2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)

Analyzing, Modeling, and Provisioning QoS for NVMe SSDs

Shashank Gugnani, Xiaoyi Lu, Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering
The Ohio State University
{gugnani.2, lu.932, panda.2}@osu.edu

***Abstract*—NVMe-based SSDs are in huge demand for Big Data analytics owing to their extremely low latency and high throughput for both read and write operations. Their inherent parallelism in request processing makes them ideal to be used in virtualized environments, where sharing of resources is a given. Given the shared resource-driven ideology of cloud environments, it is imperative to design middleware which can provide some guarantee of service to applications. In this paper, we show how such QoS can be provided for NVMe SSDs in virtualized environments. Our contributions are threefold: (1) design of accurate NVMe emulation mechanisms in QEMU to provide QoS schemes, (2) theoretical modeling of arbitration mechanisms for assisting in SLA provisioning, and (3) proposing designs in Intel SPDK to seamlessly use the hardware-based QoS provided by NVMe. We provide a complete case for our designs and validate them through thorough experimental evaluation. We show that Deficit Round Robin (DRR) as a hardware-based arbitration scheme is more suited for providing bandwidth guarantees for NVMe SSDs. Our evaluations show that by combining our proposed QoS-aware NVMe emulator in QEMU and enhanced SPDK runtime, we can achieve I/O bandwidth SLA guarantees in an application oblivious manner.**

***Keywords*-QoS, NVMe, Cloud, QEMU, SPDK, SSD**

I. Introduction

The architecture of enterprise clouds is rapidly changing. Novel hardware and software innovations constantly drive the evolution of cloud environments. The Non-Volatile Memory express (NVMe) [1] standard is a recent innovation which has significantly impacted research in storage systems. The standard allows flash storage devices such as SSDs to achieve profound improvements in latency and throughput. NVMe-based SSDs have been emerging as the latest storage technology bridging the dreaded performance gap between hard disks and memory. These new devices are built for extremely low latency and achieving high degrees of parallel I/O. This makes them ideal to be used in cloud environments, where sharing of resources is a given.

In cloud environments, users expect a certain guarantee of service. Considering the new NVMe technology being introduced in enterprise clouds, it is only natural to ask whether a similar guarantee of service can be provided for this emerging hardware. In fact, this issue has been addressed to some extent in the NVMe standard itself. The standard includes provisions to enable request arbitration through mechanisms which are to be provided by hardware. However, there is limited knowledge on using these provisions to enable service guarantees in cloud environments. Prior research [2]–[10] has mostly focussed on software-based provisions for service guarantees. While previous approaches [11], [12] have considered using such hardware provisions to provide some Quality of Service (QoS) to users, their approaches are not holistic. The designs proposed do not provide a complete solution for providing Service Level Agreement (SLA) based guarantees to users. For providing such a solution, there are two key requirements. First, the SLA provisioning should be completely application oblivious, i.e., it should be completely handled by the cloud provider based on the SLA negotiated by the user. Second, there must be mechanisms in place which allow for the provisioning of SLAs without violations. Achieving these requires a holistic approach which we propose in this paper. We show how existing runtimes can be modified for application oblivious QoS provisioning. We also theoretically model the arbitration mechanism available in NVMe and discuss how the model can be used for SLA mapping and provisioning.

NVMe SSDs are still considered as emerging hardware. While costs have been rapidly declining in recent years, they are still high enough to prevent wide-scale adoption in cloud environments [13]–[15]. Having a system which can provide NVMe device emulation can prove to be very useful. Emulation allows for testing NVMe related code without the need for buying expensive hardware. In addition, emulation also allows for approximate performance modeling of such applications. Existing schemes for NVMe emulation do not provide any mechanisms to test and evaluate the arbitration mechanisms available in the standard. Moreover, no flash device has implemented weighted arbitration schemes yet [10]. Thus, we propose designs for accurate modeling of QoS schemes in the NVMe part of Quick Emulator (QEMU) [16], [17]. With this solution, cloud providers can not only verify their middleware and schedulers, but can also use it for performance modeling and benchmarking.

To summarize, the main contributions of this paper are as follows:

- Design of QoS-aware NVMe emulator which provides support for weighted round robin and deficit round robin arbitration
- Theoretical modeling of arbitration schemes with queuing theory
- Extension of Storage Performance Development Kit (SPDK) runtime to allow for application oblivious SLA provisioning

Our evaluations show that by combining our QoS-aware NVMe emulator and enhanced SPDK runtime, we can achieve

This research is supported in part by National Science Foundation grants #CNS-1513120, #IIS-1636846, and #CCF-1822987.

978-1-5386-5504-7/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC.2018.00033

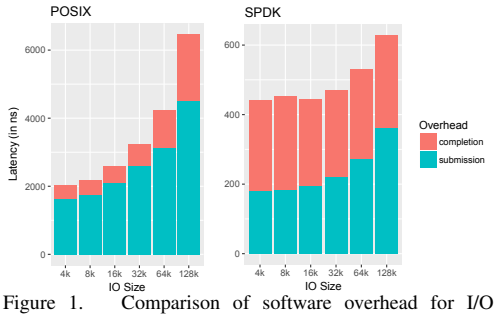

Figure 1. Comparison of software overhead for I/O submission and completion with NVMe SSDs measured using SPDK overhead benchmark. Note the difference in scale between the two graphs.

I/O bandwidth SLA guarantees in an application oblivious manner. We also observe that our QoS-aware emulator can deliver similar or better performance as compared to the existing emulator in QEMU. To the best of our knowledge, our proposed emulator is the first NVMe emulator to offer support for QoS.

The rest of this paper is organized as follows. Section II presents the motivation behind our work, Section III discusses our proposed approach for QoS-aware NVMe emulation in QEMU. Section IV presents a theoretical modeling of arbitration schemes in NVMe, Section V presents a QoS-aware SPDK runtime solution, and Section VI discusses NVMe and Flash specific aspects related to our work. Section VII studies related work and Section VIII concludes the work.

II. Motivation

In this section, we discuss the motivation behind our work.

A. Time for a change

In this subsection, we compare the performance of the Linux NVMe driver and Intel SPDK and discuss the benefits of shifting to SPDK. Before comparing their performance, we briefly introduce SPDK.

