H-NVMe: A Hybrid Framework of NVMe-based Storage System in Cloud Computing Environment

Zhengyu Yang*, Morteza Hoseinzadeh‡, Ping Wong†, John Artoux†, Clay Mayers†,
David (Thomas) Evans†, Rory (Thomas) Bolt†, Janki Bhimani*, Ningfang Mi*, and Steven Swanson‡
*Dept. of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115
‡ Dept. of Computer Science and Engineering, University of California San Diego, San Diego, CA 92093
† Samsung Semiconductor Inc., Memory Solution Research Lab, Software Group, San Diego, CA 92121

***Abstract*—In the year of 2017, more and more datacenters have started to replace traditional SATA and SAS SSDs with NVMe SSDs due to NVMe's outstanding performance [1]. However, for historical reasons, current popular deployments of NVMe in VM-hypervisor-based platforms (such as VMware ESXi [2]) have numbers of intermediate queues along the I/O stack. As a result, performance is bottlenecked by synchronization locks in these queues, cross-VM interference induces I/O latency, and most importantly, up-to-64K-queue capability of NVMe SSDs cannot be fully utilized. In this paper, we developed a hybrid framework of NVMe-based storage system called "H-NVMe", which provides two VM I/O stack deployment modes "Parallel Queue Mode" and "Direct Access Mode". The first mode increases parallelism and enables lock-free operations by implementing local lightweight queues in the NVMe driver. The second mode further bypasses the entire I/O stacks in the hypervisor layer and allows trusted user applications whose hosting VMDKs are attached with our customized vSphere IOFilters [3] to directly access NVMe SSDs to improve performance isolation. This suits premium users who have higher priorities and the permission to attach IOFilter to their VMDKs. H-NVMe is implemented on VMware EXSi 6.0.0, and our evaluation results show that the proposed H-NVMe framework can significant improve throughputs and bandwidths compared to the original inbox NVMe solution.**

***Index Terms*—NVMe Driver, Lock-free I/O Queue, User Mode Polling, Datacenter, Tier and Cache, Big Data, Cloud Computing, Resource Management, Virtualization**

I. Introduction

Ever decreasing price per gigabyte of SSDs (Solid-state Drive) and their capability of fast operation make them indispensable for large-scale cloud services. They are overtaking HHDs (Hard Disk Drive) and leaving them far behind by providing orders of magnitude more IOPS and lowering I/O latency. Thanks to the rapid improvement of the process technology, SSDs can possess several chips. Consequently, they may boost up their bandwidth and capacity by obtaining higher degrees of parallelism. With this regards, Non-Volatile Memory Express (NVMe) [1] interface has been introduced to better utilize the parallelism of the emerging storage technologies.

However, the limitation induced by software overheads prevents users to perfectly perceive this performance advancement. The overhead of the legacy kernel I/O stack, which has been optimized for slow HDDs, is more noticeable as the storage devices and the connection interfaces get faster. There is a tremendous number of researches trying to reduce the kernel overhead by eliminating unnecessary context processing [4]–[6], employing a polling mechanism instead of interrupts [4], [6]–[8], and performance isolation [9], [10].

In addition to the new emerging storage technologies which provide high performance operation through parallelism, container-based virtualization has been a key cloud computing platform which can perfectly take advantages of this parallelism through allowing multiple isolated instances (i.e., containers) of the storage resources. Because of the independence of containers running on top of a single host operating system (i.e., hypervisor), the management of resources through isolation and sharing becomes more important in container-based virtualization. Obviously, having advanced guest kernels is not enough for efficiently sharing system resources. VMware ESXi [2] is one of the most common commercial virtualization platform which can potentially use the NVMe SSDs in the most efficient way by leveraging the massive parallelism and isolation characteristics of NVMe SSDs. Accordingly, VMware ESXi provides Virtual Machine Disks (VMDK) for each VM, which are completely independent of each other. However, in the case of using NVMe SSDs, they access the storage resource through a single submission and completion queue in NVMe driver, regardless of high levels of parallelism provided by NVMe. This inefficiency originates from the NVMe driver in the hypervisor, and has become a bottleneck in the storage I/O stack.

In this paper, we propose H-NVMe, a novel NVMe framework on VMware ESXi. To best utilize NVMe SSDs, H-NVMe provides two different working modes: "*Parallel Queue Mode*" and "*Direct Access Mode*". In the former working mode, H-NVMe circumvents the built-in *Adapter Queue* of ESXi by emptying it and spreading out its entities between multiple lightweight subqueues in our customized NVMe driver in order to use the parallelism of the device more efficiently. The latter working mode bypasses all the hypervisor queues and directly connects the trusted user application threads to the NVMe *Driver Queue*, to achieve better performance isolation. H-NVMe can work in either of these two modes in whole or in partial (details see in Sec. III). It means that both parallelism and isolation can be provided at the same time. We evaluate the performance of H-NVMe with a set of representative applications. The experimental results show that H-NVMe can significantly improve the I/O performance.

The rest of this paper is organized as follows. Sec. II presents the background and motivation. Sec. III proposes our hybrid framework algorithm. Experimental evaluation results and analysis are presented in Sec. IV. Sec. V discusses the related work. We finally present the conclusions in Sec. VI.

II. Background and Motivation

A. Hardware Development

Hardware technology development trends can be categorized into two major paths: storage technology and hardware I/O interface.

