Enabling Realistic Logical Device Interface and Driver for NVM Express Enabled Full System Simulations

Donghyun Gouk, Jie Zhang and Myoungsoo Jung

Computer Architecture and Memory Systems Laboratory,
School of Integrated Technology,
Yonsei University
kukdh1@camelab.org, jie@camelab.org and mj@camelab.org

Abstract. The data volumes are exploding, immense information has been created more than the storage capacity across all media types over the past 10 years. While the storage systems play a critical role in modern memory hierarchy, their interfaces and simulation models are overly simplified by computer-system architecture researches. Specifically, gem5, a popular full system simulator, includes only Integrated Drive Electronics (IDE) interface, which is originally designed at three decades ago, and simulate the underlying storage device with a constant latency value. In this work, we implement an NVMe disk and controller to enable a realistic storage stack of next generation interfaces, integrate them into gem5 and a high-fidelity solid state disk simulation model. We verify the functionalities of NVMe that we implemented, using a standard user-level tool, called NVMe command line interface. Our evaluation results reveal that the performance of a high performance SSD can significantly vary based on different software stack and storage controller even under the same condition of device configurations and degrees of parallelism. Specifically, the traditional interface caps the performance of the SSD by 85%, whereas NVMe interface we implemented in gem5 can successfully expose the true performance aggregated by many underlying Flash media.

Keywords: Non-volatile Memory Express, Interface, Solid State Disks, Parallel I/O Subsystem

1 Introduction

The size of information is exploded and continues to grow in the coming few years, which in turn can make I/O subsystems critical to all aspects of the modern memory hierarchy [3]. This data explosion also increases the loads of storage systems at an alarming rate, which in turn require improving the capability of data processing. To satisfy the demands of data explosion, storage systems, including the device media and interface, significantly shift their technologies and enhance the performance. For example, the state-of-the-art solid state drives (SSDs) employ hundreds of non-volatile memory (NVM) such as Flash and expose their aggregated performance over multiple channels [15]. As the performance and device characteristics behind these high performance SSDs are by far different with spinning disks, traditional storage interfaces are also evolved into high speed interface and communication protocol, such as NVM Express (NVMe) [1].

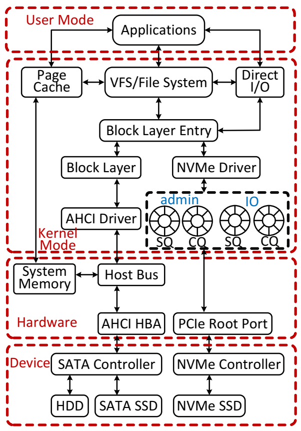
Specifically, NVMe interface defines a software-based communication standard, which is optimized for high performance SSDs through PCI Express (PCIe) bus. The brand-new standard is designed towards not only for a better performance but also for accommodating new storage systems that emply emerging NVM technology such as spin-transfer torque magnetic random-access memory (STT-MRAM) [10] or phase change memory (PCM) [20]. Consequently, the performance of an NVMe-based SSD is nowadays affordable to process a million I/O request per second [18]. However, while the performance of NVMe-based SSDs can blur the boundary of block devices and working memory such as DRAM, the details of storage interfaces and models are overly-simplified by modern system simulation studies. Since many full system simulation models more focus on encompassing system-level architecture and microprocessor architecture, they employ constant latency value for the underlying storage accesses and simulate them over an out-of-date storage interface. For example, gem5 [7] returns 1 us latency for any storage accesses via an Integrated Drive Electronics (IDE) interface class, which is designed at three decades ago [2].

In this work, we enable the full system simulation of gem5 to have a realistic storage stack of next generation interfaces by implementing an NVMe disk and an NVMe controller in its storage datapath. Our NVMe-related components[1] are also integrated with a high fidelity SSD simulator, SimpleSSD [14], which can capture different levels of SSD internal parallelism and the complexity of Flash firmware. With our NVMe implementations, Linux 3.4.112 can appropriately build up NVMe software stack upon gem5 x86 architecture and mount an instance of SimpleSSD into the Linux kernel (`/dev/nvme*`). Our NVMe controller can log all events between the Linux NVMe driver (implemented in Linux 3.4.112) and SimpleSSD, which allows system architects and designers to examine very details of NVMe command sets and all data movements in the DMA procedure of NVMe. The functionalities NVMe we implemented are verified by a user space tool, called NVMe Express command line interface (i.e., `nvme-cli` [16]).

