NVMe Virtualization for Cloud Virtual Machines

Lixiang Luo, I-Hsin Chung,
Seetharami Seelam, Ming-hung Chen
IBM Research
Yorktown Heights, NY

Yun Joon Soh
UC San Diego
La Jolla, CA

ABSTRACT

Public clouds are rapidly moving to support Non-Volatile Memory Express (NVMe) based storage to meet the ever-increasing I/O throughput and latency demands of modern workloads. They provide NVMe storage through virtual machines (VMs) where multiple VMs running on a host may share a physical NVMe device. The virtualization method used to share the NVMe capability has important performance, usability and security implications. In this paper, we propose three NVMe storage virtualization methods: PCI device passthrough, virtual block device method, and Storage Performance Development Kit (SPDK) virtual host target method.

We evaluate these virtualization methods in terms of performance, scalability, CPU overhead, technology maturity, security, and availability to use one or more of these methods in IBM public cloud.

CCS CONCEPTS

• **Information systems** → **Storage virtualization**; *Information lifecycle management*; • **Computer systems organization** → *Cloud computing*; • **Security and privacy** → File system security.

KEYWORDS

qemu, kvm, virtio, NVMe namespaces, data privacy

ACM Reference Format:
Lixiang Luo, I-Hsin Chung,, Seetharami Seelam, Ming-hung Chen and Yun Joon Soh. 2022. NVMe Virtualization for Cloud Virtual Machines. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3489525.3511688

1 INTRODUCTION

Performance and latency requirements on storage continue to increase, driven by increased data needs of modern workloads such as databases, high performance computing, and artificial intelligence (AI) training [15, 20]. NVMe-based storage offers superior price-performance compared to serial ATA (SATA) and serial attached SCSI (SAS) solid-state drives (SSDs) and hard disk drives (HDDs). Major public cloud providers such as IBM Cloud, Amazon Web Service (AWS), Microsoft Azure, and Google Cloud provide NVMe-based storage for selected system profiles. At the start of this work in 2019, IBM and AWS offer NVMe storage with their selected bare-metal offerings, AWS, Azure, and Google offer the NVMe storage as part of their virtual machine offerings.

NVMe devices can be virtualized and shared among the multiple virtual machines running on a host system. The host operating system (Linux), together with the virtual machine manager (KVM [5]), acts as the hypervisor, which handles the virtualization and sharing of NVMe and other devices with hardware assistance. The hypervisor must ensure the requirements such as the performance, isolation, data security, and quality of service (QoS) are met.

Since different virtualization methods may have different characteristics, we propose and investigate three virtualization methods for a locally attached NVMe device, i.e., PCI device passthrough, virtual block device (backed by either NVMe namespaces or image files), and SPDK [10] virtual host targets.

We evaluate these virtualization methods in terms of performance, scalability, CPU overhead, technology maturity, security, and availability, to enable NVMe storage in virtual machines of IBM's virtual private cloud (VPC) offering. In a public cloud, the cloud provider is responsible for the host operating system and the resources necessary at the host level to safely provide the most performant and isolated capability into the virtual machines, while the end user is responsible for effective use of the resource to feed their applications.

So, in this paper, we answer the following two questions: What are the implications of the different virtualization methods for the public cloud infrastructure provider? How can end users get the best NVMe performance for applications in virtual machines?

We will start with a description of the different NVMe virtualization options.

2 NVME VIRTUALIZATION OPTIONS

Host attached NVMe storage can be provided to virtual machines using different methods. Figure 1 shows three common models of NVMe virtualization. Table 1 provides a comparison of these different models across different dimensions. Different sharing mechanism allows different number of VMs sharing the same NVMe device. This also affects the performance such as the CPU consumption, bandwidth utilization, and QoS control. Additional features including security, resource flexibility (e.g., elastic scaling and live migration) may need to be taken into consideration when deploying the NVMe virtualization. These sharing mechanisms are described in the subsections below.

2.1 PCI device passthrough

Full disk *PCI device passthrough* (DP) is a mechanism to give a virtual machine direct control over a PCI device from the host machine. As shown in Figure 1 with the title "Full disk passthrough", it allows the VM to have exclusive access to the PCI device, which behaves

ICPE '22, April 9–13, 2022, Bejing, China.

ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511688

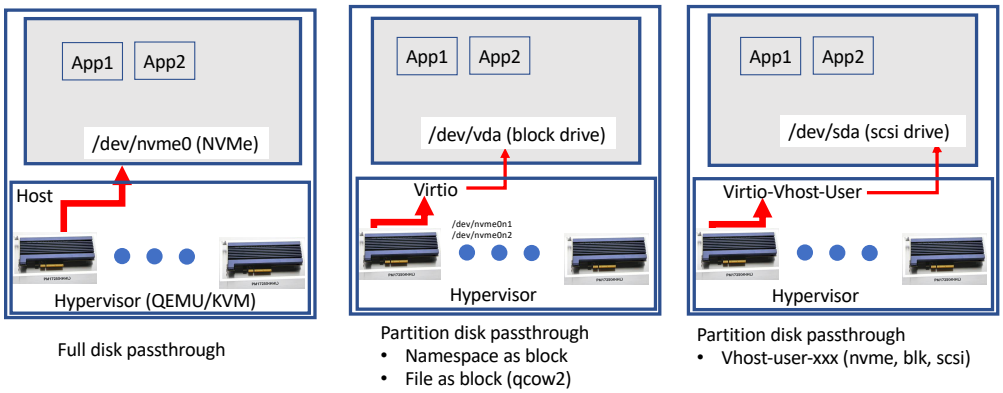


Figure 1: Three models of NVMe virtualization in VM clouds

Considerations	**Disk passthrough**	**Namespace as block**	**SPDK Vhost target**
VMs per disk	1 VM per disk	1 per namespace	Unlimited
Performance	Best (Native)	Limited	Close to native
Elastic scaling	N/A	Possible	Possible
Live migration	No	Yes	Possible
CPU Consumption	None	1 CPU per VM	1 CPU per drive
Security	full control	DoD-3 erase	DoD-3 erase
QoS	N/A	Virtio	Virtio
Maturity	mature	mature but limited	not as mature

Table 1: Comparison of NVMe options under different considerations

as if it is physically attached to the VM. Linux VFIO allows this disk passthrough with support from hardware (such as VT-d in Intel) [22].