1) Storage Technology: The storage technology is the means of maintaining digital data in form of binary information, and plays a key role in the latency of a device. In HDD devices, data is stored in form of sequential changes in the direction of magnetization in a thin film of ferromagnetic material on a disk. Reading and writing to such a device has mechanical nature of the rotating disks and moving heads [11], [12]. Later on, flash memory cells came into play, and a new trend of storage technology formed by introducing Flash-based SSDs [13]. Since flash memory cells need to be erased before storing new data, a Flash Translation Layer (FTL) [14] makes SSDs to present HDD-like I/O interface to an operating system. Even though the latency of an SSD varies under different workloads because of FTL, it is orders of magnitude faster than HDDs. As a result, the new trend in developing SSD devices is rapidly going on to provide cheap storage devices with large storage capacity. There have been a large number of studies on other Storage Class Memory (SCM) including Phase-Change Memory (PCM) [15], Resistive RAM [16], and 3D XPoint [17], which use these storage devices either as primary storage in form of Non-Volatile Dual In-line Memory Module (NVDIMM) [18]–[21], or as secondary storage in form of SSDs [22], [23]. The ultimate goal of developing SCM is to provide near-DRAM latency and also to maintain data persistent. Up to 2017, there is no matured SCM in the market other than flash-based SSDs which will to replace HHDs very soon. Modern SSDs tend to exploit the parallelism of multiple flash chip, and reorder I/O requests for better scheduling [24], in order to amortize per-request latency overhead and achieve high bandwidth by performing concurrent data transfers.

2) Hardware I/O Interface: With the emergence of new and fast storage technologies, hardware I/O bus interface also has experienced significant upgrades to facilitate the use of these modern technologies. Parallel ATA interface (i.e., IDE) has become outdated as it was standardized for slow HDDs with a few megabytes of data transfer per second. Recent fast storage devices usually support Serial ATA (SATA) with a few gigabytes per second transfer rates. However, new storage devices demand much higher transfer rates. The maximum I/O bandwidth is determined by the I/O bus interface which is migrating from IDE and SATA toward PCI Express and is expected to support up to 128 GB/s in PCI Express 5.0 by 2019 [25].

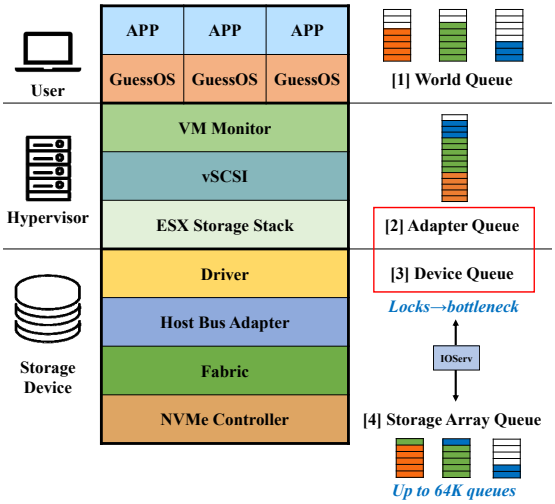

Fig. 1: Architecture of current NVMe-based VM Hypervisor

B. Software Development

Despite the mature infrastructures of recent hardware technology for using fast storage devices, the development of kernel storage stack still needs to pay more attention to these trends.

1) NVMe Driver: In the past, SATA, SAS or Fibre Channel buses were most common interfaces for SSDs. According to the marketplace, SATA is the most typical way for interfacing SSDs and OS. Also, there was a few number of PCI Express based bus interfaces for high-end SSDs, but they use non-standard specification interfaces. After standardizing a universal interface of SSDs, the OS may only need one driver to interact with all SSDs. It is no longer needed to use additional resources to develop specific interface drivers.

Non-Volatile Memory Express (NVMe) [1] is a scalable host controller interface designed for both enterprise and client systems to use SSDs over PCI Express. Recent NVM-Express standard [26] abridges the I/O path with several deeper queues. It provides a device driver which bypasses the block layer and the SCSI I/O subsystem. This driver directly issues the requests to a deeper hardware queue (up to 64K in depth) which enervates the need of background queue running context in the I/O completion path. Therefore, host hardware and software can fully exploit the highest levels of parallelism offered by modern SSDs which form a single unit that plugs into the PCIe bus.

2) I/O Stack in Virtualization Environments: Storage I/O stack in virtualization environments requires low latency. The main challenges are from (1) the presence of additional software layer such as guest OS; (2) context switching between VM and hypervisor; and (3) queuing delay for I/O operations. These challenges do not cause serious problems and lead to high latency when HDDs are dominating the datacenter [27]. However, in the year of 2017, more and more cloud service vendors started to adopt NVMe SSDs into their storage systems, and for some historical reasons, the current popular deployment of NVMe SSDs in cloud computing hypervisor

