# Performance Analysis of NVMe SSD-based All-flash Array Systems

Young Tack Jin, Sungjoon Ahn and Sungjin Lee*
Circuit Blvd., Inc. and *DGIST
{*youngtack.jin, sungjoon.ahn*}*@circuitblvd.com, sungjin.lee@dgist.ac.kr*

***Abstract*—In this paper, we analyze and optimize I/O latency of a petabyte scale, high performance all-flash array (AFA) system based on NVMe SSDs. A flash-based SSD itself shows relatively low and consistent latency but, in AFA systems where several tens or hundreds of SSDs are combined in a single host machine, applications often see higher and more diverged I/O latency compared with a standalone SSD. To figure out a main source of such high I/O fluctuations, we analyze end-to-end I/O latency characteristics of a real-world AFA system. We find out that suboptimal kernel policies, parameters, and configurations result in serious degradation of I/O response times, causing very long tail latency. Based on our observations, we manually reconfigure several kernel parameters and revise storage firmware to achieve consistent I/O latency. Our experimental results show that, with the finely tuned kernel for AFA systems, the mean and standard deviation of the maximum latency can be reduced by x8 and x400, respectively. The findings in this work provide useful wisdom in designing system software and operating systems – CPU schedulers need to be revised to take into account the priority of IO-bound jobs, CPU isolation, and CPU-SSD affinity, and moreover, storage housekeeping protocols like SMART should be improved to avoid long tail latency.**

## I. Introduction

NVMe SSD-based all-flash array (AFA) systems recently receive serious attention from industry because their low latency and high throughput can aid various applications such as high-performance computing and real-time big data analysis [1]. An AFA also offers plenty of other benefits over disk storage – it consumes less power, does not require as much cooling as HDDs, and takes up less floor space. Thanks to all those benefits, an AFA is emerging as fast and economic storage solutions, rapidly replacing existing HDD-based storage clusters in data centers.

In spite of its increasing popularity, there have been no serious studies that analyze detailed performance profiles of all-flash arrays, along with their performance implications. While it is based on the same storage media (i.e., NAND flash), the internal architecture of an AFA differs from a conventional SSD. As shown in Fig. 1, an AFA is composed of several tens or hundreds of SSDs, each of which is connected to a host via high-speed PCIe over network switches. Such a large number of SSDs in an AFA machine are shared and accessed by host systems equipped with tens of x86 CPUs. These architectural and operational differences require us to reevaluate performance characteristics of an AFA in a somewhat different point of view from what we have seen in systems with few SSDs.

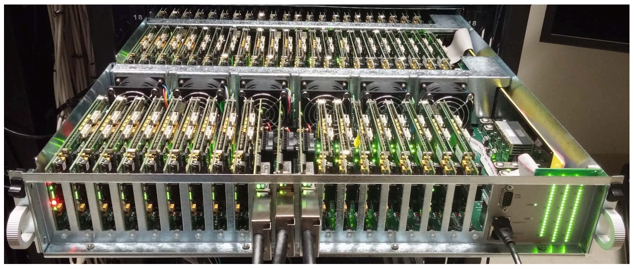


Fig. 1: An all-flash array (AFA) system with 64 NVMe SSDs, which is capable of offering 16 GB/s throughput through a Gen3x16 PCIe uplink.

The most urgent issue in an AFA may be the efficient management of a multitude of SSDs. For example, in an AFA, CPUs may frequently suffer from an interrupt storm coming from hundreds of SSDs. The mapping of limited CPUs to many SSDs would be another concern to be addressed to enjoy its high raw performance with little loss. Long tail latency is even more important than other systems. In an AFA, one request from a client is divided into multiple I/Os, which are then distributed to many SSDs in parallel as in RAID. In such a setting, long tail latency of the slowest SSD would decide system's overall responsiveness [2] – even if one SSD out of many, say 128 SSDs, shows long tail latency, the entire I/O from the client is delayed by the same amount. Compared with systems having few SSDs (8 to 12 SSDs), therefore, the impact of tail latency is much higher in an AFA.

In this paper, we carry out quantitative performance measurements over a large number of SSDs housed in an NVMe all-flash array system. We focus on analyzing latency profiles of *bare-bones* AFA hardware at a microscopic level, identifying its unique characteristics. Our goal is to provide useful performance implications of AFA hardware with system engineers so that they develop better system software or operating systems running on top of it. Thus, it is *not* our primary concern to understand how much firmware algorithms (e.g., FTL's garbage collector) and high-level system layers (e.g., file systems) have effect on AFA's performance. Including activities of high-level software layers would make it difficult for us to interpret and understand the performance of underlying AFA hardware, owing to noise caused by intermediate layers. For this reason, all the experiments in this paper are directly done on top of raw devices exposed by an AFA with minimal intermediate software layers.

Through extensive experiments on a real AFA machine, we find that inadequate configurations of the Linux kernel and host software result in suboptimal AFA performance, in particular, in terms of I/O latency. To improve the latency of many NVMe SSDs, we manually optimize Linux kernel configurations by changing key parameters relevant to a process priority, CPU scheduling, CPU affinity, and so on. I/O latency obtained after the manual optimizations is far closer to what underlying hardware can provide. The problems and solutions that we find from this study can be summarized as below.