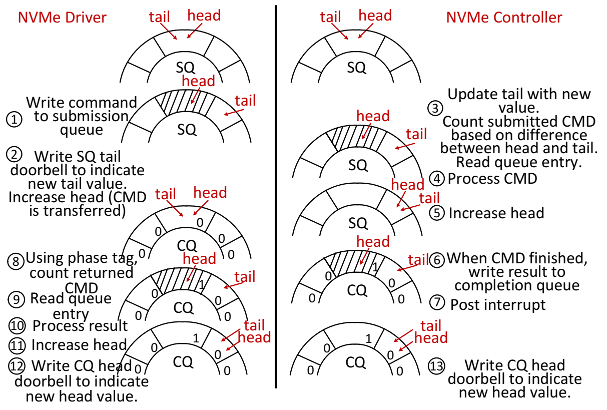
Since the NVMe interface is a relatively brand-new standard, there is no specific study that enables NVMe in a full system simulation and no method that simulates the NVMe logical device interface and driver at system-level. To the best of our knowledge, this is the first paper that simulates a fully-functional NVMe software stack in a full system simulator. In Section 3, we will explain NVMe interface and the corresponding software stack as preliminaries. Section 4 describes the details of our NVMe disk and controller implementations, and Section 5 compares the performance of an NVMe-based SSD and an IDE-based SSD on gem5. Lastly, we conclude this work at Section 6.

[1] All the source codes of our NVMe-related components will be available to freely download

2 Background



(a) Software stack.


NVMe Driver
tail
head
SQ
Write command to submission queue
Write SQ tail doorbell to indicate new tail value. Increase head (CMD is transferred)
CQ
Using phase tag, count returned CMD
Read queue entry
Process result
Increase head
Write CQ head doorbell to indicate new head value.
NVMe Controller
Update tail with new value. Count submitted CMD based on difference between head and tail. Read queue entry.
Process CMD
Increase head
When CMD finished, write result to completion queue
Post interrupt
Write CQ head doorbell to indicate new head value.

(b) Queue operation.

Fig. 1: Linux kernel I/O stack and NVMe Queue operation.

Conventional storage protocols such as SATA, SCSI or SAS are designed for hard disk drives (HDDs), which allows the underlying storage to communicate with CPU through an interface controller (also known as host bus adapter, HBA). For example, the gem5 full system mode simulation employs an IDE controller to handle storage media. Since low-level flash memory is not compatible with conventional block storage due to erase-before-write and in-order write characteristics [13], flash-based solid state drives (SSDs) in practice employ flash firmware that hides all the complexities of underlying flash and makes them suitable to block I/O services over the conventional storage protocols. However, SSDs have significant technology shifts and improve their performance by taking advantage of internal parallelism which in turn allows them to hit the maximum bandwidth that the storage protocols and interfaces can deliver [8, 17]. To address the issues on such performance limits, many SSD architectures employ PCI Express (PCIe), which can address inefficiency of conventional storage protocols. As it is necessary for PCIe to have a host-to-controller protocol, different SSD vendors implement different software stacks by defining their own PCIe storage protocols. This interface variation unfortunately requires per-device operating system modifications and feature implementations. To address such incompatibility and system challenge, non-volatile memory express (NVMe) protocol are proposed as a standard for PCIe-based emerging storage devices.

Figure 1a illustrates the datapath for storage accesses in SATA(left) and NVMe(right). As shown in the figure, NVMe interface removes many unnecessary storage components from the datapath and allows the applications to directly access the underlying SSD through an NVMe driver. In the conventional SATA-based software, the requests arrive at a block layer and are inserted into a queue of the block layer. The requests are then reorderred and issued to SATA by a I/O scheduler such as noop scheduler [4] or anticipatory scheduler [12], which can introduce contention and serialization of a single point contention. In contrast, NVMe can have multiple queues (upto 2^{16}), each of queues is paired by a *submission queue* (SQ) and a *completion queue* (CQ). Similar to native command queue (NCQ) [9], the queue is saved to the host, but it is managed by the NVMe driver and NVMe controller in a cooperative manner. Generally speaking, while new I/O commands to an SQ are managed by the NVMe driver, the NVMe controller places the response into the corresponding CQ.

On the other hand, Figure 1b shows how the NVMe driver and NVMe controller manage an NVMe queue that pairs of an SQ and a CQ. At the initial phase, the head pointer and tail pointer refer a same location of queue entry. If there is an arrival of I/O request, the NVMe driver writes the command to the target SQ (①), which in turn increases the index of tail and rings the tail doorbell to inform the controller about such update of tail index (②). As the tail index is also updated from the controller side, the NVMe controller reads the corresponding entry (③), processes the command by serving it with the underlying storage media (④) and increases the head index (⑤). When the process of the command is completed, the NVMe controller writes a result to the corresponding CQ (⑥), increases its tail index, and posts an interrupt to the host (⑦). Note that, the result, written in CQ, includes the entry ID of SQ and a *phase tag*, which can indicate whether this entry has been updated or not. Thus, when the host receives the interrupt, it can figure out where is the entry of CQ that the NVMe driver needs to handle (⑧). Thereby increasing the tail index based on the state of phase tag. The NVMe driver completes the results by referring the CQ (⑨, ⑩) and increases the head index (⑪). The NVMe driver rings the CQ head doorbell to indicate such update on the head value (⑫), and Finally NVMe controller clears the interrupt if there is no entry to process (i.e., the head is equal to the tail of CQ) (⑬).