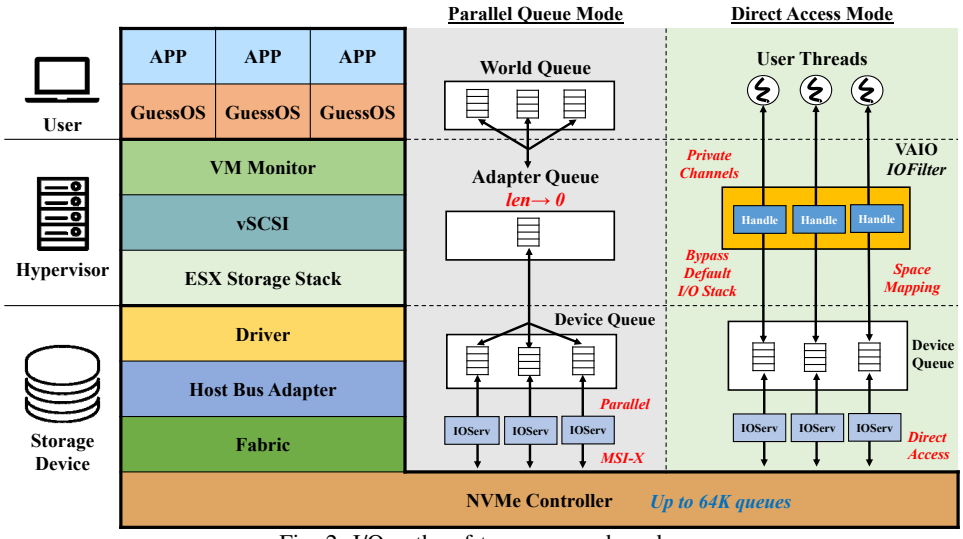

Fig. 2: I/O paths of two proposed modes.

cannot best utilize the NVMe. For example, Fig. 1 depicts a simplified architecture of NVMe I/O stack in VMware's ESXi hypervisor solution (i.e., VMware ESXi Pluggable Storage Architecture (PSA) [28]). We can see that there are multiple intermediate queues in the I/O stack. In detail, when an I/O comes to the current original NVMe framework the following steps are needed:

1) *World Queue* is responsible for each VM I/O requests.
2) *Adapter Queue* in the hypervisor layer gathers jobs from *World Queues*.
3) Hypervisor translates the request to `ScsiCommand()` [29] and sends it to the *Device Queue* in the driver.
4) *IOServ* (I/O server) in the Driver acquires an internal completion *lock*.
5) *Driver* sends I/O to the device, waiting for the device to finish the I/O request, while holding the *lock*.
6) Once the I/O is done, *Driver* releases the *lock* and completes the I/O asynchronously when NVMe controller I/O completion interrupt occurs.

This architecture has many issues. Specifically, VMware's PSA layer virtually completes all I/O asynchronously via interrupts. Historically, interrupt driven I/O is efficient for high latency devices such as hard disks, but it also induces a lot of overheads including many context switches. However, it is not efficient for NVMe SSDs. Most importantly, this interrupt driven approach is not lock-free and limits the parallel I/O capacity, since *IOServs* in th eupper layer will not send another I/O to the NVMe controller until previous submitted I/O returns pending status. Thus, the NVMe's multiple cores/multiple queues mechanisms are not fully utilized.

To further find the evidence of the disadvantage of having these non-lock-free queues, we conduct a preliminary test where we break down the five locks initialized in the submission and completion queues in the *Adapter Queue*. The statistic results in Fig. 3 show that temporal wastes on these locks are not negligible, especially the *lock*1 in submission

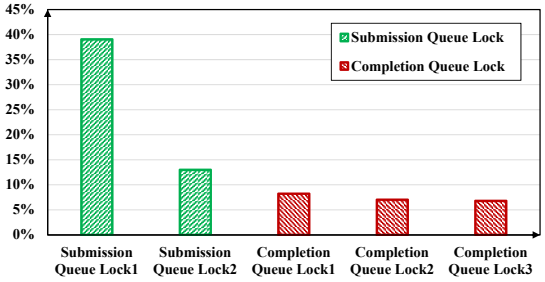

Fig. 3: Normalized time spent of locks in each queue.

queue takes near 40% of the total queuing time. As a result, the huge bottleneck between the *Adapter Queue* and the *Device Queue* makes the system not able to best utilize up to 64K queues in the NVMe controller, in addition to the cross-VM interference issues.

In summary, these issues motivate us to develop a novel solution that is highly parallelism with lock-free queues and supports performance isolation cross VMs.

III. Algorithm Design

In this section, we propose a hybrid NVMe utilization approach called "H-NVMe" to support parallelism and performance isolation by introducing two modes, namely "*Parallel Queue Mode (PQM)*" and "*Direct Access Mode (DAM)*". The PQM is to enforce the *Adapter Queue* to be empty and use our enhanced subqueues in the driver to speed up the I/O stack. This is relatively straightforward to implement and is a more generalized solution. On the other hand, the DAM can achieve better performance by directly connecting those trusted user application threads to the NVMe driver queues through our customized VAIO (vSphere APIs for IOFiltering) IOFilters [3] attached to their corresponding hosting VMDKs. In contrast, this approach breaks the encapsulation of NVMe resources and thus needs higher permission control processes. Based on SLA (Service Level Agreement) of VMDKs owned

by different users, and the security audit requirement (such as attaching IOFilter permission), the cloud manager can select from these two modes for VM deployment in the NVMe-based storage system.

A. Parallel Queue Mode

The challenge of solving the major bottleneck located in the hypervisor layer (observed in Sec. II-B2) is that unlike the "*Device Queue*" encapsulated in the driver layer which we have full control, the "*World Queue*" and "*Adapter Queue*" are much engineering-unfriendly and any changes on them will affect the system capability. Thus, our solution is to "force" the *Adapter Queue* to be emptied and forwarded to the driver layer (through some "*fake completion*" signals from the driver), where the enhanced subqueues can help to increase the parallelism and speed up the I/Os, see in the middle column of Fig. 2 and Alg. 1.

