An In-Depth Study of Next Generation Interface for Emerging Non-Volatile Memories

Wonil Choi[1], Jie Zhang[3], Shuwen Gao[2], Jaesoo Lee[3], Myoungsoo Jung[3], Mahmut Kandemir[1]
[1]Department of EECS, The Pennsylvania State University [2]Intel Corporation
[3]School of Integrated Technology, Yonsei Institute Convergence Technology, Yonsei University
wonil@camelab.org, jie@yonsei.ac.kr, shuwen@camelab.org, jslee7897@yonsei.ac.kr, m.jung@yonsei.ac.kr, kandemir@cse.psu.edu

***Abstract*—Non-Volatile Memory Express (NVMe) is designed with the goal of unlocking the potential of low-latency, random-access, memory-based storage devices. Specifically, NVMe employs various rich communication and queuing mechanism that can ideally schedule four billion I/O instructions for a single storage device. To explore NVMe with assorted user scenarios, we model diverse interface-level design parameters such as PCI Express, NVMe protocol, and different rich queuing mechanisms by considering a wide spectrum of host-level system configurations. In this work, we also assemble a comprehensive memory stack with different types of emerging NVM technologies, which can give us detailed NVMe related statistics like I/O request lifespans and I/O thread-related parallelism.**

Our evaluation results reveal that, i) while NVMe handshaking is light-weight for flash memory that uses block-based accesses (Block NVM), it can impose tremendous overheads for memristor technology (DRAM-like NVM), ii) in contrast to the common expectation, the performance of an NVMe-equipped system may not improve in a scalable fashion as the queue depth and the number of queues increase, and iii) more- and deeper-queue systems atop a Block NVM can significantly suffer from tremendous host-side memory requirements, whereas a DRAM-like NVM can cause frequent system stalls due to NVMe's inefficient interrupt service routine.

I. Introduction

In the past decade, non-volatile memories (NVMs) have come into the spotlight as a major component of storage systems in various domains ranging from embedded or personal computing to high-performance computing [1], [2], [3], [4], [5], [6]. However, the bandwidth of NVMs by far exceeds that of conventional storage interfaces, due to the significant architectural and technological changes they went through recently [7], [8]. To address the performance disparity between NVMs and storage interfaces, the storage community has improved existing storage interfaces by enabling much higher data transfer rates. For example, the current version of SATA 3.0 [9] and SAS 3.0 [10] interfaces improved the bandwidth by four times compared to their first generations. However, in the meantime, the bandwidth of NVMs has been improved by 152 times, which clearly far exceeds the performance capability of such high speed storage interfaces [11], [12], [13]. Instead of continuing to revise such storage interfaces, one of the promising solutions to accommodate the high bandwidth capacity brought by modern NVMs is to employ a high-speed system memory bus such as PCI Express (PCIe) [14], [15], [16], [17], [18].

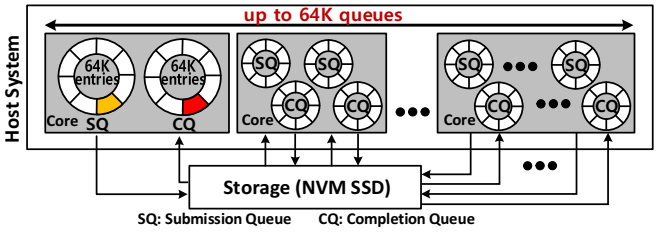


Fig. 1: A high-level view of NVMe. Based on system configuration and user/application demand, the number of queues and queue depth can be scaled.

NVM Express (NVMe) is a brand new interface designed from scratch with the goal of exploiting the potential of high-performance NVMs and standardizing the PCIe-based memory interfaces [19], [20]. NVMe is expected to significantly improve random and sequential I/O accesses by reducing interface latency [21]. Furthermore, it gives better communication flexibility to both the host and NVMs in not only scheduling I/O requests [22] but also buffering data by removing an unnecessary handshaking process [23]. Figure 1 illustrates a high-level view of the NVMe interface. The NVMe interface can maximize parallelism across multiple host-internal resources by supporting a rich queuing mechanism, which allows an NVMe driver to enqueue 64K (=2^{16}) commands. Based on this rich queuing mechanism, the host can secure up to 64K queues, each employing 64K entries. In practice, the number of these host-side queues initiated depends on a wide spectrum of system configurations and user demands. As shown in the figure, different processor cores in the host can employ different number of submission/completion queues in an attempt to avoid locking, hide NVM access penalties, or handle different NVM data structures and I/O services. Thanks to this NVM-specific design, NVMe can accommodate conventional flash memory (imposing block granularity) as well as different types of memristor technologies that use byte-access granularity such as phase change RAM (PCM) [24], [25].