- **Process Priority:** To improve user-perceived latency, the Linux kernel tends to give I/O-bound processes higher priority than CPU-bound ones [3]. The priority assignment policy of the current Linux kernel, however, is not optimal for AFA systems. In our experiments, even if I/O-bound jobs are assigned higher priority, performance drops caused by interference with CPU-bound jobs are unexpectedly high. This implies that the Linux kernel must be revised to employ a more aggressive policy that assigns the highest or higher enough priority to I/O-bound jobs.
- **CPU Isolation:** The Linux CPU scheduler often assigns CPU-bound jobs to CPUs which are already allocated to run I/O-bound jobs, if these I/O-bound jobs are in the idle state, waiting for acknowledgement from the SSD. Such typical CPU job scheduling is effective in terms of improving system-wide throughput since tasks are able to fully utilize available CPU cycles. However, it seriously degrades I/O latency of I/O bound jobs owing to interferences by CPU-bound jobs. Particularly, in an AFA with many SSDs, individual SSDs show wide and diverged latency distributions. Therefore, unless it does not hurt system throughput so much, the CPU scheduler should isolate CPU- and IO-bound jobs so that they run on separate cores.
- **CPU-IO Affinity:** The handling of interrupt signals from a large number of SSDs becomes a burden on AFA systems. The IRQ load balancer of the Linux attempts to evenly distribute interrupt handling jobs to available CPUs, but this load-balancing often degrades I/O latency because it involves frequent context switches and cache pollution. A similar phenomenon is also reported in network I/O-bound applications, but it may be even worse for AFA systems. Unlike networked systems employing 1-4 NICs, an AFA has several tens or hundreds of SSDs whose throughput and latency are better. This problem may be mitigated by manually forcing IRQ handlers to run on specific cores where corresponding tasks are currently executing.
- **Housekeeping Operation:** Housekeeping operations between host CPUs and NVMe SSDs, such as SMART, even cause very long tail latency. To get rid of its negative effects on I/O latency, it is necessary to improve SSD firmware together with new protocols so that housekeeping operations do not interfere with normal I/Os.

Even though the scope of this work is limited to the manual optimizations on raw AFA devices, our work can be directly used to improve various real-world applications in AFA systems. Modern system software and operating systems are organized in multi-layer structures [4], and thus our experiences would help software engineers build better systems and application software for all-flash arrays. For example, software engineers can optimize I/O latency of their applications by pinning CPU-bound and IO-bound threads on difference CPUs. As another example, mapping IRQ handlers to CPUs where designated I/O-bound tasks run would help lower fluctuations on I/O response times. Our framework and experimental methodology presented here even could be an excellent tool to profile SSDs in a fraction of time compared to traditional methods. With a large number of NVMe SSDs being profiled in parallel, one can finish the same task x10 or even x100 faster while still using a single host server.

This paper is organized as follows. Section II presents related work. Section III describes a system setup for our experiments. Section IV presents experimental results and analysis. Section V discusses how to use our work in storage systems software designs. Section VI concludes the paper.

## II. Related Work

In this section, we review prior studies closely related to this work, which analyze performance of flash-based SSD and optimize operating systems for fast storage. We also briefly introduce other techniques that attempt to refactor existing I/O stacks and propose new storage management algorithms for all-flash array systems.

**Performance Analysis of Flash-based SSDs:** There are studies that analyze I/O latency of NAND flash-based storage. Huang *et al.* presented black-box flash performance models to understand the state-of-the-art of SSDs and to explore the design space of such storage systems [5]. Desnoyers conducted empirical evaluations using a variety of real NAND devices to understand their latency characteristics [6]. Chen *et al.* also analyzed intrinsic characteristics and system implications of a single SATA SSD, focusing on understanding the impacts of various FTL activities inside the SSD [7].

Our SSD latency profiling presented in this paper can complement such performance models and measurements through fast yet relatively accurate latency distributions data collection. In addition, while the scope of the above studies is limited to a single storage system or NAND device, this work focuses on the analysis of I/O latency on a large-scale all-flash array system, comprised of multiple flash devices and connected to a host system through high-speed interfaces like PCIe.

**OS-level Optimization for Fast Storage:** For a better system integration with high-speed storage devices, a number of approaches were proposed in different areas of operating systems, such I/O interrupt handling [8], I/O queueing [9], [10], CPU affinity [11], [12], and virtualized I/O [13], [14], [15], respectively. Yang *et al.* showed that, for fast storage like NAND flash and PCM, interrupt handling overheads became a serious burden, and thus sometimes I/O polling could be a better way of delivering short I/O response times [8]. Bjørling *et al.* found that, in multi-core systems, lock-contentions between applications and kernel's I/O queues were a main

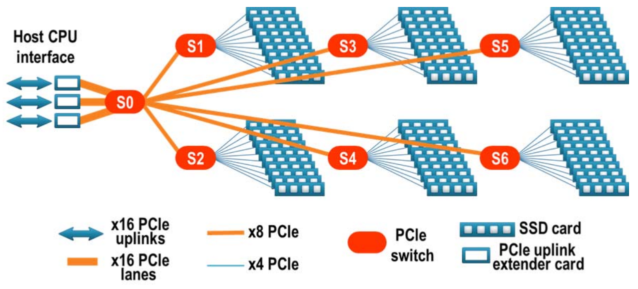


Fig. 2: PCIe fabrics structure of all-flash array systems

bottleneck in the entire I/O path [9]. They presented a multi-queue architecture and implemented it on the Linux kernel. Wu reported that interrupt affinity was an important factor that decided user-perceived latency of NVDIMM and proposed a run-to-complete method which was an extension of what proposed in [11]. Oh *et al.* addressed I/O scheduling delays in virtualized environments by proposing pipelined polling and multiple issues/completions techniques [13].