Intel SPDK. Intel SPDK [18] is a userspace library built for applications with high-performance storage requirements. SPDK moves all necessary drivers to userspace and operates in polling mode, thereby enabling high-performance access to storage. In addition to legacy storage protocols, SPDK offers support for the NVMe standard, including NVMe over Fabrics [19]. For processing requests over NVMe, the user should create a queue pair (QP), which is a set of submission and completion queues. I/Os for each QP can be submitted and processed in parallel. Synchronization within a QP is left for the user to handle. In general, each application thread requiring I/O is recommended to create a separate QP, allowing maximum parallel processing and eliminating the need for synchronization. Processing of I/O operations is completely asynchronous. Applications need to explicitly ask the SPDK runtime to poll the completion queues. This asynchronous operation allows for complete or partial overlap of I/O and application processing. The SPDK design solves most of the performance related issues that plague the Linux NVMe driver [20].

Comparison of POSIX and SPDK. To determine the best driver to use for NVMe-based applications, we conduct

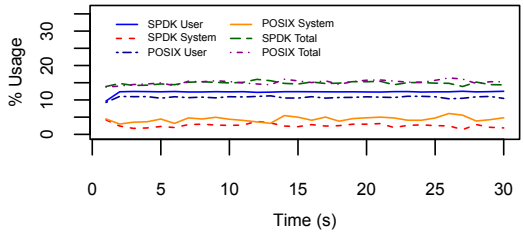

Figure 2. Comparing the CPU Overhead of POSIX and SPDK with NVMe SSDs. POSIX and SPDK achieve nearly the same total CPU usage (top two lines): The high polling overhead of SPDK is offset by the additional software overhead in POSIX.

an experiment to measure the software overhead of request submission and completion for SPDK and POSIX. Our testbed consists of one node with a 8 core, 16 thread sandy bridge CPU with 32GB DRAM and an Intel P3700 NVMe SSD. We use CentOS 7.1 with the 4.9 Linux kernel. We run a random read workload with a queue depth of one (to nullify the effects of queuing delay). Timestamps are collected before and after the submission and completion calls; since the calls are asynchronous, this gives us the software overhead. Figure 1 shows the result of this analysis. We observe that the total software overhead for POSIX is very high compared to SPDK. In fact, for a 4k write, the software overhead is almost 30% of the request processing time (∼2us v/s ∼7us). Next-generation Intel Optane SSDs have even lower latency, further increasing the software overhead [21]. The Linux NVMe driver suffers from some basic flaws [22] including interrupt-based processing and context switches. In terms of latency, SPDK clearly has much better performance (629.1 ns v/s 6465.9 ns overhead for 128k I/O; more than 10x improvement). For both POSIX and SPDK, the submission latency increases with message size. This increase is minimal for SPDK, where the increase in latency can be attributed to gathering the list of pages that hold the request. For POSIX, the increase in latency is significantly higher as it involves a context switch. Completion latency is constant for SPDK (around 260 ns), only involving polling the completion queue. Overall, SPDK latency is negligible compared to processing latency. This implies that the SSD throughput is no longer limited by the software overhead.

We also measure the CPU overhead while running the random read workload, as shown in Figure 2. SPDK has higher user CPU overhead, while POSIX has higher system CPU overhead. SPDK uses polling mode to process I/O completions resulting in increased user CPU usage. POSIX driver is built into the kernel, hence a higher system CPU usage is expected. Interestingly, the total CPU usage for both POSIX and SPDK is almost the same. The high polling overhead in SPDK is offset by the software overhead in POSIX. Thus, while the CPU usage for both is similar, SPDK is able to deliver much better performance.

These experiments point to a clear conclusion; for low latency applications or cloud environments requiring highly parallel access to storage, SPDK should be the obvious choice. While SPDK provides the best performance, it does not offer compliance with the POSIX standard. It offers its own set of low-level APIs for accessing NVMe devices. Applica-

tions need to be modified to enable storage access using SPDK. While application developers are in general reluctant to modify existing production applications, we believe that the performance-related benefits are sufficient to encourage migration from POSIX to SPDK. In fact, many existing applications and frameworks have already embraced this change and moved to SPDK for accessing storage. For example, OpenStack Ceph [23] has recently announced support for SPDK and Facebook's RocksDB [24] key-value store has been modified to run over it. We argue that other applications and frameworks will follow suit and SPDK is the way to go moving forward.

B. QEMU NVMe Emulation

NVMe SSDs are still considered as emerging hardware. Although NVMe SSDs have been commercially available for several years now, their cost is a significant barrier to their adoption in large-scale cloud environments. We believe that over time, as the performance of SSDs increases and their cost decreases, their adoption will see an exponential increase. While it is important to design runtime and middleware that can take advantage of the NVMe standard, it is also important to provide mechanisms to emulate NVMe devices. Emulation allows for testing NVMe related code without the need for buying expensive hardware. In addition, emulation also allows for approximate performance modeling of such applications.

There exist many solutions that provide the ability to emulate NVMe devices. The most robust and stable of these is provided as part of QEMU. QEMU [25] is a popular open-source hypervisor for hardware virtualization. QEMU introduced NVMe emulation support a few years ago which has now developed into a stable tool for NVMe device virtualization. The reason that we chose QEMU for this purpose is that it is a popular and well-known framework for hardware virtualization and it already includes mechanisms to virtualize Memory Mapped I/O (MMIO), block I/O, hardware interrupts (e.g. MSI-X), and PCIe devices. This makes it perfect for emulation of NVMe devices.

The working of the QEMU NVMe emulator is as follows. QEMU has a main thread which is used for processing interrupts. This thread is also used to process NVMe related requests in the emulator. There is a timer interface provided by QEMU which is used by other QEMU subsystems. Timers provide a mechanism to call a given routine (a callback), after a time interval has elapsed. Timers are handled by the main thread. The main thread is essentially a loop which processes interrupts, checks timers, and processes appropriate callback functions as and when timers expire. The NVMe emulator makes heavy use of timers for request submission. Each submission queue that is created has a timer associated with it. The callback function of the timer executes the command submission procedure. The actual command processing is handled by the Linux aio system. On submission queue initialization and upon receipt of any request, the emulator sets a 500 ns timer for that queue. After the timer expires, the main thread runs the callback function and submits requests for the queue.

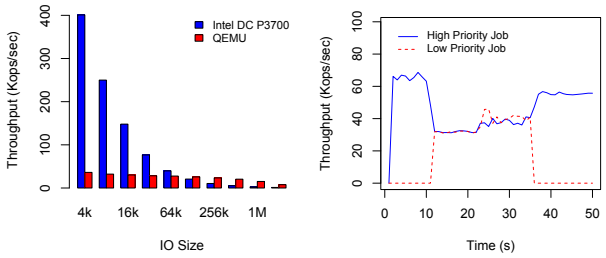

Figure 3. (a) Comparison of random read throughput for Intel P3700 SSD and QEMU NVMe Emulator. Results taken with SPDK fio benchmark with queue depth of 128, (b) Demonstrating the lack of QoS Support in QEMU NVMe Emulator. Results taken with synthetic applications each using a queue depth of 128 and 4KB I/Os.