While NVMe promises to enable high levels of I/O parallelism and serve streaming services with extremely large queues, so far these design parameters have not been studied in detail. Furthermore, it is unclear what kind of system considerations and limitations need to be taken into account

when designing a new memory interface, storage stack, and controller based on the NVMe specification. This lack of information on NVMe makes it challenging for not only system designers but also hardware architects to employ NVMe as their new storage interface. As NVMe is defined across host-side system modules and storage-side NVM controllers, the overall system performance highly depends on how systems can take advantage of this next generation interface's unique characteristics. Unfortunately, it is not trivial to explore a large number of design choices configured by 4 billion queue entries with various user scenarios.

In this paper, we propose *NVMeSim*, a novel NVMe analytical simulation tool that models numerous interface-level design parameters such as PCIe/NVMe commands sets, protocols, host interface controllers, and rich queuing mechanisms. To explore the full design space, our NVMeSim also supports the corresponding storage stack as well as different types of NVMs, which can capture not only the lifespan of an I/O request but also its performance. Using this simulation framework, one can investigate both host-level and interface-level design tradeoffs, including different bus speeds, varying access granularities (i.e., DRAM-like NVM accesses [26] vs. block-based NVM accesses), queue utilization, and I/O thread-related parallelism.

The main **contributions** of this work can be summarized as follows:

• *Interface protocol overheads.* The state-of-the-art NVMe 1.2 [20] is designed to minimize handshaking and register writes, thereby enabling streamlined communication without major protocol overheads. Specifically, NVMe needs to write the device-side doorbell registers only two times, which can reduce register accesses of the conventional storage interface protocol [14], [9], [10] by 78%. However, as NVMe is designed to handle a "storage" device over the PCIe electric connection, it still requires extra communication packets to control the payload data movement [14]. These overheads introduce a different set of handshaking problems, which prevent NVMe from achieving a fully-streamlined communication. In this work, we study the details of the communication overheads imposed by NVMe, considering the underlying NVM technology employed in the system.

• *Queue handling issues.* The conventional storage interfaces only offer a single queue with a few entries (e.g., 32 entries), and this queue is in practice shared by all processor cores on the host side. These shared and limited queue entries introduce two main challenges: i) poor device-level resource utilization and ii) performance degradation caused by physical link contention and thread-level I/O synchronization. To address these shortcomings, NVMe provides 64K queues, each capable of queuing 64K incoming I/O requests. In this work, we investigate critical hardware resources and queue management challenges in supporting these rich queueing capabilities of NVMe with varying I/O accesses, threads/cores, queue sizes, and number of queues.

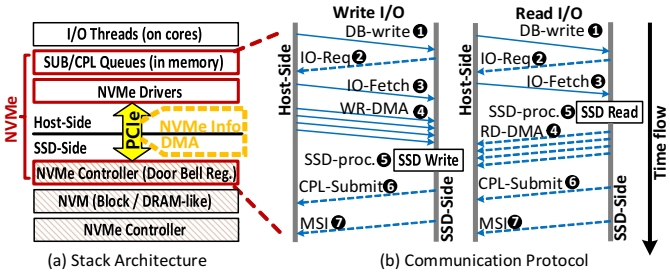


Fig. 2: A typical NVMe interface configuration.

II. Non-Volatile Memory Express

Memory Stack Architecture. Since NVMe is a bus communication protocol between the host and NVM, its memory stack resides on both the sides. Figure 2 shows the overall architecture of an NVMe-based storage system. In the figure, there exist three components integrated in an NVMe-based I/O system: i) *host*, ii) *NVM device*, and iii) *PCIe bus*. The host executes multiple I/O threads, whereas the NVM can employ different types of storage media such as flash memory and memristor memory, referred to as *Block NVM* and *DRAM-like NVM*, respectively, in this paper. Physical PCIe bus plays the role of a communication path between the host and NVM by bridging the both ends. While the host-side NVMe logic can be implemented as a kernel driver, device-level NVMe system parameters such as doorbell registers should be managed by the underlying NVMe controller. The host-side NVMe driver creates important NVMe structures such as *submission queue (SQ)* and *completion queue (CQ)* in the host memory, which are typically managed in pairs and come in multiple pairs. This queue pair (SQ-CQ) can be initiated per-core, per-task, or per-thread basis; in this paper, we allocate a separate queue pair for each I/O thread. The SQ enqueues the incoming I/O requests, whereas the corresponding CQ accommodates the completion messages brought by the NVMe controller. In this architecture, all I/O requests generated by a running thread should be inserted into the SQ and wait for service. These requests are pulled by the NVMe controller, which allows the storage system decide the order of I/O executions without any significant CPU intervention. Once the NVM device completes an I/O request, a completion message is inserted into the associated CQ. A packet-based interrupt triggered by the NVMe controller then finalizes the completed I/O request(s), and the host-side NVMe driver releases the corresponding resources managed by multiple SQs and CQs at the host side.