Note that all communications and resource management among the driver, controller and device can be done a set of register called *baseline address register* (BAR), including doorbell registers, queue attributes, base addresses for queues, and controller information such as capability, configuration and status.

3 NVM Express Interface and Controller

Figure 2 shows the datapath of NVMe interface that we implement upon a full system simulation model (gem5). While all the working memory (DRAM) banks are connected to the on-chip cache over a crossbar network, the current versions of our NVMe controller and NVMe disk are placed under the I/O controller. For the underlying storage, we integrate a high-fidelity SSD simulation framework, called SimpleSSD [14], under our NVMe disk. A SimpleSSD can simulate multiple channels, each employing multiple flash packages. Each packages can have many dies over multiplexed flash interface such as ONFi [21]. SimpleSSD can simulate host interface layer (HIL) which communicates with our NVMe disk and reconfigurable flash translation layer (FTL) that hides complexities of flash media.

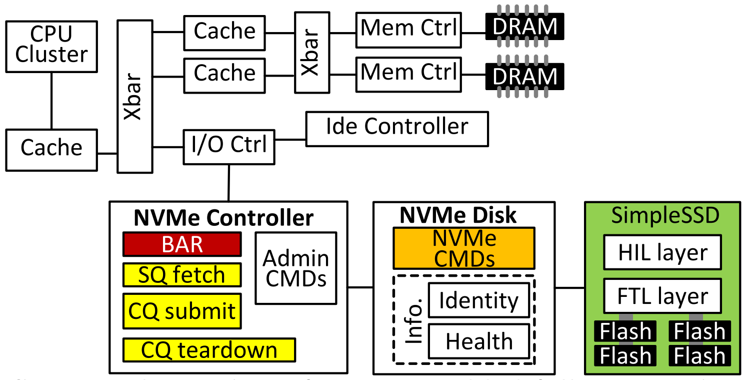


Fig. 2: Structural overview of NVMe-enabled full system simulation.

To enable an NVMe device and build up software stack on SimpleSSD, there are five different functions that should be implemented in gem5. First, PCI/PCIe interface should be appropriately configured by the NVMe controller so that the host can recognize the underlying device as NVMe. In our implementation, the NVM controller configures a class code and a subclass code as a mass storage controller and an non-volatile controller, respectively. In addition, it sets the programming interface to NVMe. Second, the NVMe controller should have memory regions for baseline address registers (BARs). Based on the NVMe specification [11], the NVMe controller should place 1KB-sized controller registers and all doorbell registers, which are supported by the NVMe controller, inside the first BAR region (i.e., BAR0). Third, the underlying device is capable of posting an interrupt for the completion of an NVMe command, which can be handled by i) MSI/MSI-X interrupt or ii) pin-based interrupt. Forth, the NVMe controller should be able to handle the a set of head and tail pointers over doorbell registers for both submission and completion queues (e.g., SQs/CQs). Lastly, the NVMe controller is required to support mandatory admin and NVM (I/O) commands.

If all above five different functions are appropriately supported, operating systems can identify NVMe devices and its kernel driver can communicate with the underlying storage media (e.g., SimpleSSD, in this work). As shown in Figure 2, our NVMe controller enables all the five aforementioned NVMe functions and communicate with the host-side NVMe driver. Specifically, the NVMe driver that exists on gem5, working on the CPU cluster, sends a packet that includes a target address offset (associated with BAR), a size and a pointer that indicates an actual data on host-side DRAMs. Once the NVMe controller receives the packet, it updates an SQ or a CQ based on the type of target doorbell register. The NVMe controller directly serves the request if it is related to the admin mandatory commands. Otherwise, it forwards the I/O services to the NVMe disk module. The disk module is responsible for transforming NVMe command to SSD-related I/O packet, which is managed by the underlying SimpleSSD instance.

In cases where it requires fetching the data from the host, the NVMe controller refers the host-side memory and brings the data over DMA operations, called *SQ fetch*. Once the request is completed by the NVMe disk and SimpleSSD, it writes the target entry of CQ and interrupts the host, which is referred to as *CQ submit*. The NVMe driver then serves the response to the CPU cluster and cleans the corresponding CQ and SQ, called *CQ teardown* in this work.

4 Details of NVMe Command Management

In this section, we will first check functional limits of the current version of gem5 and explain how our NVMe controller and disk module can be integrated into the full system implementation of gem5. We will also describe how NVMe-enabled I/O subsystem is connected to SimpleSSD.