When a PCI device is controlled by the assigned VM, the device is not accessible either from another VM or from the hypervisor. The number of PCI devices is limited by the VM system architecture. For example, for Red Hat Enterprise Linux (RHEL) [9], each VM can enumerate up to 32 PCI devices and each device may have up to eight physical functions [8].

With hardware-assisted virtualization, device passthrough can provide the best throughput with the lowest latency, but sharing among multiple VMs is not possible and live migration is very hard to realize. Since the device is exposed to the customers, the customers have the full control over the device and they can access all the features of the device, such as crypto erase before returning to the cloud provider. This technology has been around for a while and it is mature and used widely by several cloud providers. The main drawback of this method is that it is not possible to share a single device across guest VMs, limiting the cloud provider's ability to provide fast storage to multiple tenants.

SR-IOV (Single-Root Input/Output Virtualization) is hardware virtualization feature of the devices that allows sharing devices across guest VMs. SR-IOV allows a single device to provide multiple virtual functions (VFs), each of which can be assigned to a different VM. As this is a hardware-enabled sharing, it has low performance overhead (similar to full device passthrough) while providing most of the hardware features. While experimental versions of SR-IOV enabled NVMe drives are available, they are not yet available for the broader market at the time of this writing. We do expect SR-IOV enabled NVMe drives to be broadly available in the year of 2022-2023 time frame and expect cloud providers to provision NVMe storage via SR-IOV VFs as an alternative.

2.2 Virtual block device mechanisms

This class of virtualization methods includes the traditional file-backed virtual block devices (qcow2) and a more recent option which is backed by NVMe namespaces. Image file-backed storage is one or more files that are stored on the host physical machines file system which acts as virtualized hard drives for the virtual machine. QEMU can emulate file-based storage as many different types of virtual block device interfaces, such as SCSI, NVMe, and virtio-blk. We choose qcow2-backed virtio-blk in this study, because it provides good performance and is the default and the most commonly used.

The NVMe specification [6] allows an NVMe storage device to be divided into one or more `namespaces`. An NVMe namespace is a part of NVMe device that consists of logical blocks. The NVMe controller manages the namespaces such as creation, deletion and access control. The NVMe protocol provides access to namespaces through multiple controllers. Multiple controllers can provide access to the same namespace. Namespaces are commonly supported by enterprise-grade NVMe drives. Namespace-backed virtio-blk in QEMU [7], as shown in Figure 1 with "Partition disk passthrough", is realized by the same virtio block device emulation (virtio-blk) used for SSD/HDD partition passthrough.

As shown in Table 1, virtio-blk provides limited performance compared to device passthrough, with added consumption of CPU cycles for device emulation. However, it allows QoS control and

elastic scaling so the storage space can grow as needed. However, hardware-based crypto erase is not possible inside the VM instance, as the VM only sees a block device.

2.3 SPDK vhost target

SPDK is a collection of libraries and tools that enable users to write high performance, user-mode, scalable storage services. SPDK bypasses the kernel and reduces the path length so applications can fully exploit the high performance characteristics of devices like NVMe. Instead of waiting for the device to interrupt upon I/O completion, SPDK actively polls the device from user space for the progress of I/O operations to minimize or avoid all locks in the I/O path.

SPDK vhost-user is an inter-process communication protocol for device access. Its implementation relies on the vhost-user library introduced in Data Plane Development Kit (DPDK) [3], which exposes a Unix socket so QEMU can communicate to it and offload the tasks of a specific virtio device to an external vhost process. Moreover, by using data structures compatible to QEMU virtio devices and shared hugepage memory, virtio-vhost-user kernel drivers inside the VM can translate I/O requests into direct memory access (DMA) to the SPDK vhost device, largely removing QEMU from the critical path. While configuration and I/O completion still need to be processed through QEMU, the overall performance is expected to be much better than traditional virtio methods. Note that when using SPDK vhost devices with QEMU, the I/O operations on the host is handled by the vhost process using exactly the same polling mechanism as bare-metal SPDK. As indicated in Table 1, this technology is not yet mature and it pegs at least 1 CPU core fully utilized for polling.

2.4 Data security considerations

Data privacy and security are important factors for enterprise tenants, as they do not want to run the risk of leaking business secrets. Since NVMe controllers contiguously re-arrange the logic address mapping to physical memory block to extend the device lifetime, the standards such as DoD 5220.22-M [1] and NIST SP 800-88 [2] for wiping spinning disks are no longer applicable.

Linux Unified Key Setup (LUKS) [13] is the standard software-based solution on Linux to encrypt the partitions and devices with external keys. It allows the tenants to ensure that their data is no longer readable simply by destroying the encryption key. However, since LUKS uses host CPU for encryption, it may introduce significant CPU load in a multi-tenant environment and the NVMe performance may be limited by the CPU performance.

To address the security compliance issues effectively without computing overhead, the cloud provider may rely on new technologies such as NVMe Self-Encrypting Drive (SED) and Trusted Computing Group (TCG) Storage Opal Security Subsystem Class (SSC) feature set for SEDs [12].

SED solves the performance issue by adding an encryption engine to the NVMe controller, which also manages the keys. An NVMe secure erase can wipe all data simply by discarding the current key. Note that the device and namespaces are automatically decrypted once the NVMe controller is ready. Depending on the required security-level, SED may or may not fulfill the security compliance requirement. However, for the environment that only requires to securely wipe the data, which is common for most cloud providers, SED should be sufficient.