Unlike the original NVMe design where the single *IOServ* has the "lock" issue, H-NVMe allows to create more than one *IOServ* threads and subqueues in the "*Device Queue*" of the NVMe driver to handle the I/O with the NVMe controller. Therefore, the lock-free goal can be achieved by *IOServ* threads to only focusing on their own queues, see in Alg. 1 lines 11 to 13. In our implementation, the $callIOServ$ function selects an idle *IOServ* to assign jobs in the round robin order.

Apparently, having more *IOServs* will indeed improve performance, but it is not free to have infinite *IOServs*, since more CPU and memory resources will be consumed by these threads. As a result, the service rate of each *IOServ* decreases, too. Therefore, the next question that PQM has to address is "*how to dynamically assign subqueue and IOServs number?*", which motivates us to find an adaptive algorithm to automatically selecting a appropriate number of *IOServs*.

Since the the length of *Adapter Queue* is forced to be 0, we can model the problem based on the *M/M/c* queuing theory [30]. Let the total arrival rate (from the *Adapter Queue*) be λ, and let vector $\vec{\mu}$ denote each *IOServ*'s service rate. This vector has c dimensions, where $c \in [1, c_{max}]$ is the number of *IOServs*, and c_{max} is the preset maximal *IOServ* number. If we increase the number of c (e.g., creating and destroying *IOServs*), each server's service rate will change, and this change can be estimated by a regression function (see in Alg. 1 line 24) based on periodically measurement. Once λ and $\vec{\mu}$ are calculated, H-NVMe calls the $optimizeServNum$ function to decide the best number of c. The optimization objective of this function is to minimize the total latency. Specifically, different combinations of c and corresponding service rates $\vec{\mu_c}$ are tested. This procedure also uses the $ErlangC$ function [31] to calculate the probability that an arriving job will need to queue (as opposed to immediately being served). Lastly, to further reduce the cost of changing c, we need to limit the frequency of updating c to a preset update epoch window $epochLen$ in Alg. 1 line 22.

Algorithm 1: Main Procedures of Parallel Queue Mode.

```
1  Procedure IOSubmissionNVMeHypervisor()
2      while OSLibIOSubmit(ctrlr, vmkCmd, devData) do
3          submissionQueue = getSubmissionQueue(ctrlr);
4          IORequest = getIORequest(ctrlr,vmkCmd);
5          if !IORequest then
6              return VMK_NO_MEMORY;
7          IORequest.vmkCmd = vmkCmd;
8          IORequest.devData = devData;
9          OrigLink = submissionQueue.link;
10         IORequest.link = OrigLink;
11         while !update(submissionQueue.link, OrigLink,
             IORequest) do
12             OrigLink = submissionQueue.link;
13             IORequest.link = OrigLink;
14         callIOServ(submissionQueue);
15         return VMK_OK;
16 Procedure callIOServ(submissionQueue)
17     curIOServ=selectIOServ();
18     curIOServ.takeJob(submissionQueue);
19     return;
20 Procedure subQueueNum()
21     while True do
22         if curTime MOD epochLen = 0 then
23             λ = updateArrvRate();
24             μ⃗ = regressSubQueueServRate(C_max);
25             c = optimizeServNum(λ, μ⃗);
26 Procedure optimizeServNum(λ, μ⃗)
27     return argmin_{c∈[1,c_max]} [ErlangC(λ,μ⃗_c)/(cμ⃗_c − λ) + 1/μ⃗_c];
28 Procedure ErlangC(λ, μ⃗_c)
29     ρ = λ/μ⃗_c;
30     return 1 / (1+(1−ρ)[c!/(cρ)^c] Σ_{k=0}^{c−1} (cρ)^k/k!);
```

B. Direct Access Mode

Although PQM can improve the queuing performance by moving all jobs from the *Adapter Queue* to our customized multiple subqueues in the driver, it still cannot simplify the complex VMware I/O stack and thus cannot fully utilize the low latency NVMe SSDs and avoid cross-VM interference. To thoroughly reduce the I/O path complexity and support performance isolation, we develop the "*Direct Access Mode*", which allows *trusted* applications whose hosting VMDKs are attached with our customized VAIO IOFilters to bypass the entire VMware I/O stacks and directly use polling I/Os to the NVMe resource. This user-space I/O strives to fully utilize the performance of NVMe SSDs while meeting the diverse requirements from user applications and achieving performance isolation.

As illustrated in the third column of Fig. 2, in this mode, DAM bypasses the entire VMware I/O stack, and provides *Handles* to grant each *trusted* application in the VM the *root privilege* to manage the access permission of I/O queues in NVMe controller via *Private Channels*. Each user thread is assigned to each application inside VMs. The I/O submissions and completions do not require any driver interventions. As shown in the yellow box, *Handles* will take over all I/O stack of the hypervisor, and map the user space to the NVMe space.