The aforementioned works are in line with ours: they commonly claimed that the current design of operating systems was ill-suited for fast storage like NAND flash and must be revised to take full advantage of its performance benefits. However, our study is different from the above studies in that we perform quantitative experiments over a large number (up to 64) of SSDs, rather than a single SSD, and present unique observations and solutions which are useful for optimizing AFA systems.

**Storage Stack Optimization:** There have been numerous studies that proposed refactored I/O stacks or new flash management algorithms for NAND flash. Caulfield *et al.* proposed techniques that bypassed deep software I/O stacks to offer short I/O latency [16], [17]. Ouyang *et al.* defined a new I/O architecture that eliminated intermediate layers in SSDs to offer fast and cost-effective web services in data centers [18]. Kim *et at.* presented a flash management method that hided excessive GC overheads from end users by serially managing an array of SSDs in AFA systems [19]. Zheng *et al.* proposed a software solution that tackled unsynchronized garbage collection in all-flash arrays with 18 SATA [20].

As mentioned in Section I, this study is not intended to propose new I/O stacks or storage management algorithms. Instead, we focus on analyzing performance characteristics of AFA systems with the aim of providing useful hints to storage system designers. In this regard, this study is orthogonal to the above studies, and can be combined with them for further optimization.

## III. Methodology

### A. Our All-flash Array System

Our NVMe all-flash array system is an Open Compute Project (OCP) [21] compliant storage appliance. Its physical dimension is compact (H $\times$ W $\times$ L: 96mm $\times$ 538mm $\times$ 914mm) and takes up only 2OU (OCP Unit) height on an OCP rack. The high density PCIe Gen3 fabrics hardware has 64 PCIe slots in total – 61 slots for PCIe devices and 3 slots for PCIe uplinks to maximum three host servers. The PCIe fabrics are designed around seven 96-lane/24-port PCIe switches in a two-level tree structure as shown in Fig. 2. The 61 PCIe devices are statically assigned to the 3 PCIe uplinks through the switches, which means that a static set of the PCIe devices is dedicated to a particular host.

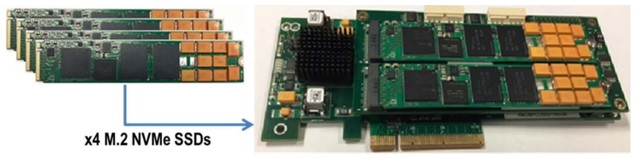


Fig. 3: M.2-PCIe carrier card

TABLE I: A Specification of the NVMe SSD

| Feature | Description |
|---|---|
| Host Interface | NVMe 1.2 - PCIe 3.0 x4 |
| Capacity (GB) | 960 |
| Random Read/Write (IOPS) | 160,000 / 30,000 |
| Sequential Read/Write (MB/s) | 1,700 / 750 |
| NAND Type | 3D MLC NAND |

The AFA can host different types of PCIe devices. The particular version we used is a PCIe M.2-PCIe carrier card, which is illustrated in Fig. 3. The card can embed 4 units of M.2 NVMe SSDs, enabling the AFA to host up to 244 M.2 NVMe SSDs. Current generation of our M.2 NVMe SSD is 960 GB capacity, making the capacity of the AFA approximately a quarter petabyte. Table I summarizes a detailed specification of the M.2 NVMe SSD.

In this paper, we focus on understanding a single host server. Thus, only one third of the AFA with maximum 64 SSDs are used as depicted in Fig. 4. This decision is made based on the server BIOS's PCIe enumeration capability as well as typical number of NVMe SSDs proposed in the latest hyper-scale AFA designs. One example system is described in [22]. The server is connected to the AFA via one Gen3x16 PCIe uplink that is capable of delivering 16 GB/s raw throughput. Our host server has two Intel Xeon E52690 v2 CPUs based on the NUMA architecture, and one of them, CPU2, is responsible for communicating with our AFA machine. As the host operating system, we use CentOS 7 distribution and the kernel version 4.7.2 which comes with the latest NVMe device driver. Our AFA system is equipped with 64 NVMe SSDs, so it exposes 64 raw block devices (i.e., `/dev/nvme0`, ..., `/dev/nvme63`) to the host OS. This enables the host to access individual NVMe SSDs through the standard block I/O interface.