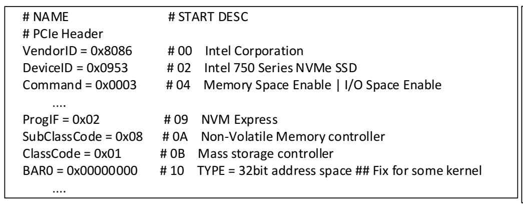
4.1 Interface Limits of gem5

One of the issues to integrate NVMe into gem5 is unability of Message Signaled Interrupts (MSI/MSI-X), which are an alternative in-band method of signalling an interrupt [6]. The current version of gem5 unfortunately has no implementation of MSI/MSI-X for PCI/PCIe interfaces. Specifically, even though gem5 has a variable to abstract an MSI-X interrupt vector table, but it has no interrupt logic or routine to handle this MSI-X variable. While NVMe specification (1.2.1 [11]) recommends to handle interrupts through MSI-X due to a performance concern or MSI to support the backward compatibility, NVMe devices can also post interrupts using a pin-based interrupt mechanism. In this work, to interface the NVMe controller with the full system implementation of gem5, we leverage the pin-based interrupt scheme.

Another issue on the NVMe integration with gem5 is the suport of PCIe interface. Unfortunately, the current version of gem5 has no PCIe implementation that allows the NVMe controller to communicate with the host. PCIe is different with PCI in terms of a bus topology (point-to-point vs. shared bus) and electrical signalling, which in turn introduces a different mechanical form factor and an expansion connector in real world implementations. However, these mechanical form factor and connector issues have almost zero functional impact for full system simulations, and moreover, operating systems communicate with the underlying device over "bus:device.function addresses", referred to as *BDF*. Since the reference mechanism of BDF addresses has no difference between PCIe and PCI interfaces, we integrate the NVMe controller by leveraging a part of gem5's PCI implementation.

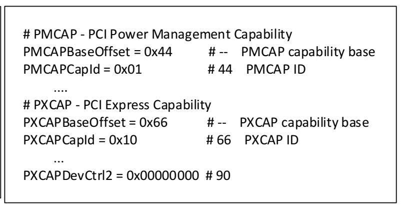
4.2 NVMe Integration

PCI configuration space. In our implementation, operating systems will access the PCI configuration space over `readConfig` and `writeConfig` of gem5 `PciDevice` class. Our NVMe controller updates the PCI configuration as shown



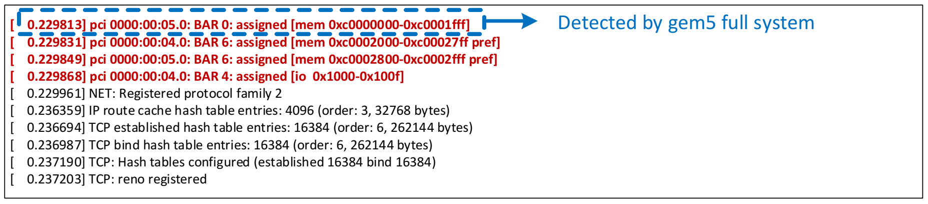
```
# NAME                    # START DESC
# PCIe Header
VendorID = 0x8086         # 00   Intel Corporation
DeviceID = 0x0953         # 02   Intel 750 Series NVMe SSD
Command = 0x0003          # 04   Memory Space Enable | I/O Space Enable
      ....
ProgIF = 0x02             # 09   NVM Express
SubClassCode = 0x08       # 0A   Non-Volatile Memory controller
ClassCode = 0x01          # 0B   Mass storage controller
BAR0 = 0x00000000         # 10   TYPE = 32bit address space ## Fix for some kernel
      ....
```

(a) Example of setting PCI Configuration.



```
# PMCAP - PCI Power Management Capability
PMCAPBaseOffset = 0x44        # --   PMCAP capability base
PMCAPCapId = 0x01             # 44   PMCAP ID
      ....
# PXCAP - PCI Express Capability
PXCAPBaseOffset = 0x66        # --   PXCAP capability base
PXCAPCapId = 0x10             # 66   PXCAP ID
      ...
PXCAPDevCtrl2 = 0x00000000  # 90
```

(b) Example of setting PCIe Capabilities.



