing the latency performance can be bound in a very low level. In this part of experiments, SPDK *vhost-scsi* and SPDK *vhost-blk* can not offer a well-matched performance with *Virtio* and MDev-NVMe on both Optane and P3600. Specially, Famz userspace NVMe driver cannot bound its maximum latency in an acceptable level because in Fig. 7(b), the maximum latency of random read with "iodepth=32" is 2499k $\mu$s. Also, SPDK *vhost-blk* present a straggler QoS performance on P3600 when doing the random write with "iodepth=32" benchmarks.




(a) OPTANE




(b) P3600

Figure 8: Tail Latency performance.

And we also present the tail latency performance in Fig. 8(a) and Fig. 8(b) with a logarithmic scale coordinate of the latency results. From the figures, MDev-NVMe can bind the 50th and 99th latency performance in similar level. We also compare 99th, 99.99th and 99.999th tail latency in different virtualization mechanisms. MDev-NVMe has the smallest intervals between 99th, 99.99th and 99.999th latency compared with other virtualization mechanisms. In general, MDev-NVMe can provide a promising QoS performance of NVMe virtualization.

## 4.4 Scalability

Our mediated pass-through mechanism can provide device sharing over a NVMe SSD on the initialization process of virtual machines with a static partition of the device. We design an experiment by separately running 1 VM, 2VMs, and 4VMs and evaluating the virtual NVMe I/O performance to discuss the scalability. In our experiment, each VM manages a 60G continuous storage area

by the unmodified guest driver. Fig. 9 presents the relative scalability performance results of MDev-NVMe on the P3600 NVMe device with the scripts in Table 1, and each result is the sum of all throughput in all VMs. As we talked about in the former section, the random write performance is not well-matched with the random read performance. When increasing VMs in the benchmark with "iodepth=1", the performance of 4K random read can increase by the numbers of VMs because the NVMe read performances are not fully exerted when there is no multi-queue optimization [8, 18]. In the other test cases, since the I/O queue resource has been fully used, the I/O performance stays in a stable status when VMs increase from 1 to 4. Since the basic units of NVMe I/O operation are the SQs and CQs, and the performance in VMs is strictly connected with the number of queues assigned for them. As a result, when the number of VMs increases, the assignment of limited I/O queues becomes the bottleneck of scalability. When the queue depth is small, setting up more VMs can utilize make better use of queue resource. When the queue depth is large, MDev-NVME can guarantee that multi VMs can work together without performance obvious drop. So we can conclude that MDev-NVME can ensure a promising scalability.

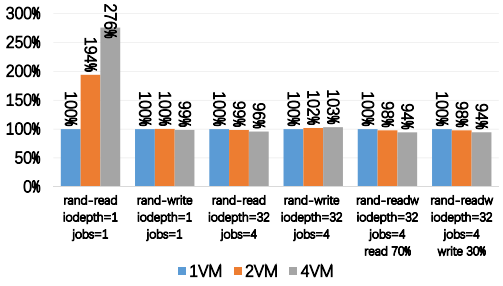



Figure 9: IOPS performance of multi VMs on P3600.

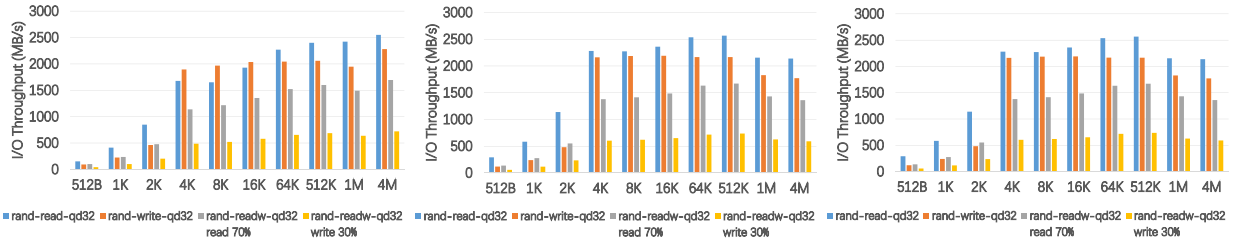
## 4.5 Influence of I/O Blocksize

The former experiments are based on the 4K random read or write benchmarks, which are concentrated points of test cases on different storage devices. Moreover, the I/O blocksize can explicitly influence the performance of devices wherever on the native devices or in virtual machines. As a result, we want to take the blocksize parameter of I/O performance into consideration to give a law of performance varying with different blocksizes, and we can also give more suggestions for different NVMe virtualization mechanisms to make a right choice of blocksizes in different I/O scenarios.