### B. Measurement, Metric, and Workload

In order to minimize interferences by host software, we have used as little intermediate layers as possible between applications and the AFA machine. To this end, we directly run benchmark instances atop individual block devices. Linux's

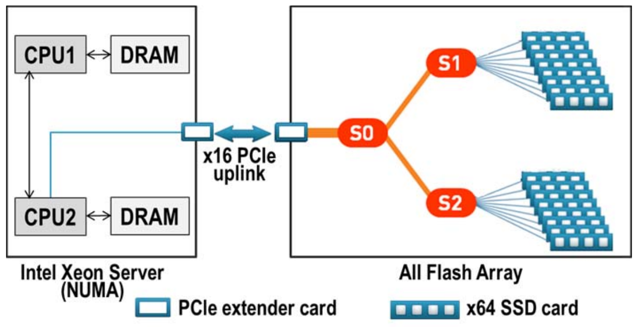


Fig. 4: A diagram of our experimental setup, including a storage host server, an all-flash array, and their interface.

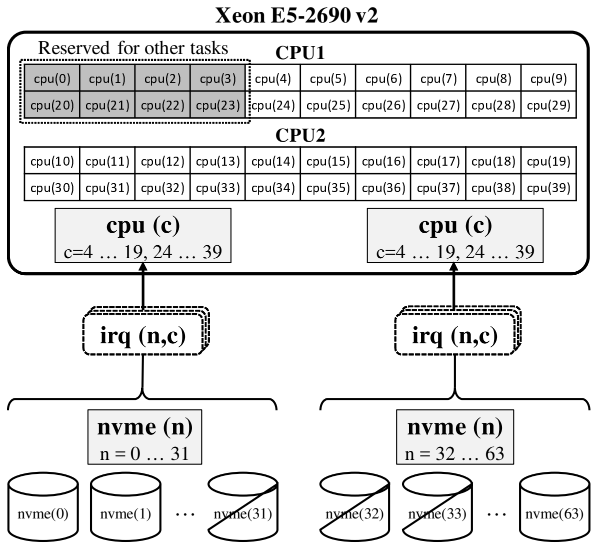
volume manager, device mapper, and file systems are disabled or not used for our experiments. To further reduce latency variations by SSD internals, such as garbage collection and wear leveling, we measure latency by keeping all the SSDs in the FOB (Fresh Out-of-the Box) state through a NVMe format command [23]. In addition to this, we follow the chapter 9 of SNIA PTS-E guidelines to minimize the systems overhead on I/O latency [24].

Throughout the paper, average random-read latency and random-read latency distributions up to 6-nine (99.9999%) percentiles are used as performance metrics. The reasons for selecting them as our main evaluation metrics are as follows. First, up to 9-nine percentiles read latency is commonly used by enterprise SSD deployments in hyper-scale data centers [25], [26]. Second, unlike writes that are able to be buffered and performed in background later, read requests cannot be delayed and must be served as quickly as possible not to degrade user-perceived I/O latency. Third, while random-read latency is highly affected by several OS policies, such as CPU scheduling, I/O scheduling, and interrupt handling, the performance of sequential reads is mostly decided by the raw throughput of underlying SSD architectures (e.g., # of channels and # of ways) [7]. In our preliminary evaluation, we observed the throughput of sequential reads was high enough all the time to fully saturate available PCIe bandwidths.

For a workload generation, we select `FIO` as a benchmark tool because it provides useful features, including statistical information of I/O latency distributions, thread pinning, direct access to raw block devices, and queue depth control. We configure `FIO` workloads to 4KB random reads with a single queue depth per thread. Linux Asynchronous I/O (`libaio`) is used for this workload.

## C. Default Configuration

Our AFA machine has many CPU cores and NVMe SSDs, which creates numerous combinations of various parameters. As examples, we should decide 1) the number of FIO instances to run in the system simultaneously, 2) how many FIO instances execute on the same CPUs, 3) how many NVMe SSDs can be shared by multiple FIOs, and so on. To reduce the parameter space, but not to miss important implications from our experiments, we carefully design an experimental setup. Our goal is to make a reasonable default configuration that is expected to give us the lowest I/O latency, using common wisdoms or practices that are well known to field engineers. Based on results from the default setup, we are able to figure out main causes that abnormally disturb consistent I/O latency unlike our expectation.



Xeon E5-2690 v2
Reserved for other tasks
CPU1
cpu(0) cpu(1) cpu(2) cpu(3) cpu(4) cpu(5) cpu(6) cpu(7) cpu(8) cpu(9)
cpu(20) cpu(21) cpu(22) cpu(23) cpu(24) cpu(25) cpu(26) cpu(27) cpu(28) cpu(29)
CPU2
cpu(10) cpu(11) cpu(12) cpu(13) cpu(14) cpu(15) cpu(16) cpu(17) cpu(18) cpu(19)
cpu(30) cpu(31) cpu(32) cpu(33) cpu(34) cpu(35) cpu(36) cpu(37) cpu(38) cpu(39)
cpu (c)
c=4 … 19, 24 … 39
cpu (c)
c=4 … 19, 24 … 39
irq (n,c)
irq (n,c)
nvme (n)
n = 0 … 31
nvme (n)
n = 32 … 63
nvme(0)
nvme(1)
nvme(31)
nvme(32)
nvme(33)
nvme(63)

Fig. 5: CPU-SSD geometry

We use a static allocation scheme that distributes `FIO` workload threads across a set of CPU cores. The `FIO` workload generation thread for each SSD run on a dedicated physical CPU core via `FIO`'s `cpus_allowed` option. As briefly mentioned before, the host server we used has two Intel Xeon E52690 v2 CPUs, each with 10 physical and 20 logical (Hyper-Threading) cores. For this system, we decide to use 32 logical CPU cores to accommodate maximum 64 SSDs, leaving out eight logical CPU cores for other system tasks.