The arbitration burst setting defines the maximum number of commands that can be submitted in one round from a queue. In case additional requests are pending after request submission, the callback function resets the timer so that the remaining requests can be processed. It is important to note that timers are required here because everything is processed by the main thread. Request completion is also handled by the main thread in the form of a callback function invoked upon completion of the Linux aio. For the actual I/O, QEMU initially writes all data in memory and lazily flushes everything to a backup file located on physical disk. This process is sufficient to emulate the working of an NVMe device. To determine how well QEMU emulates the performance of NVMe hardware, we conduct some performance evaluations.

We first evaluate the performance of the virtualized NVMe device using QEMU and compare it to that of an Intel P3700 SSD. As we can observe from Figure 3(a), the I/O throughput of QEMU is much lower than actual hardware for small I/O sizes. This is because the additional software overhead of device emulation significantly impacts the latency of small I/Os. This is especially true since the latency of small I/O sizes is extremely low. In addition, the P3700 has 18 flash channels to process requests concurrently. In fact, even within a channel, chip, die, and plane level parallelism can be used. This results in notable throughput improvement for small requests. However, large I/O performance is better with QEMU than actual hardware. Since QEMU uses memory for storing data, the high memory bandwidth of main memory leads to better performance for large I/O sizes. We are able to achieve up to 16 GB/s bandwidth for a 1 MB I/O. The difference between QEMU and the SSD is under acceptable levels for large I/O sizes, but not for small sizes. However, the ability to run any NVMe workload makes QEMU a viable and useful emulation tool for NVMe. We do not focus on improving the emulator to accurately model the *performance* characteristics of NVMe devices, but rather on accurate modeling of the *command processing* and *arbitration* mechanisms. This will allow us to provision service guarantees using the improved QEMU emulator.

C. The need for QoS-aware emulation and runtime

Quality of Service (QoS) is an extremely important part of the cloud computing paradigm. In fact, this is one of the

primary reasons for the popularity of cloud computing. Most cloud providers these days offer Service Level Agreements (SLAs) to their clients as a basic way of achieving QoS. In the context of NVMe storage, it is paramount to have a design that provides some guarantee of service (e.g., latency or bandwidth) to applications or VMs depending on the service granularity. NVMe provides hardware-based mechanisms for command arbitration. Schemes like round robin and weighted round robin arbitration (WRR) (see Section III-A) have been included in the NVMe standard, while NVMe SSD vendors are free to implement vendor specific arbitration mechanisms. However, none of these schemes have been implemented in commercially available SSDs.

While the QEMU NVMe emulator supports the entire NVMe command set, the hardware-based weighted round robin arbitration mechanism specified in the NVMe standard is not supported. Cloud providers providing service guarantees for storage might want to test their scheduling solutions using hardware virtualization. In this context, having an emulator for NVMe devices which can provide command arbitration mechanisms as described in the standard can prove to be extremely useful. We believe that making such a solution available will tremendously benefit cloud providers in designing scheduling solutions. To demonstrate the importance of QoS, we conduct a simple experiment. We run two separate applications using SPDK over the QEMU NVMe emulator, one having high priority and the other low priority. We enable the WRR scheme in SPDK to simulate a QoS scenario. These applications use the high priority and low priority submission queues, respectively. Both of these applications submit back-to-back 4k I/O requests. In the beginning, only the high-priority job is running. At the 10^{th} second, the low priority job is run. Figure 3(b) shows the result of this experiment. It can be observed that the low priority job significantly impacts the performance of the high priority job. In fact, both jobs experience the same I/O throughput, thus confirming that QEMU does not provide any support for WRR arbitration mechanism. For testing and evaluating any scenario which expects a guarantee of service for storage, this emulator will fall short of expectations. To address these issues, we propose a QoS-aware NVMe emulator, the details of which are presented in the next section.

D. Summary

So far, we have made the following observations:

1) SPDK provides the best performance for NVMe devices
2) QoS is an important feature in cloud environments
3) The existing emulator in QEMU does not provide any kind of service guarantees

These observations lead us to conclude that there is a need for QoS-aware emulation and runtime. The SPDK runtime provides some mechanisms to exploit the WRR arbitration scheme in NVMe. However, the configuration of this scheme is left to the discretion of the user. Thus, SLA provisioning cannot be application oblivious. We envision a cloud environment where SLAs can be met in an application oblivious manner to provide expected features to users. Clearly, existing emulation and runtime schemes fail to satisfy this vision. In this context, we propose a QoS-aware NVMe emulator over QEMU and a QoS-aware SPDK runtime. Our goal is to design solutions which can provide I/O service guarantees in NVMe clouds. Sections III and V provide additional details about these designs.

III. QOS-aware NVMe Emulation

In this section, we describe our proposed design for QoS-aware NVMe emulation. We first briefly describe the Weighted Round Robbin and Deficit Round Robbin (DRR) arbitration scheme, then present our solution for implementing these schemes in QEMU, and finally demonstrate via experimental evaluation the superiority of our solution over the existing NVMe emulation in QEMU.

A. WRR arbitration

The WRR arbitration mechanism in the NVMe standard provides a useful mechanism to implement QoS support in cloud middleware. The biggest advantage of this mechanism is that it is implemented completely in hardware resulting in low latency arbitration and reduced complexity of drivers and runtimes. This mechanism works as follows. There are three different priority classes for NVMe submission queues, high, medium, and low. Each class of priority is assigned a numerical weight. The NVMe controller processes commands for submission queues in order of their priorities. The maximum number of commands that can be processed for queues of a certain priority in one arbitration round is determined by the weight of that priority class. For a single submission queue, the maximum number of commands that can be processed in one arbitration round is determined by the arbitration burst setting. By adjusting the weights of different classes of priority, the desired level of QoS can be achieved. The current NVMe emulator in QEMU does not provide support for the WRR arbitration mechanism. In this section, we describe our proposed design for a QoS-aware NVMe emulator.