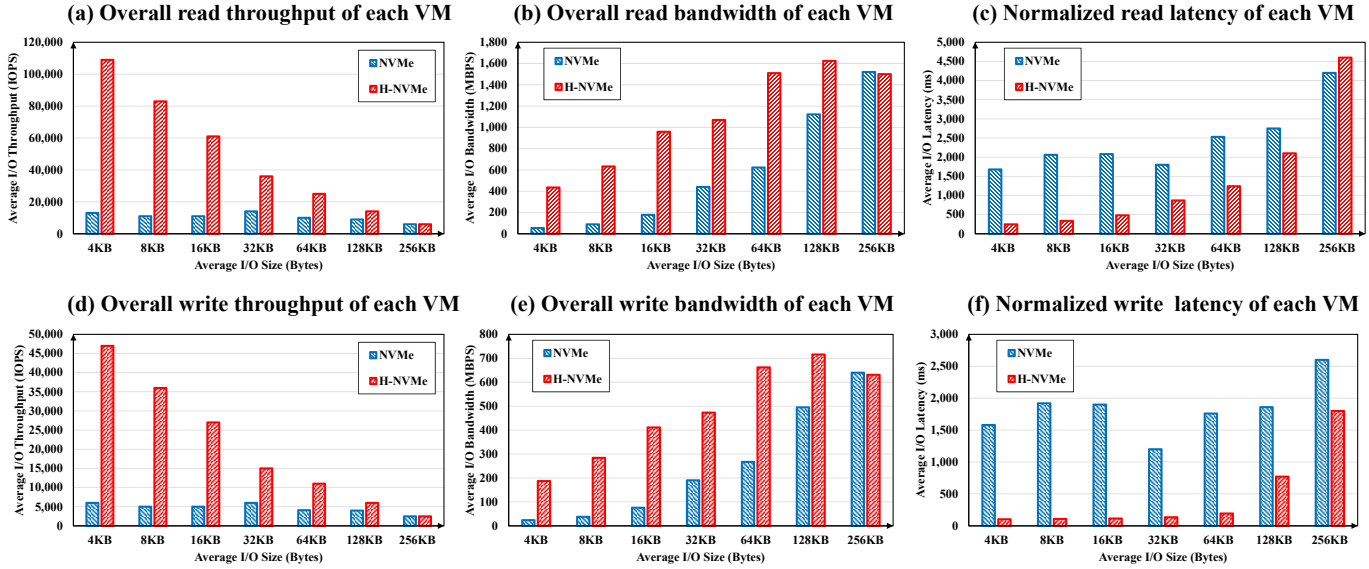

Fig. 4: Throughput, bandwidth, and latency results of 7 different workloads under original NVMe and proposed H-NVMe frameworks.

Next, the *Device Queue* will be directly assigned to each *Handle*, without being forwarded by the *World Queue* and *Adapter Queue*.

Algorithm 2: Main Procedures of Direct Access Mode.

```
1  Procedure IOSubmissionNVMeHypervisor()
2      userLib → kernelDriver ;
3      /* Initialization */ ;
4      if curApp ∈ userLevelWhitelist then
5          new subQueue;
6          new compQueue;
7          subQueue.memoryRegion.map(curApp.memRegion);
8          compQueue.memoryRegion.map(curApp.memRegion);
9          curHandle=new handle(submissionQueue,
             completionQueue, doorbellReg);
10         curHandleList+=curHandle;
11     forwardIOCmd(curHandleList, NVMeCtrlr);
```

Alg. 2 further explains the detail of DAM. When an application requests a single I/O queue to DAM, H-NVMe checks whether the application (as well as its hosting VM) is allowed to perform user-level I/Os. If it is in the whitelist (preset based on SLA and security audit requirements), the customized VAIO IOFilter creates a required submission queue and a completion queue (i.e., "*Private Channels*"), see Alg. 2 lines 4-6. H-NVMe then maps their memory regions (including those associated doorbell registers) to the user-space memory region of the application, in lines 7-8. After this initialization process, the application can issue I/O commands directly to the NVMe SSD without any hardware modification or help from the kernel I/O stack, see Alg. 2 lines 9-11.

IV. Performance Evaluation

A. Evaluation Methodology

We implement the proposed H-NVMe framework on VMware ESXi hypervisor 6.0.0 [2]. Table I summarizes our server configuration. We compare the I/O performance of H-NVMe with the original kernel-based I/O with asynchronous I/O support (i.e., *Kernel I/O*) using the Flexible IO Tester (FIO) benchmark [32]. We allocate around 10% of VMs under Direct Access Mode (DAM) to reflect the premium paid rate in enterprise market [33]. Other VMs are deployed under Parallel Queue Mode (PQM). VMDKs of those DAM VMs are attached with a modified IO Filter [3] driver to support the user polling direct access feature. The IO Filter driver is responsible for collecting and forwarding I/O to *Handles* in Fig. 2, which thus breaks the encapsulation of the VMware and lets VMs directly operate on the NVMe device.

TABLE I: Host server configuration.

Component	Specs
Host Server	HPE ProLiant DL380 G9
Host Processor	Intel Xeon CPU E5-2360 v3
Host Processor Speed	2.40GHz
Host Processor Cores	12 Cores
Host Memory Capacity	64GB DIMM DDR4
Host Memory Data Rate	2133 MHz
Host Hypervisor	VMware ESXi 6.0.0
Storage Device	Samsung NVMe 1725
Form Factor	HHHL
Per Disk Size	3.2TB
Max Seq Read Bandwidth	6,000 MBPS
Max Seq Write Bandwidth	2,000 MBPS

To compare solutions, we evaluate I/O speed results using average I/O throughputs (IOPS), bandwidth (MBPS) and latency for both read and write.

To evaluate the cost of different algorithms, we focus on two metrics, i.e., (1) total runtime of a fixed amount of workload, and (2) the amount of data involved in context switching.