Fig. 5 depicts the CPU-SSD geometry that shows how we map logical CPU cores to NVMe SSDs. There are 40 logical CPU cores, `cpu(0)`, `cpu(1)`, ..., `cpu(39)`. There are also 64 NVMe SSDs, `nvme(0)`, `nvme(1)`, ..., `nvme(63)`. Each `FIO` thread is mapped to one NVMe, and two `FIO` threads share the same logical core. For instance, two `FIO` threads for `nvme(0)` and `nvme(32)` both run on `cpu(4)`. Similarly, `cpu(39)` is shared by `nvme(31)` and `nvme(63)`.

The above setting gives us two expectations. First, since `FIO` threads are distributed across specific cores, we expect that the Linux kernel would assign CPU-bound threads/processes to eight idling CPUs, `cpu(0)`, ..., `cpu(3)` and `cpu(20)`, ..., `cpu(23)`, which are reserved for other tasks. Our another expectation is that the Linux's IRQ load balancer would assign IRQ handlers to proper CPU cores, considering affinities between `FIO` threads and NVMe devices. In the Linux kernel, individual SSDs have IRQ handlers created for every logical CPU core available on the system

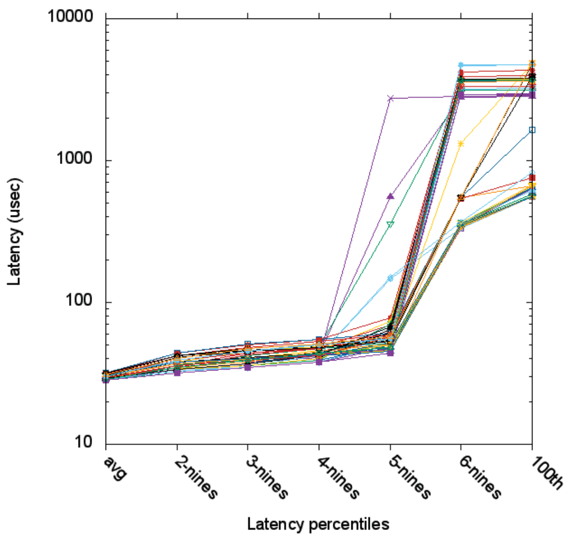


Fig. 6: Latency distributions based on default system configuration.

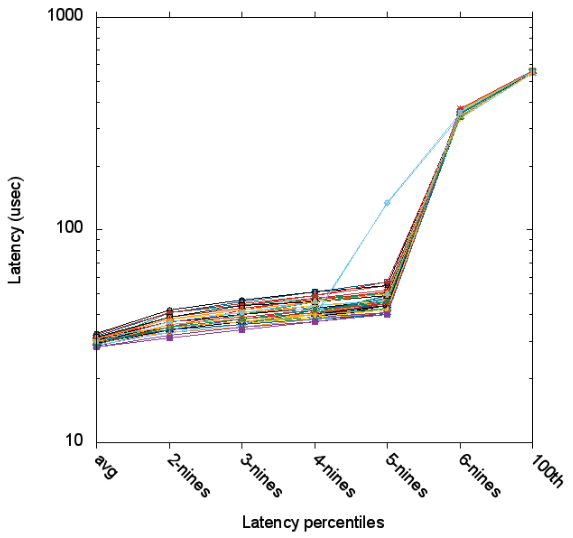


Fig. 7: Latency distributions after assigning the highest priority to `FIO`

– that is, total 2,560 IRQ handlers are created for 40 CPU cores with 64 NVMe SSDs. In other words, each CPU has 64 IRQ handlers for all the SSDs. In Fig. 5, `irq(n,c)` denotes an IRQ handler for an NVMe SSD `nvme(n)` running on a logical CPU core `cpu(c)`. In our configurations, therefore, if the load balancer works properly, only two IRQ handlers are active in each CPU core, while all the other IRQ handlers remain inactive. For example, for `cpu(4)`, `irq(0,4)` and `irq(32,4)` are active but the other 62 `irq(n,4)`s, where `n` is 1, 2, ... , 31, 33, ..., 62, 63, are all inactive.

Each test instance of `FIO` is run for 120 seconds to collect I/O latency in the unit of microseconds ($\mu$sec). From `FIO` output logs, average completion latency, completion latency percentiles from 2-nines to 6-nines, and 100th (maximum) latency are plotted. We then analyze how different Linux configuration parameters influence the latency distributions.

## IV. Performance Analysis

### A. Experimental Results with Default Configuration

Fig. 6 depicts the latency distributions of 64 SSDs obtained under the default Linux configuration. There are 64 lines with different colors, each of which corresponds to a latency distribution of a specific NVMe SSD (e.g., `nvme(0)`). In this figure, the names of NVMe SSDs themselves do not contain any useful information. For the sake of clear presentation, we do not display SSDs names in the figure.