B. DRR arbitration

While the WRR scheme is efficient in providing a guarantee of throughput, it requires the sizes of requests to be fixed or previously known. Otherwise, applications with different sized requests will result in a higher than intended weight for large requests. In this context, significant research has been done to provide solutions which can provide optimal bandwidth QoS despite request size variation. Schemes like deficit round robin (DRR) and weighted fair queuing (WFQ) are popular models widely used in networking. Both DRR [26] and WFQ [27] can provide bandwidth guarantees. However, WFQ requires $O(log(n))$ time to process each request, while DRR only requires $O(1)$, where n is the number of priority classes. Even though there are just three priority classes (as defined in the NVMe standard), DRR is simpler and satisfies our requirements for QoS. DRR is a modification of WRR, where instead of giving each request equal cost, its given a cost equal to its size. This effectively ensures that the overall bandwidths for each priority class are in the ratio of their weights. We

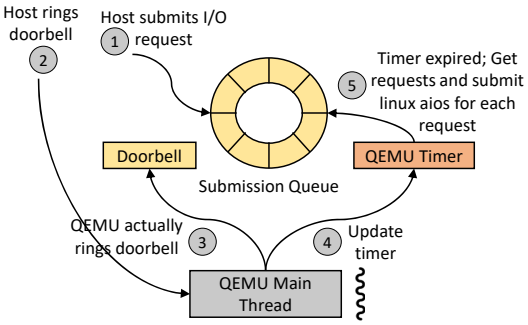

Figure 4. Command submission procedure for QEMU NVMe emulation

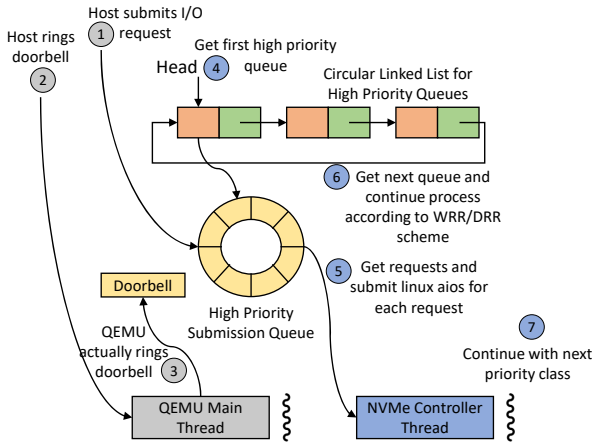

Figure 5. Proposed command submission procedure for QoS-aware NVMe emulation

believe that DRR is an effective alternative to WRR which can provide better QoS with the same overhead. We implement the DRR scheme in the QEMU NVMe emulator and show that it is much more effective in providing bandwidth guarantees to cloud users than WRR.

C. Designing arbitration schemes in QEMU

For basic NVMe emulation, the existing designs in QEMU can be re-used. However, for providing QoS, the software-based arbitration mechanism in QEMU needs to be redesigned.

Existing emulation design. Figure 4 shows the command submission process for the existing emulator. The working can be detailed by the following steps: ① The host puts an I/O request in the submission queue, ② The host rings the queue doorbell, ③ This triggers an interrupt which is processed by the QEMU main thread, ④ The QEMU main thread updates the timer of the submission queue, and ⑤ The timer expires and the main thread submits Linux aios for each request. The default design uses the QEMU main thread to execute the entire I/O processing pipeline. The fundamental flaw with this approach is that as soon as a command is placed in the submission queue, the main thread will start processing it. Submitting an I/O request involves ringing a doorbell which generates an interrupt. This interrupt is also handled by the QEMU main thread. Thus, at one time, only one command can be in any submission queue. This design allows no conception of QoS and is fundamentally different from the way commands are processed in an actual NVMe SSD.

Design Considerations. Emulation is an effective tool for testing the performance and functionality of hardware. A good emulation requires accurate modeling of hardware, i.e., the software implementation should mimic the behavior of hardware. In addition, specially while proposing new hardware logic, the functionality, complexity, and memory usage should be kept as minimal as possible. This allows for cost savings and latency benefits. In this context, we emulate arbitration schemes using simplistic space efficient data structures. Each priority class is assigned a circular linked list for holding pointers to the submission queues in the priority class. In addition, 8 bit unsigned integers are used to store remaining and pre-defined weights for each priority class which will map to hardware registers or dedicated buffers on device memory for each queue. After each arbitration round, the remaining weights are updated to the pre-defined weights. In addition, a single bit is kept for each submission queue to indicate if it has a pending request. This allows for efficient arbitration by allowing the controller to quickly skip over empty queues. We believe that this design closely mimics hardware behavior. We now describe our arbitration scheme designs in QEMU.

Proposed Design. For emulating the NVMe device in a more realistic manner, we introduce a dedicated thread for executing the functions of the NVMe controller. We also make sure that data structures shared with the QEMU main thread are locked using a mutex before access. In addition, to allow for software-based arbitration, we introduce a circular linked list for each priority class. Each submission queue is added to the appropriate linked list when it is created. We maintain just one head pointer for each linked list which points to the submission queue that should be used next by the controller for command arbitration. The controller processes commands from each priority class one by one. For each priority class, the controller will use the appropriate linked list and start processing commands for the queue pointed to by the head pointer, moving the pointer each time a queue is serviced. It will continue processing commands for the priority class until either all queues have been serviced or the total cost of commands processed is equal to the weight of the priority class. Thus in one arbitration round, all priority classes will be served, but the maximum cost of commands that can be processed for each class is equal to its weight.

Figure 5 shows the command submission process for the proposed emulator. The following steps describe the complete process: ① The host submits requests directly to the submission queues, ② The host then rings the doorbell, ③ This generates an interrupt which is handled by the QEMU main thread, ④ The NVMe controller thread first uses the high priority linked list, ⑤ It picks up each submission queue with an outstanding request and submits Linux aios for requests, ⑥ Controller thread moves onto next submission queue, and ⑦ After processing requests for all submission queues in a priority level, the controller moves to next priority linked list. The NVMe controller thread in parallel continuously scans the circular linked lists for each priority class. Whenever it finds pending requests, it submits a Linux aio operation

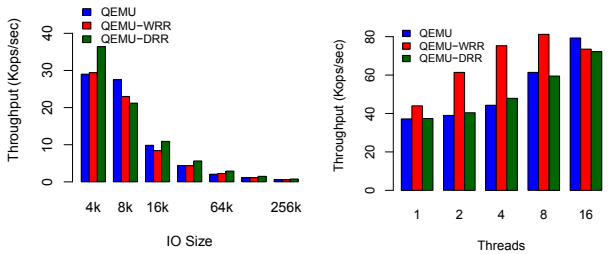

Figure 6. (a) Comparison of random write throughput for QEMU NVMe Emulation, (b) Comparison of random read throughput for QEMU NVMe Emulation. Results for both taken with SPDK fio benchmark with queue depth of 128 and 4k I/O Size.

for each request while ensuring WRR/DRR arbitration. Our NVMe controller thread can execute independently of the QEMU main thread. This allows the request submission and processing to proceed in parallel, thereby allowing for the possibility of multiple outstanding requests in submission queues. This emulates the actual behavior of an NVMe SSD more accurately. Applications submit and place a request in the submission queues independent of the processing of requests by the NVMe hardware. Thus, our solution provides a better emulation of NVMe hardware.