Communication Protocol. Figure 2b shows an example NVM handshaking protocol between the host and NVM for reads and writes. This communication protocol can be divided into seven sub-steps as follows. When a new I/O request is inserted into the SQ, the driver rings the *doorbell (DB)* of the NVMe controller by writing into the DB register (DB-Write ❶). Once the presence of new I/O requests is conveyed to the underlying NVM, the NVMe controller asks the host to fetch these requests (IO-Req ❷). The driver then sends the corresponding

information to the controller (IO-Fetch ❸). Note that the arrival of new I/O requests at the NVM is not triggered by the I/O submission of the host-side threads, but by the request from the NVM-side controller, which makes the NVM I/O scheduling highly efficient and well balanced. The target data are transferred from the host to the underlying NVM controller via DMA (WR-DMA or RD-DMA ❹). If the request type is "read", the corresponding DMA is initiated after NVM-processing (NVM-Proc ❺). Once the request is processed by the NVM, the corresponding completion message, called *CPL* is sent to the host-side driver and inserted into the target CQ (CPL-Submit ❻). At this point, the host does not know the completion of the I/O request, and therefore, the controller triggers an interrupt to notify the driver that there is a completed I/O request (MSI ❼). This message-signalled interrupt (MSI), initiates an interrupt service routine (ISR), and then the target request can be finalized by the host-side ISR, which subsequently releases the relevant system resources such as queue entries and buffers. It should be noted that the information such as DB-write, I/O-req, IO-Fetch, CPL-Submit and MSI are also packetized and consume PCIe interface bandwidth, in addition to the DMA for the actual data packets.

PCIe Bus and Packet-Based Data Transfer. While NVMe specifies a communication protocol and a rich queue mechanism for emerging NVMs, its physical communication characteristics as well as data movements are dictated by PCIe [14]. This physical PCIe connection, referred to as a *link*, consists of at least one lane and can integrate up to 32 lanes. PCIe also defines the bandwidth of a single link, which increases depending on the number of lanes employed. We summarize the important link characteristics by considering different PCIe bus versions in Table I. All types of data are transferred over PCIe bus in the form of *packets*. In particular, there are two types of packets supported by PCIe: i) *Transaction Layer Packet (TLP)* and ii) *Data Link Layer Packet (DLLP)*. A TLP can contain any message by utilizing its payload position field. Since the maximum PCIe payload size is 4KB, to transfer a chunk of data larger than 4KB, it needs to form multiple TLPs. In contrast, both the NVM and host send and receive DLLPs to manage the status of the bus without any user intervention. For example, DLLPs are utilized as ACK/NAK to send TLPs and notification of the destination buffer status to the source end. Note that all NVMe information described above and I/O data (via DMA) should transfer over the bus in the form of PCIe packets [14]. Even though PCIe is a scalable interface offering extremely high bandwidth (up to 60GB/s per direction), we note that multiple data (packets) *cannot* be transferred in parallel by using multiple lanes; instead, a data transfer should use *all* the lanes until its completion.

Version	**Ver 2**	**Ver 3**	**Ver 4**
Bandwidth /lane	500MB/s	1GB/s	2GB/s
# of lanes	x1, x2, x4, x8, x16, x32		

TABLE I: PCIe configurations for different versions.

NVMe information (TLP)					PCIe (DLLP)	Data (TLP)
DB-Write	**I/O-Req**	**I/O-Fetch**	**CPL-Submit**	**MSI**	**ACK**	**DMA**
24B	24B	20B	20B	20B	8B	4KB

TABLE II: NVMe communication components and their packet sizes.

III. The NVMe Model

Since there is no publicly-available tool to characterize NVMe under a wide variety of storage settings, we developed an *analytical simulation model*, called *NVMeSim*. NVMeSim accommodates four different models: I/O request, PCIe bus, host system, and NVM SSD models. Particularly, the I/O request model and PCIe bus model are constructed in detail based on the NVMe specification [20] and the PCIe datasheets [15], respectively.