We run the fio test cases with multi-thread optimization ("iodepth=32") which are the last four test cases presented in Table 1. We chose 10 groups of blocksize parameters in each group of random read of write experiments on OPTANE (512B, 1K, 2K, 4K, 8K, 16K, 64K, 512K, 1M, 4M). The results of the influence of blocksizes on bandwidths are presented in Fig. 10(a), 10(b) ,and 10(c). We only chose the native, MDev-NVMe, and SPDK *vhost-blk* experiments as a comparison since MDev-NVMe and SPDK *vhost-blk* show the best 4K I/O performance based on our prior experiments on OPTANE. In Fig. 10, the variation trends of the influence of blocksizes on bandwidths in the three sub figures basically agree with each other. When the blocksizes are smaller than 4K, the throughput performance improve with the blocksize parameter and the high throughput performance are not fully exerted. These is because the NVMe I/O queues can be 64K large at most and when the blocksize is too small and the device is not fully occupied, resulting in a I/O queue resource waste. Also the throughput performance can be continuously improved with the growing blocksize on native devices, but the performance results of MDev-NVMe and SPDK shows apparent declines when the blocksize is larger than 512K. The reason for this performance drop is that when the I/O blocksize is too large, then transmission of an I/O block may be automatically separated by the hypervisor and these separations bring an appreciable overhead. A single I/O command of such a large block needs cooperation of several I/O queues and the scheduling of queues will become a performance bottleneck.

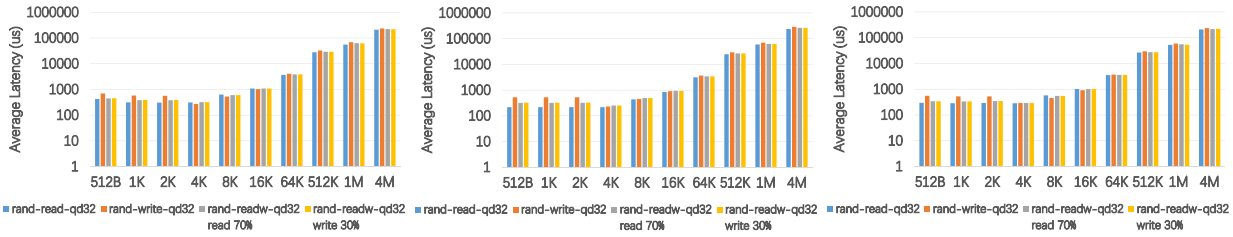
Also, we demonstrate the average latency performance of native, MDev-NVMe, and SPDK *vhost-blk* experiments in Fig. 11(a), 11(b), and 11(c). We use the a logarithmic scale coordinate to present the average latency experiment results in the three sub figures from Fig. 11. The variation trends of the influence of blocksizes on average latency performance in the three sub figures agree with each other as well as the throughput performance results. The average latency slightly decreases from 512B to 4K and then progressively increases from 4K to 4M. The MDev-NVME and SPDK can all bind their average latency performance in the same order of magnitude as the native performance. When the blocksize $\leq$ 16K, the average latency performance is in the level of several hundred $\mu$s. When the blocksize is chosen as 64K, 512K, 1M and 4M, the average latency performance increases exponentially by the magnitude. The main reason for this unacceptable latency are mainly resulted by the I/O block separations and the overhead waiting for all queue competitions.

In general, the throughput performance is good when blocksizes is bigger than 4K while the average latency performance will become unacceptable when the blocksize is bigger than 64K. Our MDev-NVMe can provide a stable and excellent performance of both throughput and latency when choosing blocksizes as 4K, 8K, and 16K, and the performance is better than SPDK and native performance. To support more optional blocksize with the high performance, we would give a more efficient I/O queue scheduling in our future works.