B. Throughput, Bandwidth and Latency of Mixed Workloads

We run 7 workloads with different average I/O sizes on 7 different VMs hosted by the same hypervisor, configuring the overall sequential ratio of each workloads varies from 30%

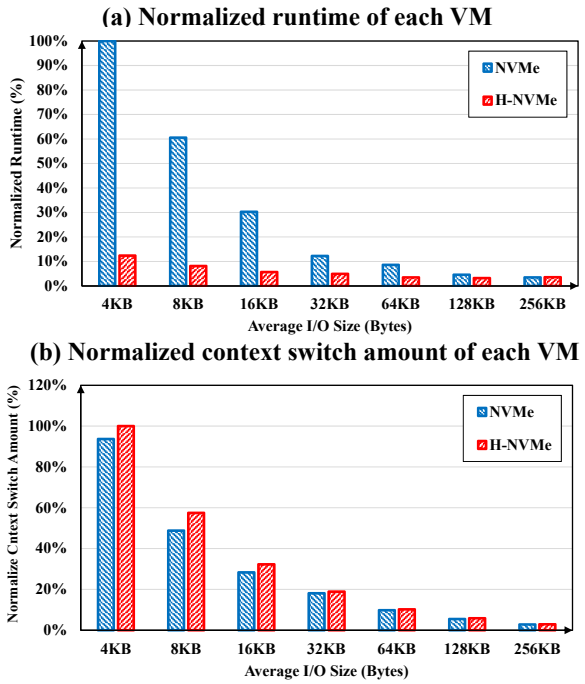

Fig. 5: Normalized runtime and context switch amount of 7 different workloads under original NVMe and proposed H-NVMe frameworks.

to 70%. Fig. 4 illustrates the average throughput, bandwidth, and normalized latency for both read and write I/Os of these workloads under the original NVMe and proposed H-NVMe frameworks. Generally speaking, H-NVMe achieves better performance in all subfigures. We also notice that for small block size workloads, H-NVMe performs much better than the original NVMe framework. However, when workload I/O size increases to 256KB (e.g., the "256KB" bars in Fig. 4(b) and (e)), performance improvement becomes less. This is because larger average I/O sizes have better locality so that there is less room for performance improvement. Similarly, once I/O size reaches 256KB, the read latency of the original NVMe is even slightly better, see in Fig. 4(c).

C. Temporal and Spatial Overheads of Mixed Workloads

We further investigate the overhead of H-NVMe, using Fig. 5 to show both the temporal and spatial overhead of H-NVMe. In detail, from Fig. 5(a), we observe that H-NVMe significantly reduces the total runtime for the same amount of workload compared to the original NVMe framework, especially for those workloads with small I/O sizes. The reason is that unlike workloads with larger I/O sizes having high locality even with the original NVMe framework, workloads with small I/O sizes are more beneficial from PQM's parallel processing subqueues. Additionally, according to our sensitivity analysis, we find that the "sweet spot" for the average number of subqueues in the NVMe driver is 4, which validates our analysis in Sec. III-A that simply deploying more subqueues will not always improve performance.

On the other hand, we evaluate the spatial overhead by measuring the amount of data in the context switching. As shown in Fig. 5(b), H-NVMe has slightly higher spatial overhead, because it needs to maintain multiple subqueues in the driver layer which requires few extra context switches. However, the difference of their spatial overheads gets smaller when the average I/O size increases. In fact, the penalty of this extra spatial overhead is much lower than the time saved by H-NVMe. Therefore, we conclude that by eliminating the synchronization locks contention from multiple queues, H-NVMe can significantly reduce the performance overhead of the I/O stack.

D. Stress Tests on Sequential and Random Workloads

To further investigate the boundary performance improvement brought by H-NVMe under extreme cases, we conduct several stress tests [34] on read- and write-intensive, pure sequential and random workloads in this subsection.

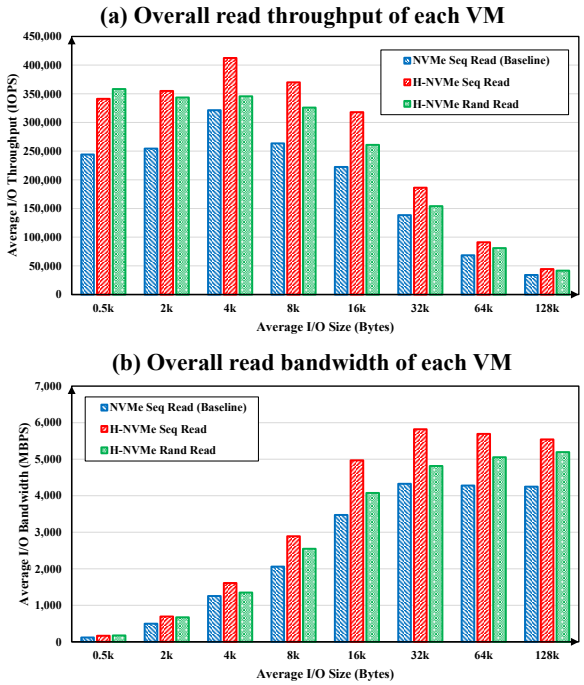

Fig. 6: Read throughput and bandwidth of each VM.

We use the original NVMe's sequential read and write as the benchmark baseline, which reflects the read and write *steady states* (i.e., the "upper bound") under the original NVMe framework. We first present the study on the read-intensive workloads. From Fig. 6(a), we observe that H-NVMe has higher throughput and bandwidth in both sequential and random read than the original NVMe upper bound, which validates the effectiveness of H-NVMe. Furthermore, once the average I/O size is greater than 16KB, H-NVMe's sequential read bandwidth is close to the device capacity (i.e., 6,000 MBPS), and meanwhile, the original NVMe cannot reach that point even at the datapoint of 128KB, see Fig. 6(b).