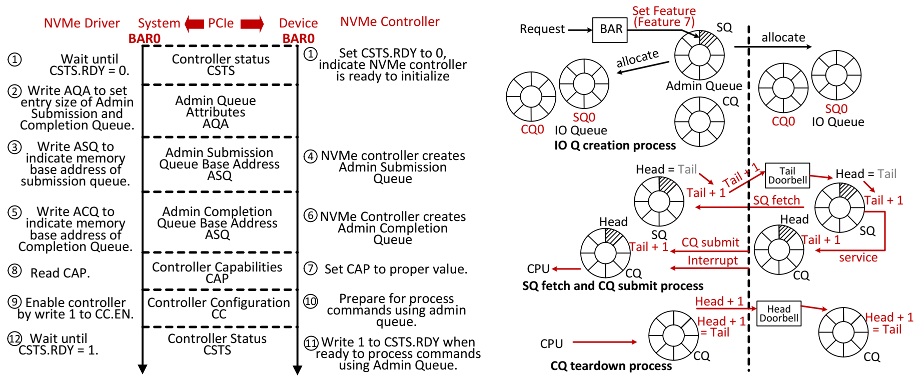
```
[    0.229813] pci 0000:00:05.0: BAR 0: assigned [mem 0xc0000000-0xc0001fff]
[    0.229831] pci 0000:00:04.0: BAR 6: assigned [mem 0xc0002000-0xc00027ff pref]
[    0.229849] pci 0000:00:05.0: BAR 6: assigned [mem 0xc0002800-0xc0002fff pref]
[    0.229868] pci 0000:00:04.0: BAR 4: assigned [io  0x1000-0x100f]
[    0.229961] NET: Registered protocol family 2
[    0.236359] IP route cache hash table entries: 4096 (order: 3, 32768 bytes)
[    0.236694] TCP established hash table entries: 16384 (order: 6, 262144 bytes)
[    0.236987] TCP bind hash table entries: 16384 (order: 6, 262144 bytes)
[    0.237190] TCP: Hash tables configured (established 16384 bind 16384)
[    0.237203] TCP: reno registered
```

(c) Example of our actual logs printed by Linux kernel.

Fig. 3: Example of source codes and debug logs from gem5.

in Figure 3a, so that the host can recognize the underlying storage as an NVMe device. For example, one can observe from the figure, it can specify the device vendor and device IDs (e.g., Intel 750) and define programming interface, subclass, and class as NVMe (0x02), NVM controller (0x08) and mass storage controller (0x01) respectively. While all these will be read as PCI configuration information through `PciDevice` class, the NVMe controller takes over other extended information, which is related to PCIe such as PCI power management and PCIe capabilities as shown in Figure 3b. Lastly, it also configures the size of BAR0, which is used for the NVMe driver to manage admin queues. Once the NVMe controller successfully exposes PCI configuration space to the host, gem5 reports BDFs, which are assigned to the BAR regions as shown in Figure 3c. The size of BAR0 depends on the number of queues that the underlying NVMe controller supports. In this work, we set the size of BAR0 as 8KB which is capable of accommodating 896 queues, by a default.

Device initializations and queue managements. The host (i.e., the NVMe driver) is now able to access the BAR regions through the `read` and `write` functions of `PciDevice`. Figure 4a illustrates the procedure that establishes an admin queue that consists of a pair of SQ and CQ. At the very beginning of a device booting, the NVMe driver (left of the figure) waits the ready flag (RDY) of NVMe Controller Status (CSTS) on BAR0, which is a set of information, including shutdown operations and a controller availability. The NVMe controller (right of the figure) sets CSTS.RDY to zero (0) to indicate that the target NVMe device is ready to initialize (①). The NVM driver then can set the entry size of admin CQ and SQ by updating Admin Queue Attributes (AQA) on BAR0 (②), and then it writes BAR0's Admin SQ Base Address (ASQ) (③). The NVMe controller creates an admin SQ as a response (④), and the host is able to



NVMe Driver
System
PCIe
Device
NVMe Controller
BAR0
BAR0
① Wait until CSTS.RDY = 0.
Controller status CSTS
① Set CSTS.RDY to 0, indicate NVMe controller is ready to initialize
② Write AQA to set entry size of Admin Submission and Completion Queue.
Admin Queue Attributes AQA
③ Write ASQ to indicate memory base address of submission queue.
Admin Submission Queue Base Address ASQ
④ NVMe controller creates Admin Submission Queue
⑤ Write ACQ to indicate memory base address of Completion Queue.
Admin Completion Queue Base Address ASQ
⑥ NVMe Controller creates Admin Completion Queue
⑧ Read CAP.
Controller Capabilities CAP
⑦ Set CAP to proper value.
⑨ Enable controller by write 1 to CC.EN.
Controller Configuration CC
⑩ Prepare for process commands using admin queue.
⑫ Wait until CSTS.RDY = 1.
Controller Status CSTS
⑪ Write 1 to CSTS.RDY when ready to process commands using Admin Queue.
Request
BAR
Set Feature (Feature 7)
SQ
allocate
allocate
Admin Queue
CQ
CQ0
SQ0
IO Queue
CQ0
SQ0
IO Queue
IO Q creation process
Head = Tail
Tail + 1
Tail + 1
Tail Doorbell
Head = Tail
Tail + 1
SQ fetch
SQ
Head
Head
Tail + 1
SQ
CQ submit
Tail + 1
service
Interrupt
CPU
CQ
CQ
SQ fetch and CQ submit process
Head + 1
Head Doorbell
Head + 1 = Tail
Head + 1 = Tail
CPU
CQ
CQ
CQ teardown process

(a) Admin queue creation. (b) Queue process.