There are many possible solutions for designing arbitration in QEMU. For example, the QEMU aio interface can be used to process commands for each submission queue parallelly. We chose our design with the goal of emulating the NVMe device behavior as accurately as possible with minimal overhead. In our design, we use just one additional thread, but the arbitration mechanism and NVMe controller are precisely emulated. Our experimental analysis also confirms this claim.

D. Comparison with existing emulator

We now present some results comparing the performance of our proposed solution with QEMU. Figure 6(a) shows a comparison of the write throughput obtained from the fio benchmark. We observe some interesting trends here. In general, the WRR and DRR schemes show better or similar performance than the default emulator. Since we introduced an additional thread dedicated to processing NVMe requests, throughput for most I/O sizes is improved. However, for 8k and 16k sizes, the overheads of threading, data transfer between cores, and software-based arbitration lead to slightly lower performance than QEMU. We conclude that our solution is similar to QEMU in emulating the performance of NVMe SSDs.

We also evaluate the performance of our solution using a multi-client benchmark. Each client uses a separate submission and completion queue for request processing. This design allows for completely lockless and synchronization free I/O processing, leading to good scalability. Figure 6(b) shows the read throughput for different number of client threads. The host node has 16 threads, thus we go up to 16 client threads. We observe a constant increase in the throughput with increasing client threads for the default emulator. DRR throughput is lower than WRR for all cases. We find that DRR needs additional arbitration rounds for processing requests since the cost of each request is higher than WRR. For our emulator, the throughput is always higher than the default. However, for 8 and 16 client threads we notice a slight decrease in throughput. We introduce a separate thread to emulate the NVMe controller which competes with client threads for resources when simulating a large number of clients. The decrease in throughput is minimal and we believe that a scenario where continuous I/Os are being submitted from each core will be rare.

IV. Performance Modeling of Arbitration Schemes

In this section, we model the performance of the WRR and DRR arbitration schemes.

A. Performance modeling

In the previous section, we presented our QoS-aware NVMe emulator design. To allow cloud providers to fully utilize this new tool, it is necessary to provide a model for performance prediction. This will allow for accurate SLA provisioning with minimal violations.

Symbol	Description
$h/m/l$	high/medium/low priority class weight
λ	input request rate
γ	average SSD throughput
μ	SSD processing rate
ρ	queue utilization
p	SSD parallelism
q	queue depth
a	average request latency

Table I
Symbols used and their descriptions

We first model the WRR throughput. The presence of many varying situations and variables makes the performance modeling of WRR challenging. For simplicity, we assume that the average I/O size for each priority class is the same. For each priority class, the maximum commands processed in one round will be equal to its weight. However, in case the submission queues in a priority class do not have more commands than its weight, other priority classes will be able to get higher throughput. Thus, the minimum throughput for each priority class should be in the ratio of their weights. Assuming that γ is the average throughput the NVMe SSD can deliver and h, m, and l are the weights of the corresponding priority classes, the minimum throughput for the high priority class can be calculated out as

$$\gamma_h \geq \frac{h}{h+m+l} \times \gamma \tag{1}$$

Minimum throughput for other priority classes can be calculated similarly. For estimating the actual throughput of each priority class, queuing theory can be used. While each priority class can have multiple queues, they can be considered to constitute one combined big queue. If λ_h, λ_m, and λ_l are the input request rates for the respective priority classes, we know from queuing theory that $\gamma = \lambda$, i.e. output rate is equal to input rate if input rate is not greater than the processing rate μ. This is generally true for NVMe since the command submission will fail if the submission queue is full. The input

rate will then automatically adjust to be equal to the output rate. For a general case, the weights in Equation 1 need to be multiplied by the utilization (ρ) of each priority class so that we can determine the average commands processed from each class in one arbitration round. The utilization of each class can be computed as $\frac{\lambda}{\gamma}$ using queuing theory. Thus, the average throughput for each class can be calculated as

$$\gamma_h = min(\lambda_h, \frac{h}{h + \rho_m m + \rho_l l} \times \gamma) \quad (2)$$

It is easy to see that if we do not submit any requests in the medium and low priority classes, i.e ρ_m and ρ_l are zero, then γ_h will be γ or the complete throughput of the SSD as long as the input rate λ_h is high enough to keep the SSD busy. The utilization of each priority class (ρ) is the ratio of its input rate to max throughput. Utilization determines how many requests are available to be processed for the priority class in one arbitration round. A utilization of one means that the number of requests available for processing is equal to the weight of the priority class. For DRR, replacing γ with the average SSD bandwidth should suffice since it is bandwidth based. In addition, no guarantees on the request size are required.

To calculate the latency for each operation of a priority class, we need to account for two factors. First, queuing delay and second, internal parallelism in the SSD. Queuing delay accounts for the time a request waits to be serviced by the SSD. In general, a request in a particular priority class must wait for requests before it in its own class as well as requests in other priority classes to finish. Assuming requests take an average of time of a to complete, p requests can be submitted in parallel, and each priority class can have a maximum of q outstanding requests, we can calculate the maximum latency for the high priority queue as

$$t_{max} = \frac{q}{h} \times \frac{h + \rho_m m + \rho_l l}{p} \times a \quad (3)$$

This equation should hold true for both DRR and WRR as long as the average request size for all classes is the same. Otherwise, a will not remain a constant. The parallelism p of the SSD can approximately be estimated based on the number of flash chips it has. This is assuming that requests are uniformly distributed over the logical address space. For workloads with different request distributions, conflicts between requests will likely reduce the parallelism. Write workloads will be adversely affected by garbage collection activities further reducing the parallelism. Assuming no conflicts between requests, the effective parallelism can be derived by dividing the number of flash channels by the write amplification factor.