TCG Opal SSC for SEDs takes an external key to encrypt the device and its Configurable Namespace Locking (CNL) feature set [11] further adds the per-namespace encryption support. This enables the possibility to use tenants' keys to encrypt their NVMe storage. TCG Opal SSC ensures the device is not accessible once it is closed or powered off, but the device will be readable to everyone after it is powered on and unlocked. Therefore, it is still necessary to implement strict isolation to protect the data. However, at the time of writing, we did not note any product in the market supports CNL feature set and only a small portion of NVMe devices support TCG Opal SSC.

3 PERFORMANCE EVALUATION

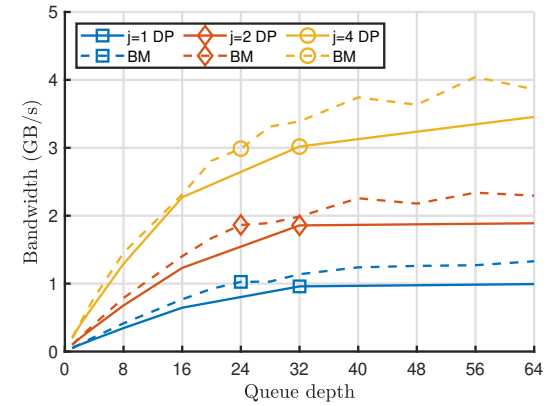
The tests are conducted on a SuperMicro SYS-4029GP-TVRT server, with two Xeon Gold 6148 @ 2.4GHz CPUs and 768GB of DDR4 RAM. The NVMe drive used is Samsung 1725 in a HHHL PCIe Gen3 x8 card form factor, with theoretical peaks of 6.2GB/s sequential read and 2.6GB/s sequential write. We use Flexible I/O tester (FIO) [4] v3.17 on Ubuntu 18.04 with QEMU v4.2.0 to measure the performance numbers with different block sizes, read/write ratios, number of parallel jobs (***j***) and queue depths. The `libaio` engine with direct access is used unless indicated otherwise. Block sizes of 4KB, 16KB, 64KB, 128KB, 1MB, 16MB are used. Our analysis shows that the performance difference between these methods is inversely correlated to the block size, i.e., 4k requests have the biggest performance difference and 16MB requests experience much smaller difference. Due to the limitation of space, we will focus on random 4KB read bandwidth (GB/s) in this paper. Note that IOPS is mostly proportional to bandwidth in 4KB random access tests, so we choose to only present bandwidth in the figures to avoid redundancy.

3.1 PCI device passthrough

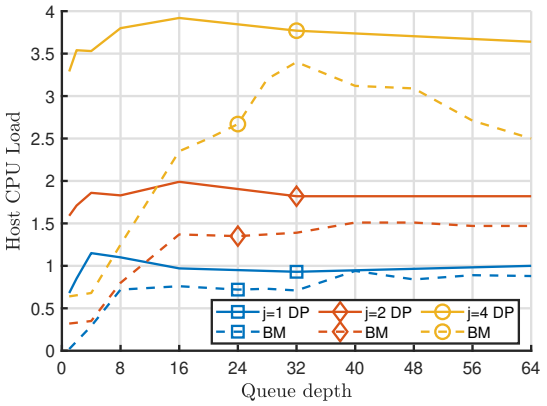
In this section, we investigate two methods of using device passthrough. The first one is to use the default "nvme" kernel driver, and the second is to use the SPDK user space driver. Because of its near bare-metal (BM) performance, device passthrough also serves as the baseline for comparison with other sharing mechanisms.

A comparison of bandwidth between bare-metal and device passthrough is given in Figure 2a. With the mixture of read/write operations the NVMe bandwidth utilization saturates around 4GB/s when using 4 I/O processes. Device passthrough curves closely follow those of bare-metal, with roughly a 10% performance penalty on throughout. Figure 2b shows CPU load, as measured from the host, for both bare-metal and device passthrough. CPU load is the number of processes being executed and waiting to be executed by the CPU. The curves of bare-metal closely follow the trend of the bandwidth curves, as CPU usage for both user application (FIO) and the kernel is driven by the completion rate. On the other hand, the curves of device passthrough observed from the host, stay mostly constant at the value near the number of FIO jobs. Note that all CPU activities are reported as user time for the host CPU. As long as an FIO job is active, the host sees roughly one CPU occupied. Hence, while the performance is similar, executing in a VM results in marginally higher CPU utilization compared to executing the

same workload on bare-metal. We believe this CPU load increase is due to the handling of interrupts coming to VM.



(a) Bandwidth



(b) Host CPU Load

Figure 2: Scaling comparison between bare-metal and device passthrough

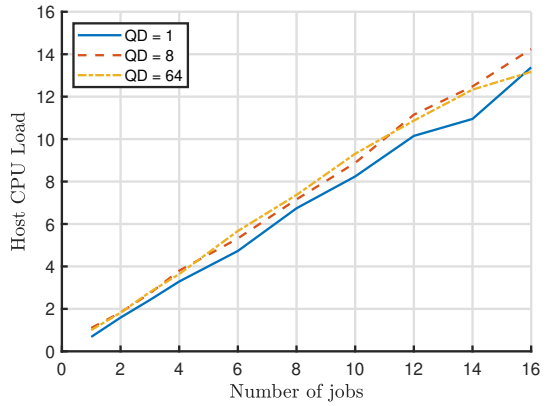
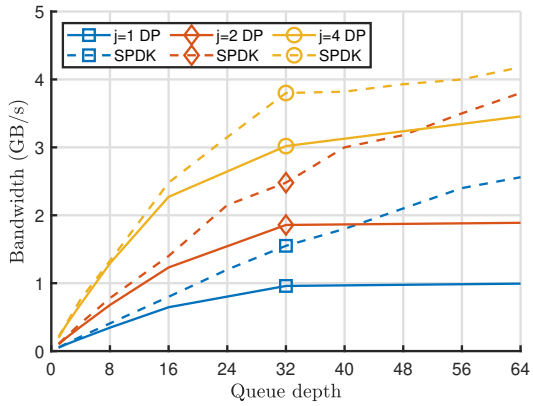


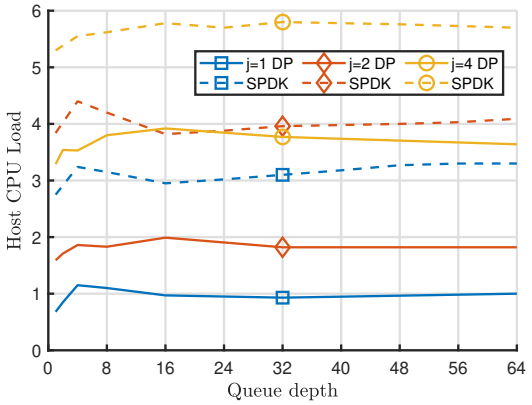
Figure 3: CPU load vs. number of FIO jobs for device passthrough. QD stands for queue depth.