Fig. 4: Our implementation for NVMe at gem5.

writes BAR0's Admin CQ Base Address (ACQ) through BAR0 (⑤). The NVMe controller then reports the Controller Capabilities (CAP) that include a set of controller-specific information such as memory page sizes or scheduling strategies (⑦), and the NVMe driver can be aware of them (⑧) by reading out BAR0. The NVMe driver enables the admin queue by writing up the enable flag (EN) of Controller Configuration (CC) (⑨), and the NVMe controller prepares the admin queue to operate (⑩). Finally, the NVMe controller writes 1 to CSTS.RDY when the queue is ready to use (⑪), so that NVMe driver that waits for CSTS.RDY can start to handle I/O services.

Controller and doorbell registers. Once an admin queue is created at the device initialization phase, the NVMe driver can create a corresponding I/O queue by pairing NVMe SQ and CQ again. In contrast to the process of admin queue creation, the NVMe driver uses the admin queue to allocate the I/O queue and communicate with the underlying NVMe controller. Specifically, the NVMe driver sends a set feature command (especially feature 0x07 for the queue allocation) to the admin SQ, as shown in Figure 4b. After the allocation of I/O SQ and CQ, the head and tail pointers indicate the same entry of SQ and CQ. Whenever the NVMe driver sets a command request to SQ, it increases the tail index and informs the update over the tail doorbell register to the NVMe controller. Our NVMe controller than fetches the data (or serve the data) through `dmaRead` and `dmaWrite` of `PciDevice`. Once it is completed, the NVMe controller updates CQ and increases the tail pointer. The host then checks the phase tags of CQ and updates the corresponding tail. During this step, our NVMe controller interrupts the host via `intrPost` of `PciDevice`.

5 Evaluation

5.1 Methodology

gem5 configurations. We simulate the NVMe interface and driver on gem5's x86 full system mode. We use Linux 3.4.112 for this evaluation and setup the core with 1GHz, which has L1 cache and L2 cache whose size is 32KB and 512KB, respectively. While the simulation is to a use small size of memory (512MB DRAM in this work), it does not have a performance impact on our SSD simulations as the benchmark tool bypasses the page cache. The important characteristics of gem5 simulation are shown in Table 1.

gem5 parameters	**value**	**gem5 parameters**	**value**
core	1, atmoic	**kernel**	x86_64-vmlinux-3.4.112
L1D cache	32KB	**entries in NVMe queue**	1024
L1I cache	32KB	**admin submission queue**	1
L2 cache	512KB	**admin completion queue**	1
mem controller	1	**IO submission queue**	1
memory	DDR3, 512MB	**IO completion queue**	1

Table 1: gem5 execution parameters.

PCIe configurations. We evaluate three different versions of PCIe interface (1.x ∼ 3.x), each having a different coding method and bandwidth. For example, PCIe 2.x can offer 5GT/s per lane, but as it uses a 8b/10b encoding method (that has 20% overheads for the PCIe packet movement), the maximum bandwidth per lane can in practice be upto 500 MB/s. The important characteristics we evaluated for PCIe interface is described at Table 2.

PCIe Version	**Raw bandwidth**	**Encoding**	**Real bandwidth**	**Xfer time/byte**
PCIe 1.x	2.5 GT/s	8b/10b	250MB/s	4000ps
PCIe 2.x	5 GT/s	8b/10b	500MB/s	2000ps
PCIe 3.x	8 GT/s	128b/130b	985MB/s	1015.625ps

Table 2: Performance of various versions (generations) of PCIe.

SimpleSSD Configurations				**FIO Configurations**	
Channel	64	**Block/Plane**	512	**ioengine**	posixaio (with direct=1)
Package/Channel	4	**Page/Block**	512	**iodepth**	256
Die/Package	2	**Page size**	4096 Bytes	**bs**	1M
Plane/Die	2	**DMA speed**	400 MHz	**size**	1G

Table 3: Configurations of SimpleSSD and FIO.