The distribution shows relatively wide spreads from 5-nines and the worst-case latency at approximately 5 $m$sec. This is somewhat surprising results. Our single NVMe hardware itself is designed to deliver 25 $\mu$sec for a read request. Even for clustered NVMe SSDs connected through PCIe switches, a read latency just increases to 30 $\mu$sec with an additional delay of 5 $\mu$sec. Moreover, we disabled almost all of the intermediate layers to get rid of their interferences on I/O latency, and garbage collectors inside SSDs were not triggered while running `FIO` threads. As a result, considering a very short latency of NVMe devices, the worst-case latency of 5 $m$sec and read latency fluctuations are unexpectedly high.

### B. Impact of Process Priority

From our analysis with a LTTng tool [27], we find that the high worst-case latency and the high latency fluctuations are caused by interferences from other processes and threads. Even though we only run 64 `FIO` threads on CPUs, there are still many other processes and kernel threads running in the Linux, such as llvmpipe (for GNOME's GUI service), lttng-consumerd (for kernel tracing), IRQ handlers, SSH daemons, and so on. We observe that these processes and kernel threads often interfere with the executions of `FIO` threads.

Before looking into the effects of detailed kernel parameters and policies on I/O latency, we first attempt to apply the simplest method that gives IO-bound jobs much higher priority than CPU-bound ones and to understand its impact. We manually set the priority of the `FIO` process to the highest value of 99 with Linux's `chrt` command. Compared to Fig. 6, the latency distributions in Fig. 7 become more converged. Also, the worst-case latency reduces to about 600 $\mu$sec. The results in Fig. 7 indicate that relatively lightweight background threads/processes could badly affect overall I/O latency if fast storage is used. In particular, in an SSD array composed of multiple SSDs, individual SSDs exhibit very diverged latency distributions.

Even with the highest process priority set for `FIO` threads, however, the other processes still interfere with `FIO`'s execution. For example, response time fluctuations are still significant in Fig. 7.

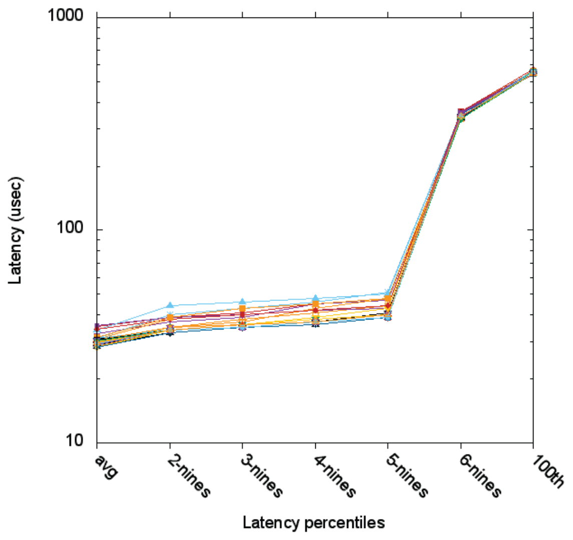


Fig. 8: Latency distributions after setting CPU isolation

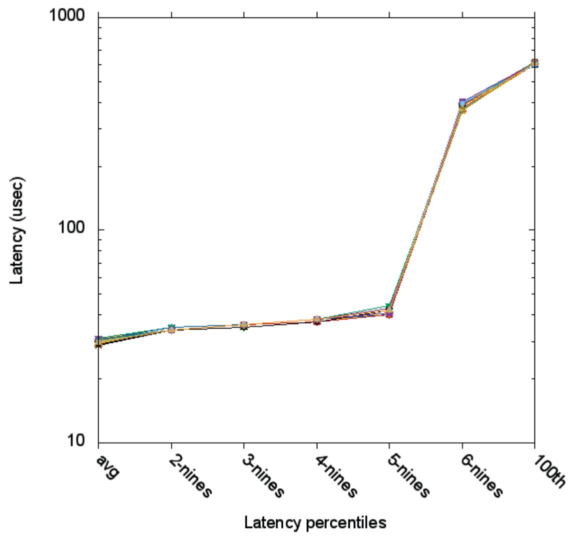


Fig. 9: Latency distributions after setting CPU affinity

### C. *Impact of CPU Isolation*

To protect from interference mentioned above, we *explicitly* dedicate 32 CPU cores `cpu(4)`, ..., `cpu(19)` and `cpu(24)`, ..., `cpu(39)` to run `FIO` in isolation. As described in Section III, we already have *implicitly* isolated `FIO` within specific CPU cores, expecting that Linux's default CPU scheduler would assign background processes/threads to reserved eight CPU cores `cpu(0)`, ..., `cpu(3)` and `cpu(20)`, ..., `cpu(23)`, separating CPU-bound jobs from IO-bound ones. However, Linux's default scheduler did not work as our expectation – many background processes and threads often interrupted the execution of FIO threads, resulting in the increase of overall I/O latency. By manually isolating such processes from CPU cores assigned to FIO threads, we are able to completely get rid of such interventions.