(a) Native performance (b) MDev-NVMe performance (c) SPDK performance

Figure 10: The throughput of I/O bandwidth on different blocksizes on OPTANE NVMe device.




(a) Native performance (b) MDev-NVMe performance (c) SPDK performance

Figure 11: The average latency of I/O operation on different blocksizes on OPTANE NVMe device.

## 5 Discussions

In this section we discuss the advantages of MDev-NVMe over other virualization mechanisms, the overhead issues, and the importance of active polling in virtualization.

**Advantages of MDev-NVMe:** As we demonstrated in §4, MDev-NVMe presents outstanding performance on throughput, average latency, and QoS. Furthermore, the data from the experiments suggests the performance in VMs can be even better than a native device. In our comparison experiment, MDev-NVMe can achieve over 2.5× throughput performance of Virtio and Famz, and achieve less than 31% average latency of *Virtio* and Famz. SPDK *vhost-scsi* and *vhost-blk* can provide a promising performance, slightly inferior to MDev-NVMe. However, SPDK needs Hugepage memory and additionally the setting up of a SPDK *vhost* device on the NVMe device, so it may restrict the flexibility of physical NVMe devices. Our MDev-NVMe is implemented as a kernel module, offering more convenience for cloud administrators and users with its better maintenance than SPDK, and MDev-NVMe brings no memory overhead.

**Overhead issues:** The shadow of guest I/O queues to physical queues is determined by the MDev-NVMe module. The scheduling of I/O queues is based on a simple FIFO policy. When aggressive I/O tasks in VMs are competing for the limited physical queues it can increase the scheduling overhead in the host and will lead to some problems in the balancing of requests. Here, a increasingly complicated scheduling algorithm is implemented or a queue resource ballooning methodology could be used in future work to overcome the potential overhead in heavy I/O workloads. To overcome the I/O performance bottleneck results from the high frequency vm-exit of guest I/O commands, an additional CPU resource is used for polling in order to accelerate the interrupt operations. A polling thread aggressively utilizes the CPU resource of a single core, which transfers overhead to the host server when running large numbers of VMs. However, the proposed system ensures no CPU resource overhead to VMs since all the guests use native NVMe drivers which are not aware of the polling threads. Moreover, polling can significantly reduce CPU usage in the VM because it avoids "vm-exit".

**Importance of polling:** The main agreement of MDev-NVMe and SPDK is that the virtualization mechanisms take active polling instead of interrupts handling. The traditional virtualization mechanisms are often based on the trap and emulations. The trap of the "vm-exit" needs additional context switches between VMs and the host kernel. The NVMe protocol is an inherently parallel and high-performing interface and command set. Therefore, when we would like to make full use of the I/O queue resource, the large numbers of interrupts bring an appreciable overhead for guests. Now, assigning exclusive CPU computing resources to handle the high frequency of interrupts can directly help to achieve high performance of I/O operations in VMs. Taking polling is the most direct and obvious idea to assign the exclusive CPU resource for queue handling.

Despite the high utilization of polling CPU may generate limited CPU computing resource waste to the host servers, polling is still necessary when the I/O workloads are aggressive in achieving extremely high performance.

For example, when shopping seasons such as "Black Friday" arrive, the servers of e-commerce companies will face great pressure of database accesses and parallel processing. Subsequently, the storage device throughput and latency performance is directly connected with the respective companies profits, losses and their customers satisfaction. Supporting polling in the virtualization mechanism is essential to take full advantage of the benefits of NVMe devices to meet the requirements on I/O performance in datacenters. Moreover, NVMe virtualization should support adaptive polling, which allows such datacenter administrators to decide when to choose a mild policy for I/O acceleration in VMs, to reduce expenses and support an increasing number of VMs. Therefore the provision of an adaptive polling mode and increasingly optimized polling algorithms is part of our focus for future work. It is also expected that the high performance I/O device hardware will be designed with a number of components to actively support or cooperate with the polling algorithm in the near future.

## 6 Related Work

Some research concentrates on NVMe virtualization, including the para-virtualization abstraction *Virtio*, a userspace NVMe driver in Qemu, and the Storage Performance Development Kit (SPDK).