Figure 3 shows the device passthrough's CPU load versus the number of FIO jobs and queue depths. The CPU load grows linearly with respect to the number of jobs, while the queue depth has little impact on CPU load. This is because tasks inside each queue are scheduled sequentially, and scheduling each request is a lightweight task.

It is possible to close the gap between device passthrough and bare-metal by using SPDK inside VM on top of device passthrough. In this scenario, which we call DP-SPDK, a special FIO engine provided by SPDK is used, so that SPDK runs as a part of the user-space FIO process, directly polling the NVMe hardware (as a contrast to the interrupt mechanism). Figure 4a shows how overall bandwidth scales with the queue depth and the number of threads, where it shows that DP-SPDK manages to scale almost linearly until it gets close to the device limit. High concurrency is required to achieve the best overall performance, which can be a serious limitation for the applications that cannot provide enough concurrent I/O tasks.



(a) Bandwidth



(b) Host CPU Load

Figure 4: Scaling comparison between device passthrough and DP-SPDK

DP-SPDK requires CPU resources to actively poll the NVMe device for completion status, as shown in Figure 4b. SPDK will consume user-space resources. We can observe that polling imposes a mostly fixed additional cost, about 2 in terms of host CPU load. An implication of using SPDK in this manner is that the user codes must be programmed to use the SPDK application programming

interface (API) directly. It is therefore non-trivial to migrate an application from using traditional system calls to using SPDK. In the following sections, other sharing mechanisms are considered which do not require explicit source code changes.

3.2 Virtual block device mechanisms

As described in section 2.2, this class of virtualization methods includes the traditional file-backed virtual block devices (qcow2) and a more recent option which is backed by NVMe namespaces. As shown in Figure 5a, these two backend options (qcow2 and NVMe namespaces) provide similar performance for scenarios with low concurrency (synchronous I/O and asynchronous I/O with shallow queues), but they diverge for scenarios with high concurrency, as qcow2 outperforms namespace by up to 40% for very deep queues, qcow2 also incurs slightly less CPU load, as shown in Figure 5b. Such results are likely due to additional host-side caching in qcow2. However, we choose to focus on the namespace option due to its built-in security capabilities, where namespaces can be erased by removing the key so it can be re-provisioned from one user to another quickly (in a few seconds) where as qcow2 image files require the DoD-3 pass erase process that not only takes a long time (~40 minutes for a 1TB drive) but also adversely impacts the lifespan of the drive. In the following discussions, "virtio-blk" and "NS" (only in figures) are used interchangeably to represent virtual block devices backed by NVMe namespaces.

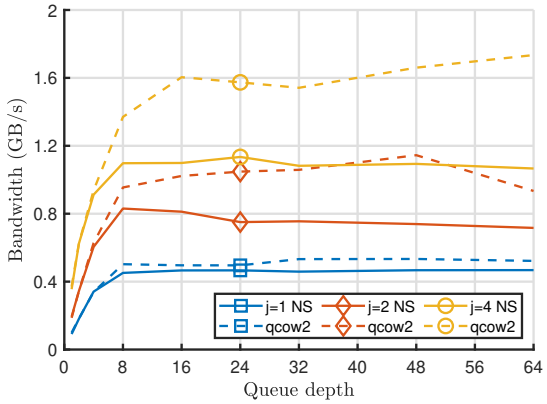
Figure 6a shows a comparison of the performance with virtio-blk and device passthrough options. Overall, device passthrough has a clear advantage for moderate-to-high concurrency scenarios, but virtio-blk is better for low-concurrency scenarios due to additional caching by both the guest-side block device layer and virtio-blk. The flip side is that host CPU load for virtio-blk is constantly higher than the device passthrough due to the involvement of additional software layers. The additional layers also severely limit the maximal throughput in high-concurrency regimes, where the performance is dominated by submission latency. Fortunately, as the number of FIO jobs increases, the difference remains at about 1 to 2 CPU cores. Hence, if the workload inside the VM does not provide high concurrency, virtio-blk can provide better performance at a price of roughly 1 host CPU core.

To achieve the best performance for random access, we turn on the "writethrough" caching by QEMU. For 4K block sizes, RAM requirement for caching is manageable. As the block size increases, more cache memory is needed for high-concurrency scenarios. However, well-behaved applications should avoid such scenarios as NVMe quickly reach peak throughput for larger block sizes even with moderate/low concurrency.

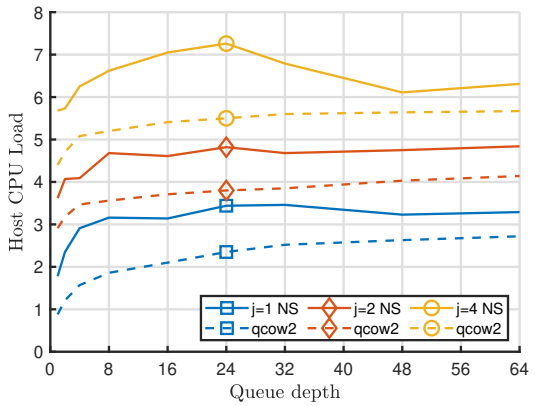
For 4KB high-concurrency scenarios, however, a substantial performance gap persists between device passthrough and virtio-blk, as shown in Figure 6a. Given that SPDK can improve performance earlier, could this improve virtio-blk performance? This is the topic of the next section.