SSD and benchmark tool configurations. We setup a high performance SSD that employs 64 channels, each having 4 flash packages. The flash package contains 2 flash dies, each having 2 memory planes. The low-level latency of flash follows up that of a multi-level cell (MLC) flash [19], which is also given by SimpleSSD as a default configuration. All the packages are operated by ONFi 3.x (400MHz) device-level protocol [21]. In addition, we leverage the flexible

I/O tester, called FIO [5] at user-level. For our FIO tests, we configure the application-level I/O depth by 256 and perform POSIX asynchronous I/Os with a 1MB block size for 1GB raw-data. The important characteristics of SimpleSSD and FIO configurations we tested are given by Tables 3.

5.2 Results

Figures 5, 6 and 7 show the bandwidths we measured from various types of storage with different interfaces, such as ideal IDE SSD without the bandwidth restriction (`IDE-ideal`), ideal NVMe SSD (`NVMe-ideal`), and practical SSDs considering the bandwidth restriction of various interfaces (`IDE-SATA3`, `NVMe Gen.1 x1`, `NVMe Gen.2x1`, `NVMe Gen.3x1`, and `NVMe Gen.3x4`). Note that the bandwidth of SATA3 interface is 600MB/s and the bandwidths of PCIe interface are listed in Table 2. In addition, Figures 6, 7, and 5 show the performance of SSDs with 512MB internal cache, 1GB internal cache, and no internal cache, respectively. As shown in the Figure 5, the ideal NVMe (cf. NVMe-ideal) achieves upto 2370 MiB/s and 353 MiB/s in sequential read and sequential write bandwidth, respectively, which exceeds the maximum bandwidth of ideal IDE SSD (cf. IDE-ideal) by 5.5 times and 8.0 times. This means the performance degrade by the IDE interface can be 82% and 88% in sequential read and sequential write, respectively. The performance degrades are observed because the legacy IDE interface does not employ any queue mechanism, which in turn serializes the procedure of I/O requests. Since the internal parallelism of SSD seriously depends on the number of I/O requests which can be served concurrently, NVMe protocol which builds from massive deep queues can utilize SSD's internal resources better and easily outperforms IDE interface.

The bandwidth of interface is another factor that can impact the performance of SSD. Specifically, PCIe Gen.3 x1 can provide 985 MB/s maximal bandwidth, which is 3.9 times faster than PCIe Gen.1 x4. This in turn improves the performance of NVMe SSD with PCIe Gen.3 x1 (cf. NVMe Gen.3 x) by 2.6 times across different types of IO accesses, on average, compared to that of NVMe SSD with PCIe Gen.1 x1 (cf. NVMe Gen.1 x1). In addition, you can observe both `IDE-ideal` and `NVMe-ideal` are capped by the maximum bandwidths of each interfaces.

SSD internal cache also can improve the performance further. Specifically, SSD with 512MB internal cache can improve the performance by 80%, on average, compared to SSD with no cache. As shown in Figure 7, SSD with 1GB internal cache achieves extremely high write throughput, which is even much faster than SSD with 512MB internal cache. This is because we configure FIO to write 1GB data to SSD, which can be perfectly accommodated by 1GB internal cache. Since all write requests can immediately directly write data in internal cache without waiting for the slow NAND flash write operations, Figure 7 illustrates the performance of sequential write outperforms the performance of sequential read by 2.3 times.

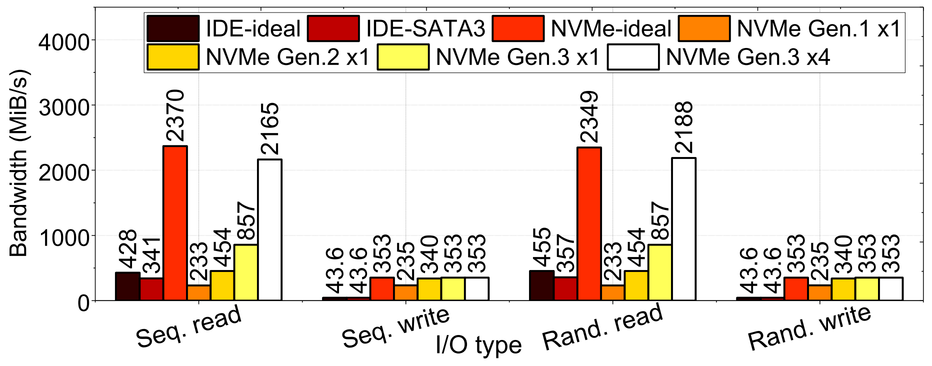


Fig. 5: Bandwidth comparisons between IDE, SATA, and NVMe SSD. No cache exists in the storage device.