We next examine the write-intensive sequential and random workloads. As shown in Fig. 7(a), the throughputs of H-NVMe under those sequential write workloads are close to those of NVMe. In fact, even with multiple queues in the I/O stack,

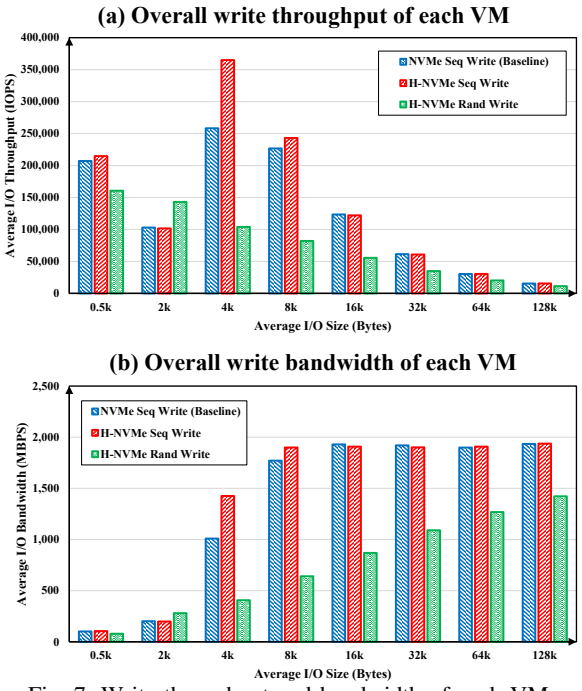

Fig. 7: Write throughput and bandwidth of each VM.

these sequential writes still have relatively low latency, and thus there is no so much room to improve for sequential writes. The only exception from our result is when the average I/O size is 4KB, H-NVMe has an outstanding performance (more than 350,000 IOPS). This ascribes to the fact that this 4KB I/O size is aligned to the cache line size in our implementation. It is worth to mention that the random write throughput of H-NVMe is getting close to the sequential write throughput of the original NVMe, once average I/O sizes are greater than 32 KB.

Similarly, Fig. 7(b) also shows that H-NVMe works slightly better than the original NVMe, and the *steady state* is reached after 8KB, where both NVMe and H-NVMe's bandwidths are close to the device capacity (i.e., 2,000 MBPS) for sequential write. Another observation from Fig. 7(b) is that the random write bandwidth (see the green bar) is increasing linearly with increasing I/O sizes, and thus reduces the gap from the sequential write, which ascribes to H-NVMe's ability to widen the queuing bottlenecks.

V. Related Work

As the hardware latency constantly decreases, many studies have been conducted to diminish the storage stack latency, along the way. Shin et al. [4] present a low level hardware abstraction layer interface which curtails scheduling delays caused by extra contexts to optimize the I/O path. Yu et al. [5] demonstrate six optimization schemes to fully utilize the high performance introduced by fast storage devices. The proposed schemes in [5] rely on a hardware support to expand parallelism inside the SSD. Similar to [4], they also eliminate context switches in the I/O path. Additionally, they exploit polling I/O completion, merging I/O, and double buffering. Likewise, Yang et al. [7] compare polling-based and interrupt-based I/O paths, and eventually come up with the fact that synchronously polling model for I/O completion is much faster in the era of non-volatile memories and very fast SSDs.

To maximize parallelism further, P. Kumar and H. Huang [35] propose Falcon which is a single flush thread per drive, instead of per volume, and separates I/O batching and I/O serving in the storage stack. Similarly, M. Lee et al. [36] propose isolating read and write request through separate I/O queues with the aim of eliminating write interference on read performance in NVMe SSD.

Besides the above designs towards optimizing the I/O path, the kernel also imposes overhead to the storage stack. To address this issue, researchers have suggested to grant direct access to the user applications without involving the kernel. Caulfield et al. [8] present a new hardware/software architecture which achieves high performance by skipping the kernel involvement and leaving the file-system permission checking phase to the hardware (i.e. their special storage device, Moneta [37]). Aerie [38], another flexible file-system, is introduced by Volos et al. to expose the storage devices to user applications for accessing without kernel interaction. HJ Kim et al. [6] propose NVMeDirect which is a user I/O framework allowing user applications directly access commercial NVMe SSDs by associating NVMe I/O queues to them upon request.

Furthermore, the parallelism provided by SSD brings an opportunity for performance isolation for multiple tenants sharing the device. Based on the convex-dependency between IOPS and bandwidth and predicting them, S. Ahn anf J. Kim [9] propose a new throttling technique to provide proportional I/O sharing for container-based virtualization solutions in NVMe SSDs. J. Huang et al. [10] introduce FlashBox, which prepares multiple virtual SSDs running on different channels of a single NVMe SSD.

On one hand, many of these approaches require specialized hardware and/or running complex software frameworks. On the other hand, their focus is on stand alone computer systems running a kernel and several jobs in form of processing threads. However, in data center clusters running hypervisors and several virtual machines, there still exist many unnecessary latency overheads. In contrast, H-NVMe targets the needs in big-data applications which results in a dramatic improvement in large data centers and better performance isolation for VMs by modifying the NVMe driver in VMware ESXi hypervisor.