B. Model Validation

To prove the validity of our proposed model, we run experiments on an NVMe SSD to confirm the latency and throughput characteristics under different load conditions. We use the same testbed as described in Section II for our experiments. However, as mentioned before, there does not exist an actual flash device offering either WRR or DRR arbitration in hardware. Hence, we build a thin software layer over SPDK, providing software-based queues offering DRR and WRR schemes. This layer is designed to be light weight with minimal locking to ensure hardware-like performance. To validate our model, we run a random read benchmark with three threads using the three separate priority classes. DRR weights are set to (32k, 64k, 128k) and each thread submits 32k I/O requests. Each priority class is assigned a separate hardware queue, although, request submission and completion is handled by a single thread to ensure DRR compliance. We vary the utilization of the low priority thread and measure the throughput and latency of the high priority thread. For our model, we estimate γ and a/p through empirical analysis. Both of these parameters can be estimated by running a simple multi-client benchmark with one thread per physical core utilizing a separate hardware NVMe queue. γ was found to be around 240k while $\frac{a}{p}$ was around 81 μs. We compare these results with those predicted by our model. This comparison is presented in Figure 7. Figure 7(a) shows the high priority throughput with varying low priority utilization (ρ_l). ρ_h and ρ_m are set to one. Figure 7(b) shows the latency of the high priority class with varying low priority utilization. We observe near perfect correlation between the observed and predicted values with a maximum deviation of less than 5%. Thus, we believe that our model works well in practical situations and is a useful tool for performance prediction. Given the estimated values of γ, a, and p, we can accurately predict the latency and throughput of any priority class.

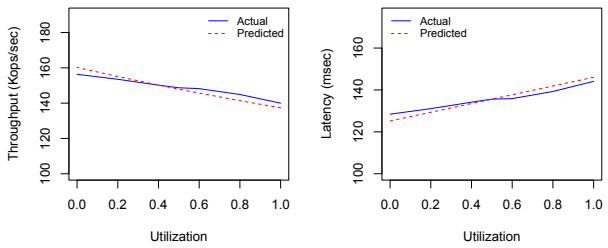

Figure 7. Model Validation: actual and predicted high priority class throughput and latency with varying low priority queue utilization (ρ_l)

C. Model usage scenarios

Calculating the average throughput and maximum latency for each priority class can prove to be useful in multiple scenarios while provisioning I/O bandwidth and latency SLAs in cloud environments. Consider a scenario where some job is already using an NVMe device and the cloud resource scheduler would like to schedule another job to use the same device. By using Equation 2, the scheduler can calculate the I/O bandwidth for both jobs and determine whether their respective SLAs will be violated. A decision can then be made about scheduling the new job to use the NVMe device. Now consider another scenario where the scheduler would like to assign weights to each priority class. Equations 2 and 3 can be used to calculate the weights, assuming that the bandwidth and latency SLAs and utilization of each job are known beforehand.

We have shown in this section that the performance modeling of arbitration schemes in NVMe can serve to be extremely useful in cloud environments.

V. QoS-aware SPDK Runtime

In this section, we describe our proposed approach for designing a QoS-aware SPDK runtime. We start by presenting our solution for enabling service guarantees in NVMe-based cloud environments, before moving on to our application oblivious design for QoS provisioning using SPDK. Finally, we present evaluation results with synthetic application scenarios to demonstrate the benefits of our design.

A. Enabling service guarantees

In modern cloud environments, users expect a guarantee of service for their applications. Cloud providers typically negotiate SLAs with users as a way to provide these guarantees. In such a scenario, cloud middleware and runtime should be able to provide mechanisms to satisfy these guarantees as QoS. From an end user's perspective, the SLA provisioning should be completely transparent. So, the service guarantee mechanisms should be completely application oblivious. In the context of NVMe storage, our goal is to use the hardware provided arbitration mechanisms as a way of providing applications with a guarantee of I/O bandwidth. Since these schemes should be application oblivious, we propose to modify the (SPDK) runtime to allow for QoS support.

A cloud environment typically charges users based on the level of priority they desire. Since, the NVMe WRR arbitration scheme allows for three priority classes as discussed before, we propose the same priority classes for application users as well. The corresponding priority classes will be mapped to each other such that an application with high priority will submit I/O requests to the high priority submission queue and so on. The DRR scheme is also based on the same three-priority system. The weight of each priority class, which determines the maximum number of I/O commands that can be processed from one class in an arbitration round, can be determined by the cloud provider based on the SLA negotiated with the user. Similarly, for scheduling multiple users to share an NVMe SSD, the I/O priority requested by each user and the priority class weights need to be considered.

B. Application oblivious QoS provisioning

The SPDK runtime has support for the NVMe WRR scheme. This is however left to the discretion of the user himself. To use the WRR scheme, the user has to explicitly enable it using an SPDK function and set the priority for each submission queue created. This existing mechanism does not satisfy our application oblivious requirements. We thus propose a new priority mapping design in SPDK which does not require application changes to modify priority. To this end, we propose to use the Linux I/O priority framework as a means to transfer the priority class information from the application to the SPDK runtime, similar to the approach proposed in [11].

The I/O priority class for an application can be set using the *ionice* command which expects a value from 0 to 3. We use the 1-3 classes and map them to high, medium, and low priority classes. 0 is mapped to the urgent priority class and is left for cloud administrative purposes. The priority class for the application can then be obtained by SPDK using the Linux I/O priority interface (`ioprio_get` syscall). This design works because the SPDK runtime will be run in the context of the thread submitting I/O. We modify the SPDK runtime to always use the WRR or DRR scheme and set the priority for a QP based on the I/O priority of the application it is associated to. The I/O priority of each application can be set by the cloud middleware based on the SLA with each user. In this manner, any application built using SPDK can be provided any level of service without the need to modify the application.

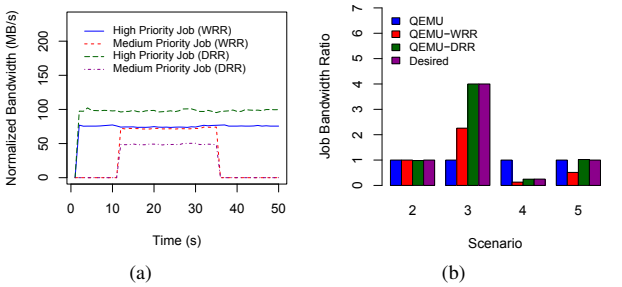

Figure 8. Evaluation with Synthetic Application Scenarios: (a) Bandwidth over time with Scenario1, (b) Job bandwidth ratio for Scenarios 2-5

C. Handling rogue tenants

Our QoS-aware runtime provides tenants with an interface to provide a certain guarantee of service. In this regard, it is important to ensure that a rogue tenant cannot influence the performance of others. In our design, each tenant is allocated a separate QP for I/O request processing, resulting in performance isolation. The actual request processing is done in the QEMU emulator which ensures that the service guarantees of each priority class are maintained. A rogue tenant might submit requests at a rate higher that its service guarantee. This will only result in its submission queue getting filled up, and eventually it will be unable to submit requests. This will result in its throughput getting limited to its service guarantee. Thus, QoS will be ensured regardless of how the tenants submit requests.