I/O Request Model. As illustrated in Figure 2, in order to get serviced, an I/O request should go through a handshaking that involves NVMe information. To extract each component of the NVMe communication protocol, the latencies of reads and writes are modeled based on the NVMe specification and the time taken for the handshaking, which are given below by Equations (1) and (2), respectively :

$$\begin{aligned} T_{ReadI/O} = {} & T_{DB-Write} + T_{I/O-Req} + T_{I/O-Fetch} \\ & + T_{RD-NVM} + T_{RD-DMA} \\ & + T_{CPL-Submit} + T_{MSI} + T_{Stall} \end{aligned} \tag{1}$$

$$\begin{aligned} T_{WriteI/O} = {} & T_{DB-Write} + T_{I/O-Req} + T_{I/O-Fetch} \\ & + T_{WR-DMA} + T_{WR-NVM} \\ & + T_{CPL-Submit} + T_{MSI} + T_{Stall} \end{aligned} \tag{2}$$

where $T_{DB-Write}$, $T_{I/O-Req}$, $T_{I/O-Fetch}$, $T_{CPL-Submit}$, and T_{MSI} are the times taken to process the communication between the host and the NVM device based on NVMe protocol. T_{RD-NVM} and T_{WR-NVM} are the response times of the underlying NVM, whereas T_{RD-DMA} and T_{WR-DMA} are the corresponding latency values for the DMA service. Finally, T_{Stall} indicates the delay for the NVMe information to wait for the bus service; this delay can increase as the queue depth increases. While this I/O request model can capture detailed NVMe handshaking, the communication overheads can be expressed using Equation (3). The magnitude of these overheads can vary, depending on their contribution to the whole I/O latency, by comparing $T_{Overhead}$ with the NVM response time (T_{RD-SSD} or T_{WR-SSD}) and DMA latency (T_{RD-DMA} or T_{WR-DMA}).

$$\begin{aligned} T_{Overhead} = {} & T_{DB-Write} + T_{I/O-Req} + T_{I/O-Fetch} \\ & + T_{CPL-Submit} + T_{MSI} + T_{Stall} \end{aligned} \tag{3}$$

Specifically, the latency taken by an NVMe communication is determined by its information packet size as well as the PCIe bus data rate. The NVMe information moves over the PCIe bus in the form of TLP, which is also defined in the PCIe specification. The size of each type of packet is given in Table II.

PCIe Bus Model. Depending on the PCIe version and the number of lanes, the bandwidth of the selected bus configuration can significantly vary, which can in turn impact the NVMe performance significantly. Note that I/O data (DMA) and NVMe information associated with each I/O request contend

to utilize a single bus in the form of packets. Since the PCIe bus is allowed to transfer one packet at a time (per direction), all types of packets that belong to the large number of I/O requests across multiple queues should wait for the bus service to get to the other side (the host or the NVM).

Host System Model. Among a wide range of resources and components in the host, the I/O queues and I/O finalization latency are accurately modeled in NVMeSim. This model can accommodate 65536 queues, and each queue depth can also vary up to 65536, as indicated by the NVMe specification [20]. In addition, the host memory utilization is also calculated based on the number of queues and the sum of the sizes of all generated I/Os. Finally, this host system model also monitors the finalization of all I/O requests in the system.

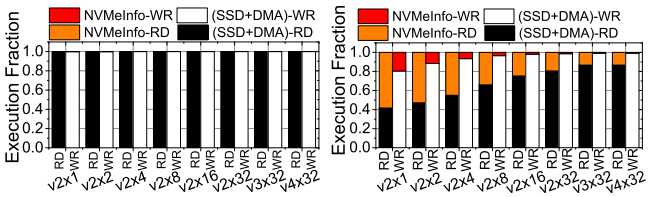
NVM Model. As two different types of memory access granularities impose totally different NVM performance characteristics, we model a block-addressable NVM (e.g., flash) and a byte-addressable NVM (e.g., PCM and STT-MRAM [27]) technologies, referred to as *block NVM* and *DRAM-like NVM*, respectively. For the block-based NVM, we model one of the most common NAND flash memories in the market [28], which on average reads and writes a 4KB page in 30us and 200us, respectively. In contrast, the response time of DRAM-like NVM is borrowed from the timing values of PCM memory [24], read and write latency values which are set for a 64B page, to 50ns and 1us, respectively. While we picked up these specific values for NVMs, our simulator is highly *reconfigurable* and can cover a broad range of NVM characteristics.