3.3 SPDK vhost targets

SPDK vhost provides another option to pass the NVMe namespaces to VM instances which could achieve better performance for heavy I/O workloads. SPDK supports three types of vhost targets for



(a) Bandwidth



(b) Host CPU Load

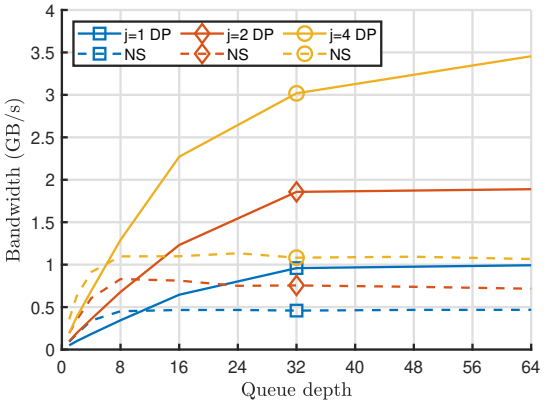
Figure 5: Scaling comparison between NVMe namespace and qcow2 backends of virtual block devices

QEMU, i.e., NVMe (vhost-user-nvme), SCSI (vhost-user-scsi) and virtual block devices (vhost-user-blk). A performance comparison is conducted to evaluate which type of vhost target can provide better performance.

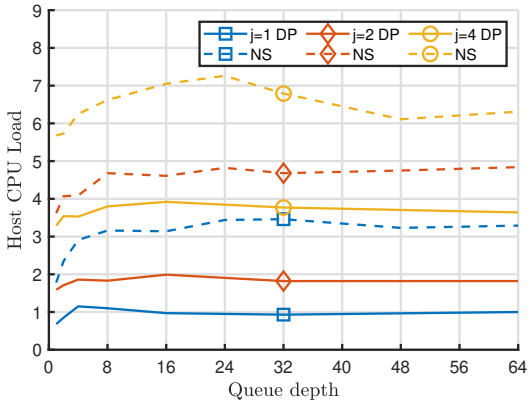
Table 2: Bandwidth (MiB/s) comparison of three sharing mechanisms (vhost-user-xxx)

FIO test	nvme	scsi	blk
randread	764	963	940
randwrite	405	421	415
rand5050	294	302	280

The results are shown in Table 2, where 32-job FIO runs are conducted for various read/write ratios using the default psync engine. Currently, as shown in Table 2, vhost-user-scsi remains the best overall choice for using the SPDK vhost target. Note that there is no fundamental performance limitation for the vhost-user-nvme, compared to vhost-user-scsi. The performance difference is likely due to software maturity in the current implementations. Further references to "vhost" imply vhost-user-scsi, unless indicated otherwise.

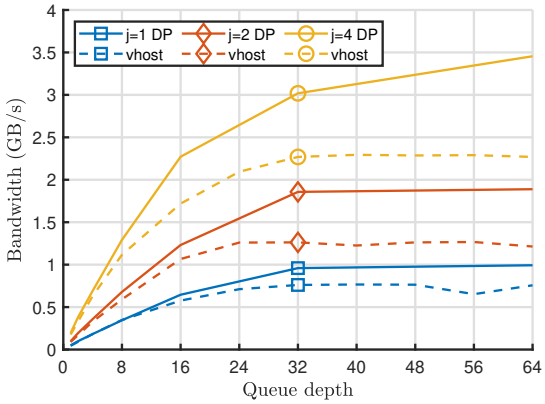


(a) Bandwidth

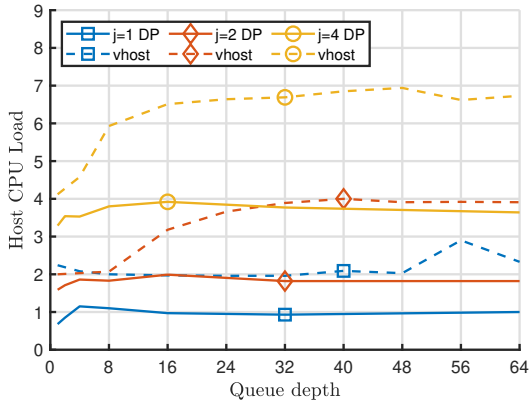


(b) Host CPU Load

Figure 6: Scaling comparison between device passthrough and virtio-blk



(a) Bandwidth



(b) Host CPU Load

Figure 7: Scaling comparison between device passthrough and vhost

Bandwidth curves in Figure 7a show that vhost is similar to the passthrough for shorter queues, with the former performing much better in high-concurrency scenarios. However, in low concurrency scenarios, SPDK vhost targets do not have a clear advantage over virtio-blk, considering vhost requires similar host CPU loads as virtio-blk, as shown in Figure 7b.

Since vhost requires shared hugepage memory for DMA to the devices, its efficiency is limited by available memory capacity. To the best of our knowledge, although vhost may achieve better performance, it has not been adopted by any public cloud provider.

3.4 Completion latency comparison

Completion latency is another commonly used metric to quantify the overall I/O performance. Completion latency is defined as the round-trip time from quest submission until its completion, which is a good indicator of overall overheads. A comparison of completion latency is given in Figure. 8.