**VMware virtual NVMe device:** After vSphere 6.5, VMware adds an NVMe controller for NVMe devices in its virtualization solution. The biggest benefit of using an NVMe interface over the traditional SCSI is that it can significantly reduce the amount of overhead, as well as reducing the IO latency for VM workloads.

**Virtio for NVMe:** *Virtio* is an abstraction for a set of common emulated devices in a para-virtualized hypervisor, which allows the hypervisor to export a common set of emulated devices and make them available through the costumed API. Briefly, *Virtio* is easy in the implementation of storage device virtualization. *Virtio* inevitably increases the I/O path which indicates that guest I/O request goes through both guest and host I/O path. Data replication between guest and host can also have an impact on performance.

**Userspace NVMe Driver in QEMU:** Fam Zheng [34] implements a NVMe driver with *VFIO* as the driver of NVMe device to cooperate with the modified userspace NVMe driver in Qemu. Compared with the *VFIO* method, this userspace driver emulates several software layer, that is the BlockBackend, Block layer, and Qcow2 layers in Qemu. This userspace NVMe driver takes advantages of *VFIO*, enabling NVMe device to gain more I/O software layer features at the cost of device sharing capability and this virtualization mechanism brings apparent latency to the VMs.

**SPDK:** The Storage Performance Development Kit (SPDK) is a user-mode application which aims at providing a high performance and scalable I/O application interfaces for different storage devices. It integrates all of the necessary drivers into userspace and operates with an enforced polling mode to achieve high I/O performances by taking advantage of DPDK [14]. Specifically, SPDK application extends "SPDK vhost" to present *Virtio* storage controllers to QEMU-based VMs with *vhost-scsi* and *vhost-blk* [20] interfaces.

## 7 Conclusion

Within this paper, there is introduction of a new virtualization implementation for NVMe storage device MDev-NVMe. The MDev-NVMe is a full NVMe storage virtualization mechanism where all the VMs can run native NVMe driver. The proposed solution takes a mediated pass-through as the main implementation containing the Admin queue emulations and I/O queues shadowing on the basis of Device emulation for all PCI configurations. To achieve the most outstanding performance of the NVMe device among mainstream NVMe virtualization mechanisms, it is proposed that an active polling mode is implemented, which can achieve both high throughput, low latency performance and a reliable device scalability on both Intel OPTANE and P3600 NVMe SSD. With the MDev-NVMe module, the physical device can be partitioned to support device sharing, and each VM can own a full-fledged NVMe device which can directly access the physical performance-critical resources without interference with other VMs. The large numbers of Fio benchmark experiments provides evidence for the great performance of MDev-NVMe, which is better than native performance and other existing virtualization mechanisms. There is also focus and consideration into the influence of blocksize on the I/O performance to provide more suggestions to different I/O applications. Finally, we focus on the high performance of MDev-NVMe by analysis of the performance comparisons between different mechanisms, raising a discussion about polling. Therefore recommendations are provided for the optimization on polling support from both aspects for the design of storage hardware and also, virtualization. In the future, further research on the support for polling will be conducted as well as the optimization of the polling algorithm for MDev-NVMe.

## 8 ACKNOWLEDGMENT

This work was supported in part by National Key Research & Development Program of China (No. 2018YFB1003603), the Program for NSFC (No. 61772339, 61525204), and the Shanghai Rising-Star Program (No.16QA1402200). The corresponding author is Prof. Yao, Jianguo.

## References


[1] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel virtualization technology for directed i/o. *Intel technology journal 10*, 3 (2006).

[2] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Disk-locality in datacenter computing considered irrelevant. In *HotOS* (2011), vol. 13, pp. 12–12.

[3] AWAD, A., KETTERING, B., AND SOLIHIN, Y. Non-volatile memory host controller interface performance analysis in high-performance i/o systems. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on* (2015), IEEE, pp. 145–154.

[4] AXBO, J. Fio: Flexible i/o tester. `https://github.com/axboe/fio`, 2015.

[5] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 8*, 3 (2013), 1–154.

[6] BELLARD, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), vol. 41, p. 46.