IV. Evaluation

While our NVMeSim features a broad range of system design parameters, in this work, we mainly focused on evaluating the NVMe protocol and uncovering the true characteristics of NVMe. To this end, we composed micro-benchmarks that differentiate the I/O request sizes [29] and submission rates based on the type of NVM that the host employs (Block NVM vs. DRAM-like NVM). Specifically, various block access patterns (4KB to 2MB) are generated for block-based NVM, whereas the DRAM-like NVM utilizes byte-access patterns whose sizes range from 8B to 1024B. In addition, the I/O submission rate is carefully modelled in our evaluation as it captures how often the host system generates an I/O request. In reality, this significantly varies depending on not only the system software but also other parameters such as memory space and working-set size. However, to quantitatively evaluate the performance of NVMe protocol, in this paper, we set the I/O submission interval to 10us and 100ns for Block NVM and DRAM-like NVM, respectively, which are reasonable time intervals using which such NVM storage media can serve each request without a device-level scheduling delay.

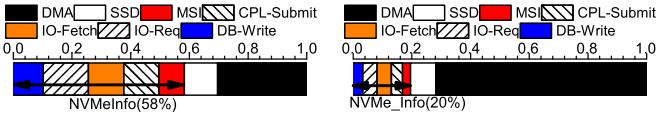
A. Communication Overhead Analysis

In this section, we answer a critical question: *how much overhead is brought up by the NVMe protocol for different types of NVMs in processing an I/O request?* We break down



(a) Block-based 4KB R/W. (b) DRAM-like 64B R/W.

Fig. 3: The latency ratio comparison between the NVMe overheads and other data.



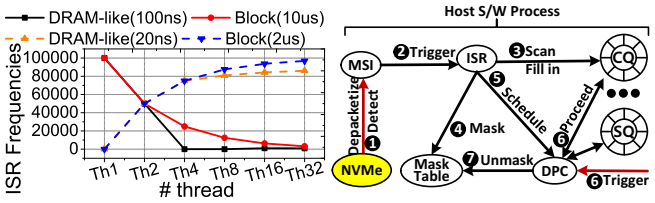
(a) DRAM-like RD breakdown. (b) DRAM-like WR breakdown.

Fig. 4: The breakdown of the DRAM-like NVM execution for 64B read(RD)/write(WR).

the total latency of an I/O request into the NVMe communication overhead (indicated by *NVMeInfo* hereafter) and the other latencies (related to the DMA and NVM-processing). To accurately capture the contribution of each NVMe protocol component, we set the queue depth to one, so that only a single I/O request is triggered into the NVM without being disturbed by other requests.

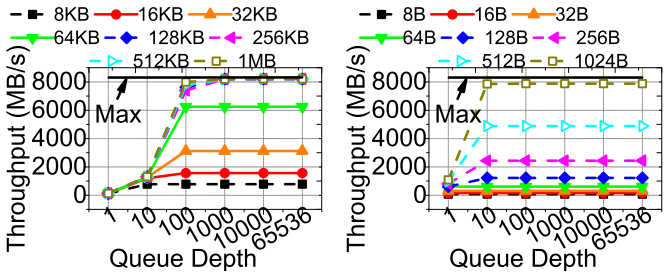
Block NVM storage + NVMe. Figure 3a plots the latency breakdown of 4KB read and write I/O requests for Block NVM. One can observe from this figure that the NVMe communication is streamlined (as promoted), which sets our Block NVM free from the handshaking process and imposes no overhead. Specifically, NVMeInfo for reads and writes only accounts for 0.15% and 0.03% of the total execution latency, respectively, irrespective of which PCIe version is used and how many lanes are employed. This is because the long latency values of Block NVM make the time to transfer the NVMe-generated packets negligible. We note that TLPs used for NVMe communication (listed in Table II) are at most tens of bytes that take a hundred nano-seconds on a PCIe bus. The DMA also takes a small portion of the execution due to the high-speed bus, and this is further reduced by improving the PCIe performance.

DRAM-like NVM storage + NVMe. In contrast to the Block NVM evaluation, the NVMeInfo overhead can be very high in a DRAM-like NVM. Figure 3b shows the execution breakdown for the DRAM-like NVM with the V2x1 PCIe interface (i.e., Version 2 with 1 lane) – we observe that, for most popular PCIe configurations (v2x1 $\sim$ v2x16), the decomposition patterns are similar to each other. One can also see from this figure that the NVMeInfo packets for reads and writes take, on average, 44% and 4% of the total execution time, respectively. This difference in the latency contributions makes the NVMe overhead look bigger in DRAM-like NVM. To extract more detailed information, we further decompose