Since the Linux kernel does not allow us to change the default CPU isolation setting at run time, we manually configure a kernel boot option for the CPU isolation. The boot option also specifies the most infrequent timer interrupts for isolated CPUs (which is called a 1 Hz timer) and it keeps CPU cores from getting into idle state. The example commands below are for a Xeon E5 host server that has 20 physical CPU cores and 40 logical Hyper Threading CPU cores.

```
isolcpus=4-19,24-39 nohz_full=4-19,24-39
rcu_nocbs=4-19,24-39 processor.max_cstate=1
idle=poll
```

On top of `FIO` priority changes via `chrt`, the CPU isolation improves the latency distributions and the worst-case latency. Fig. 8 shows more converged latency distributions when compared with Fig. 7.

One might think that such manual CPU isolation is not a viable option in practical environments. This may be true for applications servers where several thousands of CPU-bound and IO-bound jobs are running simultaneously. However, this manual setup would be feasible enough for special systems like storage servers, where the roles of CPU-bound and IO-bound tasks are clearly divided (e.g., host interface tasks and storage management tasks), and these tasks can be separated into different CPU cores in a relatively easy manner.

### D. *Impact of CPU Affinity*

As mentioned in Section III-C that described the default experimental setup, we intended to equally distribute IRQ handlers across a set of CPUs by allowing only dedicated `FIO` threads to execute actively on specific CPU cores. For example, according to our CPU-SSD mapping, for the CPU core 4, `cpu(4)`, two IRQ handlers, `irq(0,4)` and `irq(32,4)`, for `nvme(0)` and `nvme(32)` are expected to run actively.

However, our analysis with LTTng tools reveals that some IRQ handlers are executed on other CPUs than their designated CPUs. For instance, `irq(0,4)` is executed on other logical CPUs (e.g., `cpu(30)`) rather than its designated `cpu(4)`. This indicates that Linux's IRQ load balancer does not take into account CPU affinity when handling interrupt signals from NVMe devices. If `cpu(30)` services interrupt signals instead of `cpu(4)`, it should talk with `cpu(4)` later to notify of the arrival of interrupts. In particular, for fast storage like NVMe SSDs, additional latency induced by context switches and cache pollution is not negligible. Therefore, we need additional configurations to force IRQ handlers of each SSD to execute only on the designated CPU core.

To this end, we ensure proper settings of CPU affinities of 2,560 IRQ handlers which are created for 64 NVMe SSDs immediately after the system boot. CPU affinities can be manipulated through Linux kernel 4.7.2's `procfs` and `tuna` commands [28]. As illustrated in Fig. 9, when the CPU affinity

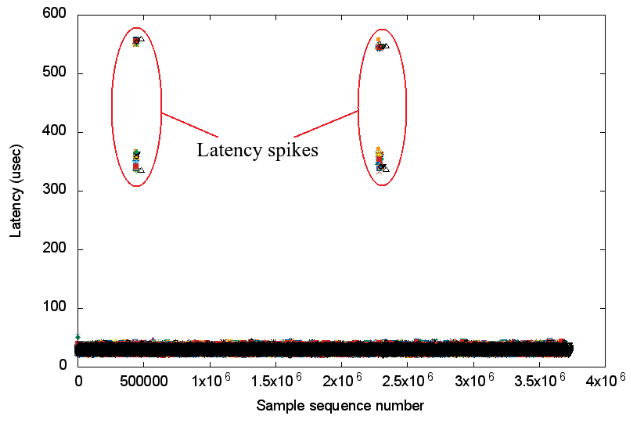


Fig. 10: Scatter plot of latency samples from 32 SSDs

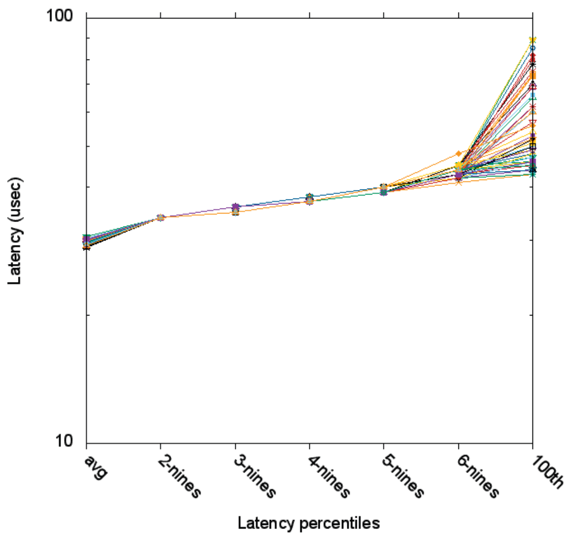


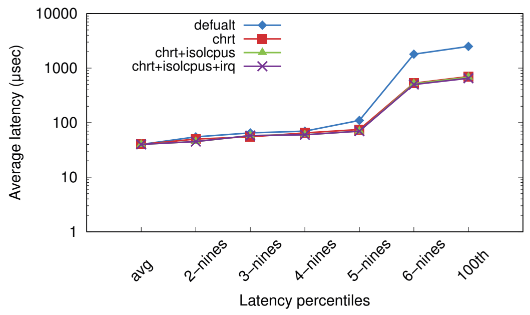
Fig. 11: Latency distributions obtained from experimental firmware

is set properly for all IRQ handlers, the latency distributions across multiple SSDs become more converged.