Completion latency generally increases as the queue gets deeper. Bare-metal and device passthrough give very close numbers, as device passthrough only adds marginal overheads. Read latency is always larger than write latency, because for NVMe random writes

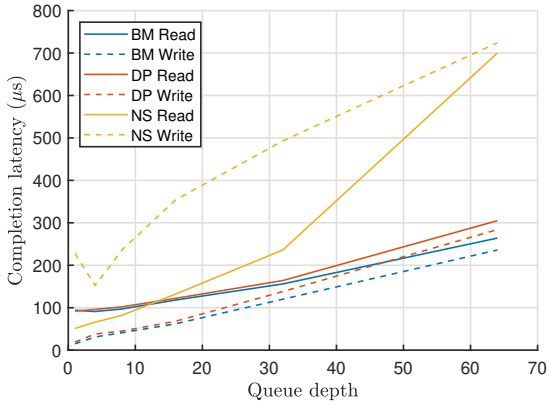


Figure 8: Average completion latency at 95% percentile for 4KB pure read and pure write

are very well buffered at the hardware level, while random reads are much harder to be buffered. The difference is less pronounced for deeper queues. virtio-blk generally gives much larger completion

latency numbers, which is expected as virtio-blk adds significant overheads due to device emulation. A notable exception in this trend can be found for very shallow queue depth (<10), where virtio-blk read latency is smaller than bare-metal and device passthrough, mostly due to host caching, which is consistent with the bandwidth observations.

3.5 Multi-VM and multi-NVMe scenarios

In the real cloud data center, due to the resource consumption requirement, it is common to share one NVMe among multiple VMs as well as having one VM using multiple NVMe devices. To show the performance overhead of sharing a physical NVMe device between the VMs, we compare the FIO and RocksDB performance in multi-VM multi-device. In both scenarios, we compare the virtio-blk and vhost against the bare-metal performance. To make a fair comparison, when measuring the bare-metal performance, we perform our best-effort to match the resource available as in the NVMe virtualization methods. We evenly distribute VMs to multiple NVMe devices and report the aggregated bandwidth and CPU load.

3.5.1 FIO performance in multi-VM multi-device. In Figure 9, we show the aggregated bandwidth and CPU load while varying the number of total NVMe devices. The overall performance is similar to scaling concurrent FIO threads on the single VM. As we increase the number of multiple devices, we observe the virtualization overhead shown as the gap between the bare-metal and the others. For VMs sharing 2 NVMe devices (Figure 9a), we observed that with enough concurrency on the device, virtualization overhead becomes negligible from the performance perspective. This motivated us to run the FIO with multiple threads (Figure 9b). In these experiments, we observed that vhost outperformed bare-metal. With further investigation, we observed that the total number of interrupts was smaller for vhost than the bare-metal, suggesting that the hypervisor selectively passes on the intercepted hardware interrupt to the targeted VM.

3.5.2 RocksDB performance in multi-VM multi-device. While FIO allows us to examine a large parameter space with convenience, we also run a more realistic workload, RocksDB, while varying the number of shared NVMe devices. Widely adopted for key-value data management, RocksDB readily provides a set of built-in benchmarks which can be used for our performance assessment purposes [14]. Particularly, we run the "bulkload" test to quickly establish a test database using 8 billion randomly generated key-value pairs, which occupy roughly 3TB of disk space. Then, we run the "readwhilemerging" test on multiple VMs using the newly generated test database, which simulates a read-dominated workload with occasional merging operations, during which I/O bandwidth and other performance metrics are measured.

Vhost provides similar performance as the bare-metal for single threaded benchmark (Figure 10a), but throttles for a single NVMe device when multiple threads running inside the VM (Figure 10b). A single RocksDB instance, even with optimal resource availability, requires a relatively small I/O throughput (<600MB/s per instance). Each NVMe drive should be able to accommodate several concurrent RocksDB instances, either within the same VM or across multiple VMs. Figure 10a shows that scaling to multiple VM has a mild impact on per-VM performance, as long as the aggregated throughput is kept inside the upper limit of the NVMe drive. Since one NVMe drive can already support 16 VMs (each running one RockDB instance), spreading the workload across more NVMe drives does not improve the per-VM performance. In Figure 10b, the number of concurrent RocksDB instances is increased to 4. As one NVMe drive can no longer provide enough throughput for higher concurrency, the per-VM throughput drops much faster as the number of VMs increases. Note that RocksDB benchmarks such as "readwhilemerging" require heavy CPU involvement, so that the CPU load of the system during the benchmark is always high. For this reason, CPU load is not shown in the figures.

We can see that, unlike FIO results, bare-metal and virtio-blk outperformed vhost in most scenarios. Vhost suffers much worse performance variance than bare-metal and virtio-blk, especially in higher concurrency regimes, as shown in the second row of Figure 10, which gives the throughput difference between best and the worst performing RocksDB instances. Such results suggest a serious resource contention (CPU cores for polling) and/or a lack of a robust load balancing mechanism in the current implementation.

4 CLOUD IMPLEMENTATION CONSIDERATIONS

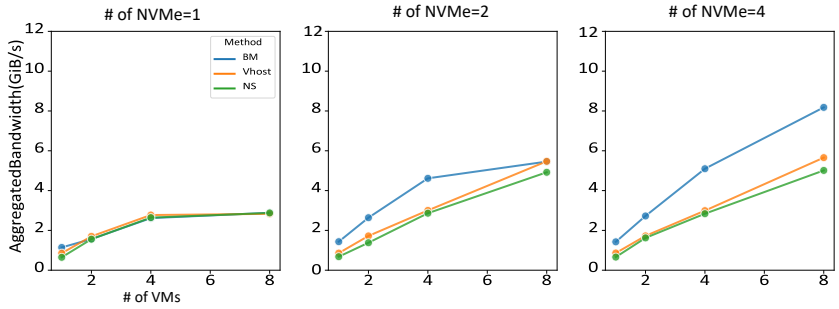
Different NVMe configurations require different handling mechanisms to maintain the lifecycle in the cloud system software. When allocating the NVMe storage resource, the cloud system software will work with the control plane software running on the compute node to find an appropriate node with sufficient residual NVMe resource. The NVMe namespace or virtio block storage is then created by the node-level control plane software, so the cloud system software can initialize the instance with the requested NVMe storage.

When a tenant releases an NVMe storage resource, the cloud system software should always guarantee the data from the previous tenant is securely and fully wiped when reclaiming storage resource. This can be done in seconds via the secure erase feature, which is available on most of the modern data center NVMe devices. However, so far as we know, many NVMe devices may stop responding to all requests when the secure erase is in progress. The cloud system software should be able to take care of this behavior to avoid unexpected failure.