VI. Conclusion

In this paper, we present a hybrid framework called "H-NVMe" to better utilize NVMe SSDs in the modern super-scale cloud computing datacenters. Our work is motivated by the bottlenecks caused by multiple intermediate queues in the current deployment of NVMe in the VM-hypervisor environment, which cannot fully utilize the maximum performance throughput (up to 64K in depth) of NVMe resources. To solve this bottleneck issue, H-NVMe offers two modes to deploy VMs, i.e., "Parallel Queue Mode" and "Direct Access Mode". The first mode is to increase parallelism and enable lock-free

operations by forwarding jobs from the *Adapter Queue* to the our proposed enhanced subqueues in the driver. This mode is a generalized solution and is relatively straightforward to implement. The second mode allows trusted applications with VAIO IOFilter attached to their user VMDKs to directly access NVMe SSDs and bypass the entire I/O stack in the hypervisor layer to further ensure performance isolation, which suits premium users who have higher priorities and the permission to attach IOFilter to their VMDKs. We implement H-NVMe on VMware ESXi, and our evaluation results show that the proposed framework outperforms the original NVMe solution under multiple benchmarks. In the future, we plan to extend H-NVMe to different hypervisors, such as Xen and KVM.

REFERENCES

[1] "NVM-EXPRESS," http://www.nvmexpress.org/.

[2] "VMware ESXi." www.vmware.com/products/vsphere-hypervisor.html.

[3] "vsphere apis for i/o filtering (vaio) program," https://code.vmware.com/programs/vsphere-apis-for-io-filtering.

[4] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "Os i/o path optimizations for flash solid-state drives." in *USENIX Annual Technical Conference*, 2014, pp. 483–488.

[5] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, "Optimizing the block i/o subsystem for fast storage devices," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 6, 2014.

[6] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds." in *HotStorage*, 2016.

[7] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt." in *FAST*, vol. 12, 2012, pp. 3–3.

[8] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 387–400, 2012.

[9] S. Ahn, K. La, and J. Kim, "Improving i/o resource sharing of linux cgroup for nvme ssds on multi-core systems." in *HotStorage*, 2016.

[10] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds." in *FAST*, 2017, pp. 375–390.

[11] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *Computer*, vol. 27, no. 3, pp. 17–28, 1994.

[12] D. I. Shin, Y. J. Yu, and H. Y. Yeom, "Shedding light in the black-box: Structural modeling of modern disk drives," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS'07. 15th International Symposium on*. IEEE, 2007, pp. 410–417.

[13] R. Micheloni, A. Marelli, and K. Eshghi, *Inside Solid State Drives (SSDs)*. Springer Publishing Company, Incorporated, 2012.

[14] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, no. 5, pp. 332 – 343, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762109000356

[15] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[16] S.-S. Sheu, M.-F. Chang, K.-F. Lin, C.-W. Wu, Y.-S. Chen, P.-F. Chiu, C.-C. Kuo, Y.-S. Yang, P.-C. Chiang, W.-P. Lin *et al.*, "A 4mb embedded slc resistive-ram macro with 7.2 ns read-write random-access time and 160ns mlc-access capability," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*. IEEE, 2011, pp. 200–202.

[17] I. Newsroom, "Intel and micron produce breakthrough memory technology, july 28, 2015." [Online]. Available: http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology

[18] M. Hoseinzadeh, M. Arjomand, and H. Sarbazi-Azad, "Reducing access latency of mlc pcms through line striping," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 277–288, 2014.

[19] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 133–146.

[20] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 2–13.

[21] M. Hoseinzadeh, M. Arjomand, and H. Sarbazi-Azad, "Spcm: The striped phase change memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 38, 2016.

[22] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[23] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," *ACM Transactions on Storage (TOS)*, vol. 10, no. 4, p. 15, 2014.

[24] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, "Ozone (o3): An out-of-order flash memory controller architecture," *IEEE Transactions on Computers*, vol. 60, no. 5, pp. 653–666, 2011.

[25] "PCI Express Specification," http://pcisig.com/specifications/pciexpress/.

[26] A. Huffman, "NVM Express specification 1.1a," http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1a.pdf, Sep 2013.

[27] M. Oh, H. Eom, and H. Y. Yeom, "Enhancing the i/o system for virtual machines using high performance ssds," in *Performance Computing and Communications Conference (IPCCC), 2014 IEEE International*. IEEE, 2014, pp. 1–8.

[28] "VMware Pluggable Storage Architecture Package." [Online]. Available: https://code.vmware.com/vmware-ready-programs/storage/psa

[29] "SCSI Command." [Online]. Available: https://en.wikipedia.org/wiki/SCSI_command

[30] "MMC Queue." [Online]. Available: https://en.wikipedia.org/wiki/M/M/c_queue

[31] "Erlang C Formula." [Online]. Available: https://en.wikipedia.org/wiki/Erlang_(unit)#Erlang_C_formula

[32] "FIO: Flexible I/O Tester," http://linux.die.net/man/1/fio.

[33] O. F. Koch and A. Benlian, "The effect of free sampling strategies on freemium conversion rates," *Electronic Markets*, vol. 27, no. 1, pp. 67–76, 2017.

[34] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge, "Understanding the robustness of ssds under power fault." in *FAST*, 2013, pp. 271–284.

[35] P. Kumar and H. H. Huang, "Falcon: Scaling io performance in multi-ssd volumes," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 2017, pp. 41–53.

[36] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom, "Improving read performance by isolating multiple queues in nvme ssds," in *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*. ACM, 2017, p. 36.

[37] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010, pp. 385–395.

[38] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 14.