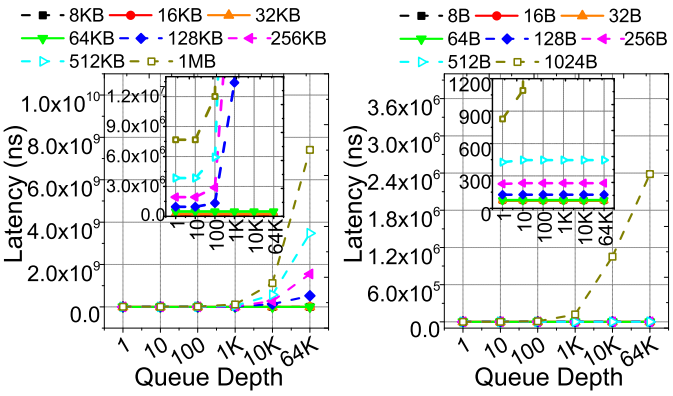
(a) ISR frequencies vs. # threads. (b) MSI handling procedure.

Fig. 5: Overhead analysis for NVMe interrupt handling.



(a) Block NVM. (b) DRAM-like NVM.

Fig. 6: Throughput values with varying queue depths.



(a) Block NVM. (b) DRAM-like NVM.

Fig. 7: Latency values with varying queue depths.

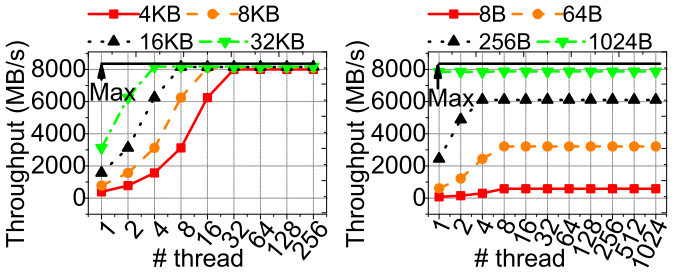
the 64B read and write executions, which are shown in Figures 4a and 4b, respectively. As shown in these figures, while NVMe is oriented towards reducing the communication overhead, the actual I/O services contend with many packets related by NVMeInfo. Specifically, DB-Write, IO-Fetch, IO-Req, CPL-Submit and MSI packets collectively account, on average, for 64% and 39% of the actual data movement (DMA) and NVM itself, respectively. In addition, we also observe that the small DMA size can be a burden on the popular PCIe configuration (v2x4∼16). Fortunately, as the version of the employed bus increases, this overhead can be hidden, which well explains the current trend of employing PCIe interfaces with very high data rates in NVMe. However, as the number of I/O requests increases (as a result of deep queues or DRAM-like small I/O size), the aggregate of these small overheads would be more significant, which can be a problem even in future PCIe buses.

NVMe ISR Overhead. MSI is capable of reducing the communication overhead between the host and NVM as it requires only a few nanoseconds in most modern systems. However, frequent MSI arrivals can impose a long CPU intervention on the host. To better understand the NVMe interrupt overhead, we explain how the host handles MSI in Figure 5b. If MSI is detected, the CPU depacketizes it (❶) and invokes the ISR (❷). The target CQ and new entries therein are then determined via the MSI vector. When multiple queues are associated with the asserted MSI vector, the host driver will scan those queues (❸), which would be a critical bottleneck of NVMe ISR. To perform this scan, the host updates the MSI mask table related to the current interrupt (❹), and then schedules a deferred procedure call (DPC) to process the completed requests (❺). Once DPC is triggered (❻), the driver processes the new completion entries as well as the associated submission entries, and finally unmasks the interrupt (❼).

Figure 5a shows the number of ISR invocations under varying numbers of I/O threads for both Block NVM storage and DRAM-like NVM storage. As each thread has its own SQ and CQ, we use the terms "threads" and "queues" interchangeably when no confusion occurs. While we evaluated different inter-arrival times varying from 2us to 10us and from 20ns to 100ns for DRAM-like NVM storage and Block NVM storage, respectively, for the sake of brevity, in the figure we only show the maximum and minimum values for each configuration. One can observe from this figure that ISRs are invoked too frequently when increasing the number of threads (queues) till 32. As NVMe supports 65536 queues, this ISR overhead may *not* be acceptable in modern systems and will be a serious issue for future NVM technologies. There is a consensus among our observations and the prior works such as [30] and [18]. It should be noted that, for each ISR handling, the host NVMe driver should appropriately handle the entry of CQ and the corresponding one in SQ, and also manage the data memory related to such entries. Thus, the complexities of interrupt handler can increase as the inter-arrival times are shorten, which is the reason why the ISR overheads go up with signalling in the figure.