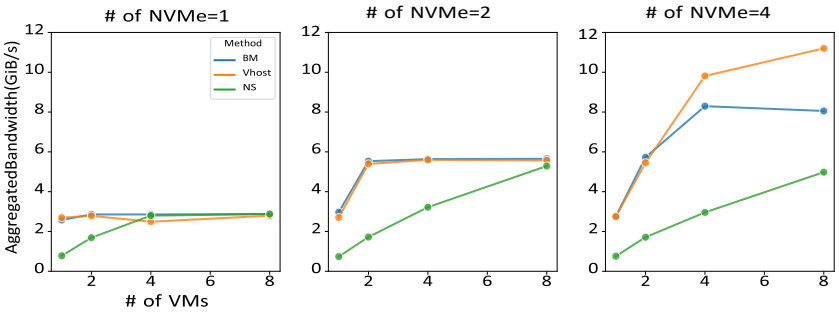
From the Section 3, we can see that in the virtualized environment, device passthrough provides the whole NVMe device to the VM and achieves close to bare-metal bandwidth utilization. This fits the needs for performance-driven applications (e.g., High Performance Computing workloads). For data intensive workloads, the SPDK provides a viable solution to achieve even higher bandwidth utilization by trading system computation resources (e.g, CPU).

When sharing of NVMe is desired (e.g., cost-driven), qcow2 option achieves better bandwidth utilization (due to caching) compared to the namespace-backed option. However, NVMe namespace provides built-in security capabilities. Standard-compliant erase of qcow2 files will take a much longer time than simply dropping encryption keys.

With SPDK, the vhost target using a virtual SCSI interface provides up to 26% bandwidth throughput compared to NVMe and blk



(a) Single-threaded workload (j=1)



(b) Highly concurrent workload (j=32)

Figure 9: FIO result, multiple-VMs sharing devices

options. SPDK vhost target efficiency is limited to memory capacity as it requires hugepage for DMA. This also increases the HOST CPU load.

5 RELATED WORK

The [17] studies the Directly-Attached NVMe Arrays in database systems. They find the arrays (4~16) PCI-attached SSDs offer high capacity at low prices with a bandwidth comparable to DRAM. Our work focused on the sharing mechanism of the NVMe device for multiple VMs.

The paper [18] proposes a new I/O architecture in KVM that optimizes the I/O path by eliminating the overhead of user-level threads, bypassing unnecessary I/O routines and reducing the interrupt delivery delay. Our work starts to look at NS and SR-IOV.

The work [19] evaluates the 800GB ultra-low latency (ULL) SSD prototypes and characterizes their behaviors with I/O path parameters. It also discusses the interrupting, polling modes and SPDK. Our methodology is similar in these aspects but the target is the NVMe device sharing by the VMs.

In [23], it presents a hybrid framework "H-NVMe" to utilize NVMe in the large cloud computing data centers. The H-NVMe offers two modes to deploy VMs, both of which require extensive modifications to the kernel and device emulators. Our work is trying to adopt evolving technologies and to configure NVMe device properly for sharing.

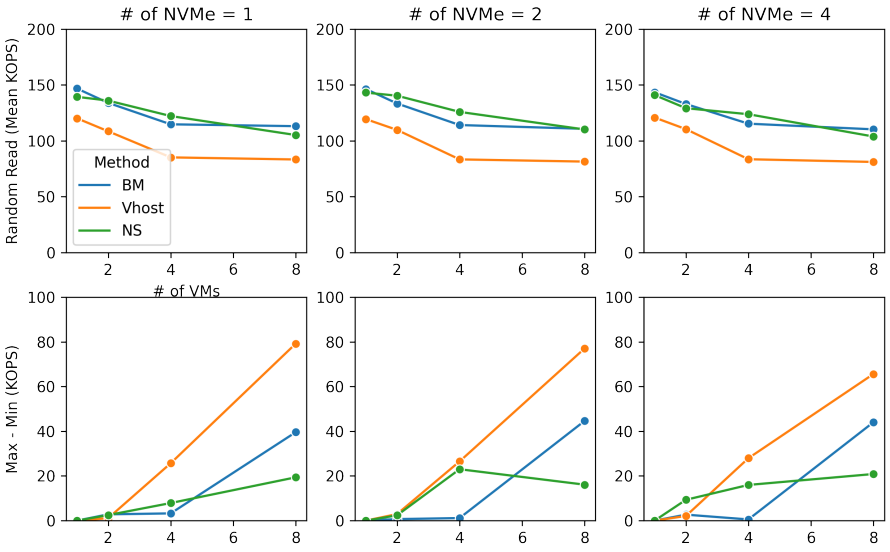
The work [21] investigates CPU utilization behavior with regard to host and guest machines. With hardware assistance and application tuning (e.g., a small number with large chunks of I/O requests), the performance impact of the overhead can be minimized.

NVMe over Fabrics (NVMe-oF) performance characterization (e.g., [16]) profile the overheads of NVMe disaggregation. The work reports negligible performance degradation with NVMe-oF both when using stress-tests as well as with a more realistic key-value store workload. When we are currently focusing on locally attached NVMe storage devices, NVMe-oF is the continuous work we will be focusing on.