[7] BERGMAN, S., BROKHMAN, T., COHEN, T., AND SILBERSTEIN, M. Spin: Seamless operating system integration of peer-to-peer dma between ssds and gpus. In *Proceedings of the Seventeenth USENIX Annual Technical Conference, USENIX ATC* (2017), vol. 17.

[8] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference* (2013), ACM, p. 22.

[9] COLOMBANI, F. A. Hdd, sshd, ssd or pcie ssd. `http://www.storagenewsletter.com/rubriques/market-reportsresearch/hdd-sshd-ssd-or-pcie-ssd/`, 2015.

[10] DONG, Y., DAI, J., HUANG, Z., GUAN, H., TIAN, K., AND JIANG, Y. Towards high-quality i/o virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 12.

[11] DONG, Y., YU, Z., AND ROSE, G. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization* (2008), vol. 2.

[12] DOWTY, M., AND SUGERMAN, J. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[13] HUYNH, K. Exploiting the latest kvm features for optimized virtualized enterprise storage performance. *CloudOpen, North America* (2013).

[14] INTEL. Dataplane performance development kit. `http://dpdk.io/`.

[15] INTEL. Storage performance development kit. `http://www.spdk.io/`.

[16] KEERIYADATH, S. Nvme virtualization ideas for machines on cloud. `http://www.snia.org/sites/default/files/SDC/2016/presentations/solid\_state\_storage/Sangeeth\_NVMe\_Virtualization\_Ideas.pdf`, 2016.

[17] KIM, J., AHN, S., LA, K., AND CHANG, W. Improving i/o performance of nvme ssd on virtual machines. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (2016), ACM, pp. 1852–1857.

[18] KIM, T. Y., KANG, D. H., LEE, D., AND EOM, Y. I. Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on* (2015), IEEE, pp. 1–11.

[19] KLIMOVIC, A., KOZYRAKIS, C., THERESKA, E., JOHN, B., AND KUMAR, S. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 29.

[20] LEI, M. Virtio-blk multi-queue conversion and qemu optimization. `http://events.linuxfoundation.org/sites/events/files/slides/virtio-blk_qemu_v0.96.pdf`.

[21] NVMEXPRESS. Nvm express specification. `http://www.nvmexpress.org/specifications/`.

[22] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. Sdf: software-defined flash for web-scale internet storage systems. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, ACM, pp. 471–484.

[23] RUSSELL, R. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 95–103.

[24] SON, Y., HAN, H., AND YEOM, H. Y. Optimizing file systems for fast storage devices. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015), ACM, p. 8.

[25] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. Gpuvm: Why not virtualizing gpus at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 109–120.

[26] TECHNOLOGIES, A. Storage and pci express a natural combination. `http://www.avagotech.com/applications/datacenters/enterprise-storage`, 2016.

[27] TIAN, K., DONG, Y., AND COWPERTHWAITE, D. A full gpu virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 121–132.

[28] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., LE MOAL, D., BUNKER, T., XU, J., SWANSON, S., ET AL. Dc express: shortest latency protocol for reading phase change memory over pci express. In *Proceedings of the 12th USENIX conference on File and Storage Technologies* (2014), USENIX Association, pp. 309–315.

[29] WILLIAMSON, A. Vfio: A user's perspective. `https://www.linux-kvm.org/images/b/b4/2012-forum-VFIO.pdf`, 2012.

[30] XIA, L., LANGE, J., DINDA, P., AND BAE, C. Investigating virtual passthrough i/o on commodity devices. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 83–94.

[31] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015), ACM, p. 6.

[32] ZHAI, E., CUMMINGS, G. D., AND DONG, Y. Live migration with pass-through device for linux vm. In *OLS08: The 2008 Ottawa Linux Symposium* (2008), pp. 261–268.

[33] ZHANG, Y., AND SWANSON, S. A study of application performance with non-volatile main memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on* (2015), IEEE, pp. 1–10.

[34] ZHENG, F. Userspace nvme driver in qemu. `https://events.static.linuxfound.org/sites/events/files/slides/Userspace%20NVMe%20driver%20in%20QEMU%20-%20Fam%20Zheng_0.pdf`.