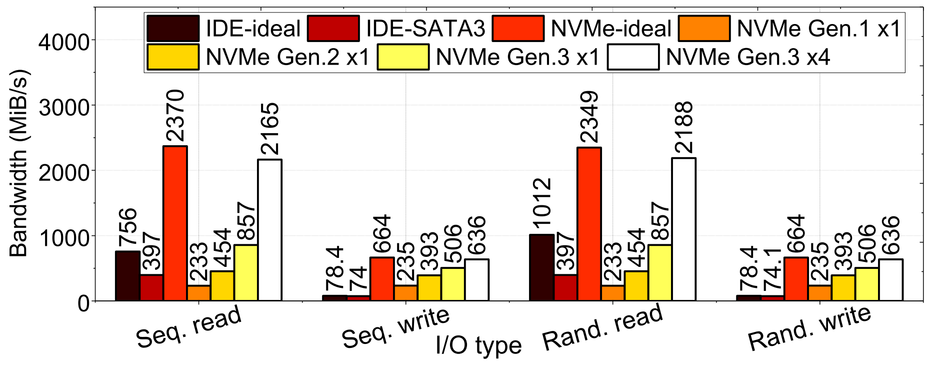


Fig. 6: Bandwidth comparisons between IDE, SATA, and NVMe SSD. Each storage device contains 512MB cache.

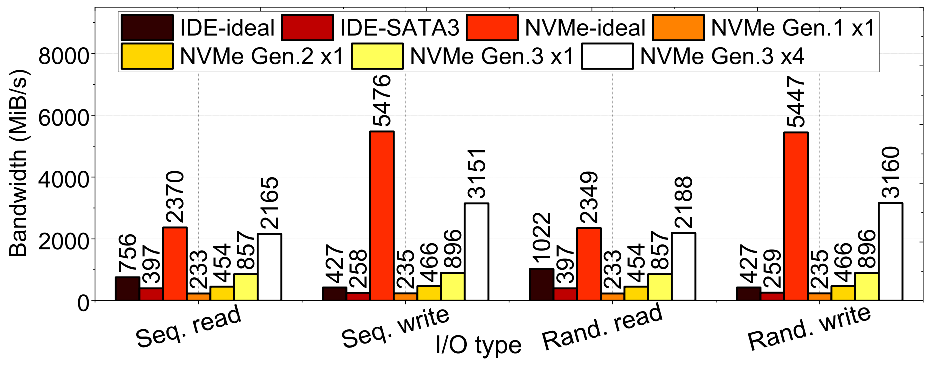


Fig. 7: Bandwidth comparisons between IDE, SATA, and NVMe SSD. Each storage device contains 1GB cache.

6 Conclusion

In this work, we implemented an NVMe controller and disk module to enable a realistic storage stack of next generation interfaces. We integrated the NVMe controller and disk module into gem5 and SimpleSSD for NVMe-enabled full system simulation. We verified the functionality of NVMe interface and software stack that we implemented, using the NVMe command line interface. Our evaluation results revealed that the performance of a high performance SSD can significantly vary based on a different software stack and storage controller even under the same condition of device configuration and degree of parallelism.

References

1. "Nvm express," *NVM Express website*.
2. "Parallel ata," *ATA-ATAPI website*.
3. G. Atwood, "Current and emerging memory technology landscape," *Flash Memory Summit*, 2011.
4. Axboe, "Fio: Flexible i/o tester," 2011.
5. J. Axboe, "Noop scheduler," 2010.
6. P. R. Bashford, "Message signaled interrupt generating device and method," Sep. 30 2003, uS Patent 6,629,179.
7. N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, 2011.
8. F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *HPCA*. IEEE, 2011.
9. B. Dees, "Native command queuing-advanced performance in desktop storage," *IEEE Potentials*, 2005.
10. Y. Huai, "Spin-transfer torque mram (stt-mram): Challenges and prospects," *AAPPS bulletin*, 2008.
11. A. Huffman, "Nvm express, revision 1.2.1," *Intel Corporation*, 2016.
12. S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o," in *ACM SIGOPS Operating Systems Review*. ACM, 2001.
13. M. Jung, "Exploring parallel data access methods in emerging non-volatile memory systems," *TPDS*, 2017.
14. M. Jung *et al.*, "Simplessd: Modeling solid state drives for holistic system simulation," *IEEE CAL*, 2017.
15. M. Jung and M. Kandemir, "Revisiting widely held ssd expectations and rethinking system-level implications," in *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2013.
16. linux nvme, "NVMe management command line interface https://github.com/linux-nvme/nvme-cli," 2017.
17. S.-y. Park, E. Seo *et al.*, "Exploiting internal parallelism of flash-based ssds," *IEEE Computer Architecture Letters*, 2010.
18. Samsung Electronics Co., Ltd., "Samsung ssd pm1725a," 2016.
19. SK Hynix, "NAND Flash Memory H27UBG8T2BTR-BC," 2011.
20. H.-S. P. Wong *et al.*, "Phase change memory," *Proceedings of the IEEE*, 2010.
21. Workgroup, ONFI, "Open nand flash interface specification revision 3.0," *ONFI Workgroup, Published Mar*, 2011.