D. Synthetic application scenarios

We use the same testbed as described in Section II for our evaluations. However, since our SSD does not support hardware-based arbitration, we instead use our QEMU-based emulator. We use a modified version of QEMU [17] built for Open-Channel SSDs and SPDK v17.10 for our evaluations and as a base for our designs.

To show the benefits of a QoS-aware SPDK runtime, we simulate five application scenarios and measure the bandwidth achieved by each job over time. For all scenarios, we set the priority class weights to (32, 16, 8) for WRR and (128k, 64k, 32k) for DRR, ensuring that the weight ratios are the same. For the first scenario, we use an experiment similar to the one in Section II-B. In this scenario, we run one high priority job with 4k requests and one medium priority job with 8k requests. Figure 8(a) shows the results of this experiment. With the WRR, both jobs receive the same bandwidth despite having different

priorities. Each request is given equal priority regardless of its size, leading to a skewed bandwidth distribution. However, DRR is able to achieve near perfect bandwidth distribution as per priority weights. We also note that the bandwidth over time is relatively stable pointing to a robust hardware emulation.

The remaining scenarios all have 2 simultaneous jobs submitting back-to-back requests. Scenario 2 has two high priority jobs, both with 4k requests. Scenario 3 has a high priority job with 4k requests and a low priority job with 8k requests. Scenario 4 is the same as Scenario 3 with the priorities exchanged. Scenario 5 has two high priority jobs, one submitting 4k and 8k requests and the other 8k and 16k requests. We measure the total average bandwidth for both jobs in each scenario and calculate the bandwidth ratio. This analysis is presented in Figure 8(b). We also provide the expected ratio as per the priority weights. We are more interested in the bandwidth ratios rather than their actual values since we are focused on QoS and the hardware performance is only emulated. In all scenarios, the difference in request sizes leads WRR to incorrectly favor the job with larger request size, while DRR is able to achieve close to the expected ratio. The only case where WRR provides the desired ratio is Scenario 2, where the request sizes for both jobs are the same. Default QEMU just provides equal throughput distribution, not providing any service guarantees whatsoever. This analysis clearly demonstrated the superiority of DRR over WRR in achieving bandwidth guarantees.

Although our results are based on NVMe emulation, we expect similar behavior with actual hardware. There are two reasons for this expectation. One, both WRR and DRR have been shown to be easy to implement in hardware [26], [28] and provide accurate bandwidth ratios. Two, usage of separate hardware queues for each application along with lockless request submission and completion paths ensure that the hardware performance characteristics are reflected in the application performance. **We thus believe that DRR should either become part of the NVMe standard or be accepted by vendors as a good implementation choice for vendor specific arbitration schemes.**

VI. Discussion

In this section, we discuss aspects of NVMe and Flash devices that are particularly relevant to our work.

Performance: The performance of NVMe devices in virtualized environments is a source of concern. The NVMe standard provides Single Root I/O Virtualization (SR-IOV) [29] for NVMe device virtualization. With SR-IOV, the NVMe device can be presented to each VM as a separate physical device. The VM can directly access the device without hypervisor intervention. Studies [12], [30], [31] have shown than SR-IOV performance is close to native. Cloud providers should provide support for SR-IOV-based NVMe virtualization to allow for maximum performance. For the software stack, SPDK offers significantly lower latency and higher throughput for I/O requests as compared to POSIX. Of course, not all applications are written using SPDK or can be easily modified to use it.

Isolation: The NVMe standard offers the ability to create namespaces as a way to achieve logical isolation. Namespaces are a set of logical blocks in the hardware. Each namespace contains a distinct continuous set of logical blocks and can be addressed using a namespace id. This design does not provide any guarantee of performance isolation since the namespaces are logical and the physical blocks used to store the data are common for all namespaces. Separate namespaces offer complete isolation only in terms of security. In a cloud environment, each VM can be assigned a separate namespace for security isolation, however performance isolation is still a matter of concern. Recent work [12] has shown that modifying the Flash Translation Layer (FTL) to map namespaces to separate physical blocks can show significant improvement in I/O performance in virtualized environments.

QoS: We find that DRR is better suited for achieving bandwidth QoS. According to the NVMe specification, NVMe vendors can implement their own arbitration schemes. We believe that NVMe vendors should implement DRR for enterprise Flash devices to provide an efficient mechanism for hardware-based QoS. Flash devices suffer from the write amplification effect [32], [33] which leads to read/write interference. This can disrupt the performance of DRR when workloads with mixed read and write requests are executed simultaneously. We have ignored the effects of write amplification in this paper. This remains an interesting avenue for future research.

VII. Related Work

There has been a lot of research in emulating PCIe devices. In particular, the rising interest in NVMe has led to the development of several emulators which provide some basic NVMe functionality. QEMU [25] provides support for NVMe emulation which has been discussed at length in this paper. In addition, VirtualBox also provides basic NVMe emulation. Several other solutions [34]–[40] are also available. However, none of these emulators provide mechanisms to exploit the arbitration mechanisms provided by the NVMe standard. Our solution is thus unique in providing this support.

Several prior works have studied QoS support over shared cloud and data-center storage systems [3]–[7]. Particularly for Flash-based storage, specific cost model based I/O schedulers such as FIOS [8] and FlashFQ [9] designed for fairness and throughput guarantees were proposed. On the other hand, providing QoS-aware runtimes for NVMe devices has been a topic of recent research. Joshi et al. [11] propose to implement WRR support for NVMe in the Linux driver. They employ a similar I/O priority-based approach for application oblivious QoS provisioning. However, their design suffers from two fundamental flaws. First, they implement their design in the Linux driver which we have shown to perform poorly as compared to SPDK. Second, they provide no mechanism for cloud providers to provision SLAs using their solution. We argue that our solution is more applicable in terms of performance and usability in cloud environments.

With the availability of fast network interconnects, storage disaggregation-based technologies and systems are being

extensively explored [41], [42]. Along these lines, software-based systems for accessing remote NVMe Flash at latencies as low as local NVMe access, such as ReFlex [10], have been proposed. Open-source disaggregated I/O architectures, such as Crail [42], are built exclusively with user-level I/O support (e.g., NVMeoF, SPDK, RDMA), allowing heterogeneous storage and networking hardware to interact with each other in an optimal manner within the data processing engine.