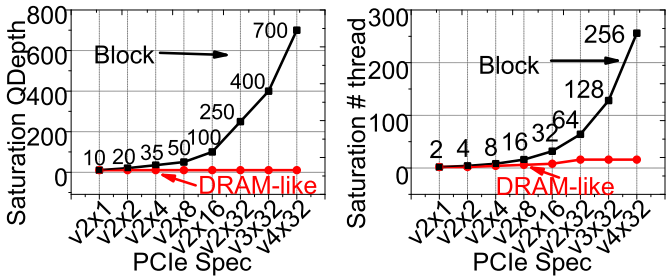
B. Towards More and Deeper Queues

NVMe is designed to support plenty of large queues, which can increase up to $2^{16}queues \times 2^{16}entries = 2^{32}requests$, depending on the user and system requirements. A critical question regarding this rich queue system is whether just increasing the queue sizes will be sufficient to deliver scalable performance. Specifically, *Is there any need to provide up to four billion requests? Is it possible to continuously extract additional performance improvements, as the number of queues and queue entries increases?* To answer these questions, in this section, we mainly quantify the relationship between NVMe and PCIe, and examine the limits of NVMe's rich queue mechanism by utilizing ideal NVM models, which do not have a bandwidth limitation, but capture all relevant NVM latencies.



(a) Block NVM. (b) DRAM-like NVM.

Fig. 8: Throughput comparison with varying numbers of threads (queues).



(a) Queue depth saturation. (b) Thread number saturation.

Fig. 9: Saturation point analysis.

Deep-Queue. Figures 6a and 6b plot the performance of NVMe v2 (x16 lanes) with different request sizes in the case of block NVM and DRAM-like NVM, respectively. Even though NVMe can offer a very large number of queue entries, one can see that the throughput values for both Block NVM and DRAM-like NVM saturate, for any I/O size, before the queue depth reaches 65536. This result indicates that, simply working with a very large queue may not necessarily improve the I/O system performance in a scalable fashion. We also evaluate this saturation issue with a wide variety of PCIe configurations. Figure 9a shows the saturation points for each version of PCIe on all the NVM devices we tested. As shown in the figure, 700 is the sufficient depth to support a Block NVM, whereas the throughput of DRAM-like NVM saturates with a very low queue depth (10), irrespective of which PCIe interface is employed. We believe that this is because the NVMeInfo overhead and DMA are a big burden on the DRAM-like NVM.

Regardless of the throughput saturation, users can fill the entire queue with I/O requests. However, this severely hurts the latencies for both Block NVM and DRAM-like NVM, as can be seen in Figure 7. One can observe from the plots in this figure that, as the queue size increases, the average I/O latency significantly degrades. This is because more and more I/O requests allowed by the deeper queue stall without getting any bus service, once the bandwidth of the given PCIe bus runs out. Because of this, even though NVMe can support 65536 queue entries, after the throughput saturates due to the exhaustion of bus bandwidth, any further increase in queue depth is not recommended, to avoid the latency degradation problem.

Many-Queue. We observed that, as the number of threads increases, the NVM throughput can, in general, improve up to the maximum bandwidth capacity of the given PCIe. Figures 8a and 8b plot the NVMe performance with varying number of threads (and queues) in the case of Block NVM and DRAM-like NVM, respectively. As shown in Figure 8a, the Block NVM can address the bus underutilization issue observed in the previous deep-queue evaluations when increasing the number of queues (32 threads at most). This is because more threads generating small I/O requests are allowed to run concurrently under the block NVM. However, the throughput brought by the DRAM-like NVM by doubling the number of threads is far less than the PCIe bus bandwidth capacity, as illustrated in Figure 8b. Unlike Block NVM, in this case, the reason why the saturated throughput is lower than the theoretical PCIe bandwidth is the significant contribution of the NVMeInfo overhead.

We note that, while the increased number of I/O requests that comes with multiple queues is likely to maximize the utilization of the major I/O system resources, *not* all workloads require 65536 queues. Figure 9b shows the performance saturation points with different thread counts. The performance of the Block NVM saturates with 32 threads, even with the high-speed next generation NVMe (V4) with the maximum lanes (32). On the other hand, unlike the expectation that more threads would be allowed and beneficial for the DRAM-like NVM, at most 16 threads perform the best in the majority of the cases. Lastly, we also observe that that, similar to the deep-queue analysis, continuously increasing the number of threads can significantly hurt the average I/O latency. When the thread count goes beyond the number of threads that saturate the throughput, the latency rapidly degrades, since the saturation point means that the bandwidth of the given PCIe bus has already run out.