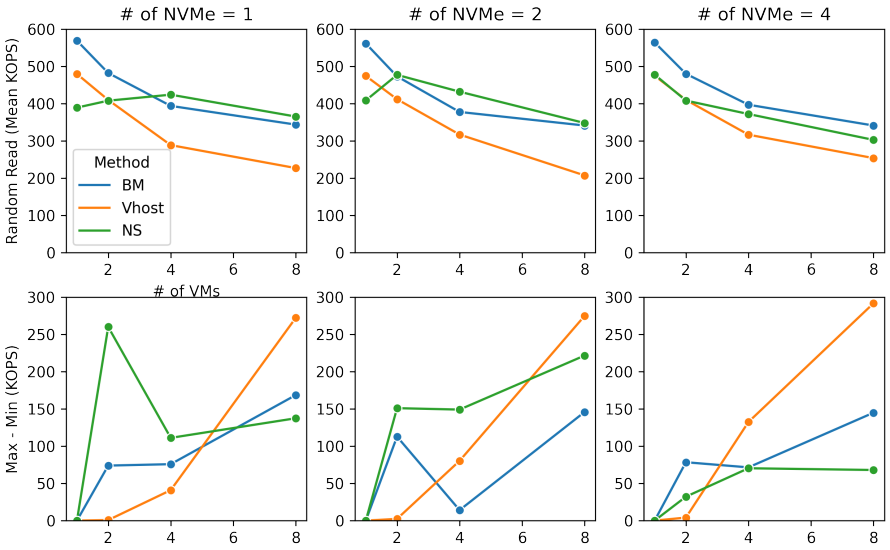
6 CONCLUSION

In this paper, we study different NVMe virtualization mechanisms with comparisons aiming to deploy NVMe at the cloud server scale.

NVMe storage can be provisioned via different mechanisms to virtual machines such as PCI passthrough of the entire device, namespace passthrough, and SPDK virtual host method. While PCI passthrough provides the best performance, it does not allow the host to share the device among multiple VMs. NVMe namespace passthrough allows device sharing among VMs up to the number of namespaces supported by the device. SPDK vhost targets also allow the sharing of the device among virtually unlimited numbers of virtual machines. In addition to the differences in sharing, each of these methods results in different performance characteristics, isolation, security and imposes different overheads on the host and guest CPUs. DP is essentially the same as BM, but lacks of any



(a) Single-threaded workload



(b) Multi-threaded workload (4 threads)

Figure 10: Rocksdb result, multiple-VMs sharing devices

sharing and security mechanism. NS manages to achieve 60% or more of peak BM performance at low-to-medium concurrency scenarios, while providing flexible sharing and security options. SPDK vhost achieves nearly 100% of peak performance at the expense of dedicated CPU resources for polling.

In all methods, applications could achieve near bare-metal performance with minimal changes with optimizations such as increasing concurrency of I/O accesses, polling for I/O instead of interrupt-driven I/O and coalescing I/O requests into larger block sizes.

Data security is critical in a cloud environment. While NVMe devices can be securely shared among users using conventional software-based encryption technologies, emerging NVMe standards such as TCG Opal SSC are expected to provide similar security without the overhead.

As NVMe technology matures, we expect increased adoptions of NVMe in the public cloud and we hope that this paper addressed some of the key trade-offs in this emerging space.

ACKNOWLEDGMENTS

Authors would like to thank Drew Thorstensen, Carl Pecinovsky, Alise Spence from IBM Cloud for their help and collaboration to bring this capability to IBM Cloud.

REFERENCES

[1] 2006 (accessed June 30, 2020). *National Industrial Security Program Operating Manual (NISPOM).* https://www.dcsa.mil/mc/ctp/nisp/
[2] 2014 (accessed June 30, 2020). *NIST Special Publication 800-88 Rev. 1: Guidelines for Media Sanitization.* https://csrc.nist.gov/publications/detail/sp/800-88/rev-1/final
[3] 2020. *Data Plane Development Kit.* https://www.dpdk.org
[4] 2020. *Flexible I/O tester.* https://fio.readthedocs.io
[5] 2020. *Linux Kernel Virtual Machine.* https://www.linux-kvm.org
[6] 2020. *NVM Express Specification.* https://nvmexpress.org
[7] 2020. *QEMU: the FAST! processor emulator.* https://www.qemu.org
[8] 2020. *Red Hat Enterprise Linux 5, Virtualization Guide, Chapter 15. PCI passthrough.* https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/virtualization/chap-virtualization-pci_passthrough
[9] 2020. *Red Hat Enterprise Linux operating system.* https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux
[10] 2020. *Storage Performance Development Kit.* https://spdk.io
[11] 2020. *TCG Storage Opal SSC Feature Set: Configurable Namespace Locking Specification.* https://trustedcomputinggroup.org/resource/tcg-storage-opal-ssc-feature-set-configurable-namespace-locking/
[12] 2020. *TCG Storage Security Subsystem Class: Opal Specification.* https://trustedcomputinggroup.org/resource/storage-work-group-storage-security-subsystem-class-opal/
[13] 2021. *Linux Unified Key Setup.* https://gitlab.com/cryptsetup/cryptsetup/
[14] 2022. *RocksDB Benchmarking Tools.* https://github.com/facebook/rocksdb/wiki/Benchmarking-tools
[15] Jean-Daniel Bonnetot. 2020. *How our Public Cloud instances benefit from NVMe architecture.* https://www.ovh.com/blog/how-our-public-cloud-instances-benefit-from-nvme-architecture/
[16] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. 2018. Performance Characterization of NVMe-over-Fabrics Storage Disaggregation. (2018).
[17] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020.*
[18] Jungkil Kim, Sungyong Ahn, Kwanghyun La, and Wooseok Chang. 2016. Improving I/O Performance of NVMe SSD on Virtual Machines. Association for Computing Machinery, New York, NY, USA.
[19] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. 2019. Faster than Flash: An In-Depth Study of System Challenges for Emerging Ultra-Low Latency SSDs. In *IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019.* IEEE.
[20] Hyungro Lee and Geoffrey Fox. 2019. Big Data Benchmarks of High-Performance Storage Systems on Commercial Bare Metal Clouds. In *2019 IEEE 12th International Conference on Cloud Computing.* IEEE.
[21] Vida Ahmadi Mehri. 2015. An Investigation of CPU utilization relationship between host and guests in a Cloud infrastructure.
[22] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, Keqiang Niu, Sijie Sun, and Minyi Guo. 2020. Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 97–110. https://www.usenix.org/conference/atc20/presentation/xue
[23] Z. Yang, M. Hoseinzadeh, P. Wong, J. Artoux, C. Mayers, D. T. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson. 2017. H-NVMe: A hybrid framework of NVMe-based storage system in cloud computing environment. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC).*