VIII. Conclusion

In this paper, we first provided results demonstrating the clear superiority of SPDK. Next, we evaluated the existing QEMU-based NVMe emulator using performance and QoS as our metrics. We concluded that the emulator is insufficient for providing any service guarantees. We then proposed designs for accurate modeling of hardware-based WRR and DRR in the QEMU NVMe emulator. We theoretically modeled the arbitration schemes and showed how the analysis can be used for SLA provisioning in cloud environments. Finally, we discussed a new approach for providing service guarantees using a QoS-aware SPDK runtime. We demonstrated through experimental evaluation that our QoS-aware NVMe emulator with DRR scheme and SPDK runtime can deliver bandwidth service guarantees in cloud environments in an application oblivious manner. This paper should prove useful to vendors while designing arbitration schemes in SSDs and to cloud providers in provisioning SLAs for tenants. In the future, we plan to explore more along the communication perspective through integration with RDMA and NVMe over Fabrics.

References

[1] NVMe Express Inc., "NVM Express," http://nvmexpress.org/, 2017.

[2] S. Ahn, K. La, and J. Kim, "Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems," in *HotStorage'16*.

[3] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, "Storage Performance Virtualization via Throughput and Latency Control," *ACM Transactions on Storage (TOS)*, vol. 2, no. 3, pp. 283–308, 2006.

[4] D. Shue, M. J. Freedman, and A. Shaikh, "Performance Isolation and Fairness for Multi-Tenant Cloud Storage," in *OSDI'12*, pp. 349–362.

[5] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin, "Maestro: Quality-of-Service in Large Disk Arrays," in *ICAC'11*, pp. 245–254.

[6] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *FAST'09*.

[7] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," in *OSDI'10*, pp. 437–450.

[8] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," in *FAST'12*, p. 13.

[9] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-based SSDs," in *USENIX ATC'13*, pp. 67–78.

[10] A. Klimovic, H. Litz, and C. Kozyrakis, "ReFlex: Remote Flash Local Flash," in *ASPLOS'17*, pp. 345–359.

[11] K. Joshi, K. Yadav, and P. Choudhary, "Enabling NVMe WRR support in Linux Block Layer," in *HotStorage'17*.

[12] X. Song, J. Yang, and H. Chen, "Architecting Flash-based Solid-State Drive for High-performance I/O Virtualization," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 61–64, 2014.

[13] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating Server Storage to SSDs: Analysis of Tradeoffs," in *EuroSys'09*, pp. 145–158.

[14] Z. Yang, M. Awasthi, M. Ghosh, and N. Mi, "A Fresh Perspective on Total Cost of Ownership Models for Flash Storage in Datacenters," in *CloudCom'16*, pp. 245–252.

[15] L. M. Grupp, J. D. Davis, and S. Swanson, "The Bleak Future of NAND Flash Memory," in *FAST'12*, pp. 2–2.

[16] F. Bellard, "QEMU, A Fast and Portable Dynamic Translator," in *USENIX ATC'05*, vol. 41, p. 46.

[17] OpenChannelSSD, "QEMU for Open-Channel SSDs," https://github.com/OpenChannelSSD/qemu-nvme, 2017.

[18] Intel, "Intel SPDK," www.spdk.io/, 2017.

[19] NVMe Express, "NVMe over Fabrics," http://www.nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf, 2016.

[20] Z. Yang, L. E. Paul, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, and V. Verma, "SPDK: A Development Kit to Build High Performance Storage Applications," in *CloudCom'17*, pp. 154–161.

[21] A. Foong and F. Hady, "Storage as Fast as Rest of the System," in *2016 IEEE 8th International Memory Workshop (IMW)*. IEEE, 2016, pp. 1–4.

[22] Z. An, Z. Zhang, Q. Li, J. Xing, H. Du, Z. Wang, Z. Huo, and J. Ma, "Optimizing the Datapath for Key-value Middleware with NVMe SSDs over RDMA Interconnects," in *Cluster'17*, pp. 582–586.

[23] Red Hat, Inc., "Ceph," http://ceph.com/, 2017.

[24] Facebook, "RocksDB," http://rocksdb.org/, 2015.

[25] QEMU, "QEMU: the FAST! Processor Emulator," https://www.qemu.org/, 2017.

[26] M. Shreedhar and G. Varghese, "Efficient Fair Queuing using Deficit Round-Robin," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, 1996.

[27] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4. ACM, 1989, pp. 1–12.

[28] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip," *IEEE Journal on selected Areas in Communications*, vol. 9, no. 8, pp. 1265–1279, 1991.

[29] PCI-SIG, "PCI-SIG Single-Root I/O Virtualization Specification," http://www.pcisig.com/specifications/iov/, 2016.

[30] J. Jose, M. Li, X. Lu, K. C. Kandalla, M. D. Arnold, and D. K. Panda, "SR-IOV Support for Virtualization on InfiniBand Clusters: Early Experience," in *CCGrid'13*, pp. 385–392.

[31] J. Liu, "Evaluating Standard-based Self-virtualizing Devices: A Performance Study on 10 GbE NICs with SR-IOV Support," in *IPDPS'10*, pp. 1–12.

[32] P. Desnoyers, "Analytic Modeling of SSD Write Performance," in *SYSTOR'12*, p. 12.

[33] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-based Solid State Drives," in *SYSTOR'09*, p. 10.

[34] M. AbdElSalam, "NVMe Solid State Drive Verification Solution using HW Emulation and Virtual Device Technologies," in *11th International Design & Test Symposium (IDT)*, 2016, pp. 47–52.

[35] K. T. Malladi, M. Awasthi, and H. Zheng, "FlexDrive: A Framework to Explore NVMe Storage Solutions," in *HPCC'16*, pp. 1115–1122.

[36] ——, "Software-Defined Emulation Infrastructure for High Speed Storage," in *SYSTOR'16*, p. 22.

[37] L. Zuolo, C. Zambelli, R. Micheloni, M. Indaco, S. Di Carlo, P. Prinetto, D. Bertozzi, and P. Olivo, "SSDExplorer: A Virtual Platform for Performance/Reliability-Oriented Fine-Grained Design Space Exploration of Solid State Drives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1627–1638, 2015.

[38] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices," in *FAST'18*, pp. 49–66.

[39] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The Case of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator," in *FAST'18*, pp. 83–90.

[40] L. C. Bona, A. Elias, A. P. Ziviani, T. Cortes, R. Nou, and M. A. Alves, "Freezing Time: A New Approach for Emulating Fast Storage Devices Using VM," in *MASCOTS'18*.

[41] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash Storage Disaggregation," in *EuroSys'16*, pp. 29:1–29:15.

[42] IBM Research - Zurich, "Crail," http://www.crail.io/, 2017.