V. Conclusions

In this work, we proposed a novel NVMe model implemented in a new simulation platform (NVMeSim), and explored the full design space of NVMe. Even though NVMe is designed towards minimizing handshaking and register writes, we observed its door-bell based I/O management introduces a different set of communication overheads and handshaking problems. We also investigated critical hardware resources and queue management challenges to support NVMe's rich queue mechanism. To our knowledge, NVMeSim is the first framework using which a large variety of NVMe parameters and characteristics can be studied.

VI. Acknowledgement

This research is supported in part by the IT Consilience Creative Program (IITP-2015-R0346-15-1008) and the Next-Generation Information Computing Development Program (NRF-2015M3C4A7065645). This work is also supported in part by NSF grants 1213052, 1205618, 1302557, 1526750,

1409095, and 1439021. Mahmut Kandemir and Myoungsoo Jung are the co-corresponding authors.

REFERENCES

[1] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *In Proc. of the European Conference on Computer Systems*, 2009.

[2] 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," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[3] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt, "Flash on rails: consistent flash performance through redundancy," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[4] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[5] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy, "Ultra-low power data storage for sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, 2009.

[6] H. Shim, J.-S. Kim, and S. Maeng, "System-wide cooperative optimization for nand flash-based mobile systems," *IEEE Transactions on Computers*, 2014.

[7] M. Abbey, "Interface trends for the enterprise i/o highway," in *Flash Memory Summit*, 2012.

[8] K. Eshghi, "Enterprise ssds with unmatched performance," in *Flash Memory Summit*, 2010.

[9] SATA-IO, *Serial ATA Revision 3.1*, 2011.

[10] T10, *Serial Attached SCSI 3 (SAS-3)*, 2009.

[11] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for flash-based ssds with nameless writes," in *the 10th USENIX Conference on File and Storage Technologies*, 2012.

[12] S. Swanson and A. M. Caulfield, "Refactor, reduce, recycle: Restructuring the i/o stack for the future of storage," in *Computer 46(8):52-59*, 2013.

[13] I. Corporation, *Serial ATA Advanced Host Controller Interface (AHCI) 1.3*, 2008.

[14] PCI-SIG, *PCI Express Base 3.0 Specification*, 2010.

[15] ——, *PCI Express Base 4.0 Specification*, 2013.

[16] M. Jung, E. H. Wilson III, W. Choi, J. Shalf, H. M. Aktulga, C. Yang, E. Saule, U. V. Catalyurek, and M. Kandemir, "Exploring the future of out-of-core computing with compute-local non-volatile memory," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[17] M. Jung and M. Kandemir, "Challenges in getting flash drives closer to cpu," in *Hotstorage*, 2013.

[18] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. Le Moal, T. Bunker, J. Xu, S. Swanson *et al.*, "Dc express: Shortest latency protocol for reading phase change memory over pci express," in *In Proc. of USENIX Conference on File and Storage Technologies*, 2014.

[19] N. W. Group, *NVM Express, Revision 1.1b*, 2014.

[20] ——, *NVM Express, Revision 1.2*, 2014.

[21] 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 *Proc. of the 43th IEEE/ACM International Symposium on Microarchitecture*, 2010.

[22] Z. Li, S. Zhang, J. Liu, W. Tong, Y. Hua, D. Feng, and C. Yu, "A software-defined fusion storage system for pcm and nand flash," in *Non-Volatile Memory System and Applications Symposium (NVMSA), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[23] D. Cobb and A. Huffman, "Nvm express and the pci express ssd revolution," in *Intel Developer Forum, San Francisco, CA, USA*, 2012.

[24] S. Eilert and G. Crisenza, "Phase change memory: A new memory enables new memory usage models," in *In Proc. of International Memory Workshop*, 2009.

[25] J.-Y. Jung, K. Ireland, J. Ouyang, B. Childers, S. Cho, R. Melhem, D. Mosse, J. Yang, Y. Zhang, and A. Camber, "Characterizing a real pcm storage device," 2011.

[26] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, 2009.

[27] T. Ohsawa, H. Koike, S. Miura, H. Honjo, K. Tokutome, S. Ikeda, T. Hanyu, H. Ohno, and T. Endoh, "1mb 4t-2mtj nonvolatile stt-ram for embedded memories using 32b fine-grained power gating technique with 1.0 ns/200ps wake-up/power-off times," in *2012 Symposium on VLSI Circuits (VLSIC)*. IEEE, 2012.

[28] MICRON, *Micron NAND Flash Memory MLC, MT29F8G08MAAWC*, 2008.

[29] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-aware i/o management for solid state disks (ssds)," *IEEE Transactions on Computers*, 2012.

[30] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *In Proc. of USENIX Conference on File and Storage Technologies*, 2012.