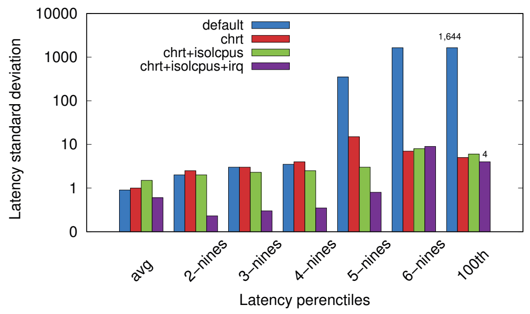
### *E. Impact of Housekeeping Operations with SSD Firmware*

After fixing the CPU-SSD affinity problem, we have converged latency profiles among all the SSDs. However, it still seems that we see abnormally longer latency in 6-nineth and 100th latency buckets compared to 5-nines and below. After close examinations, we find out that this long latency is caused by periodically repeated latency spikes as shown in Fig. 10.[1] For example, around 500,000 and 2,300,000 in Fig. 10, long latency spikes are observed. Further investigations reveal that these spikes are due to periodic SMART data collections of the NVMe SSDs. This behavior does not violate NVMe specification and the SSD meets all the performance requirements from customers. However, it is still desirable to eliminate latency spikes coming from SSD's internal operations.

[1]Note that one practical issue we encountered in obtaining Fig. 10 was that results from 64 SSDs appeared inaccurate by huge margins when latency logging was enabled. We worked around this problem by collecting latency logs from 32 SSDs and the data was sufficient to locate the root cause of this periodic latency spikes.



(a) Average latency



(b) Standard deviation

Fig. 12: Comparison of four system configurations

As a temporary remedy, we develop experimental SSD firmware which has SMART data update/save disabled. Fig. 11 shows our experimental results. The worst-case latency improves from around 600 $\mu$sec to around 90 $\mu$sec based on the experimental SSD firmware, which shows that very long tail latency disappears by suppressing SMART activities. However, with this firmware change, the range of maximum latency from all the SSDs is still wide. There might be more room to improve SSD firmware or hardware to accomplish more converged latency distributions. But, this is planned as one of our future tasks.

### *F. Comparison of Kernel Configurations*

In this subsection, we summarize and compare the impacts of four different Linux configurations on improving I/O latency. Fig. 12 compares four latency distributions with different system configurations - default (Section IV-A), chrt (Section IV-B), isolcpus (Section IV-C), and irq (Section IV-D). We plot the average and standard deviations of read latency percentiles collected from 64 NVMe SSDs.

From Fig. 12, one can see that adjustment of the `FIO` process priority yields the most impact on the average latency. On the standard deviation chart, the CPU affinity modification drastically reduces the standard deviation for the average latency and 2- to 5- nines latency distributions. The standard

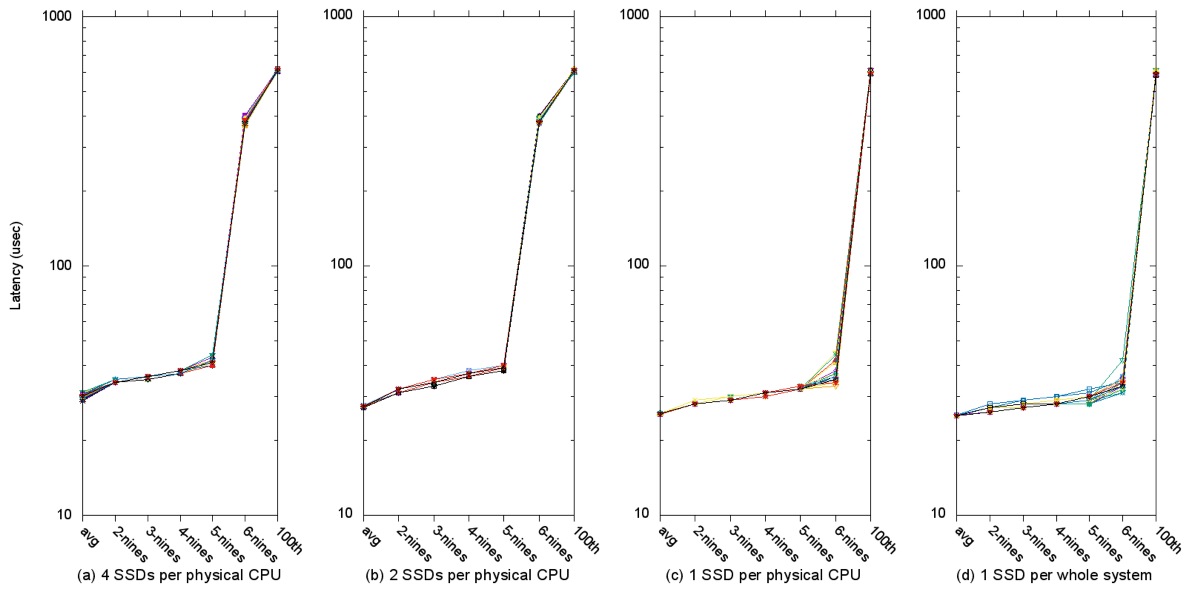


Fig. 13: Latency distributions per number of SSDs per physical CPU core

TABLE II: Varying Number of SSDs / CPU Core

| Fig # | # of SSDs / physical core | # of IRQ handlers / logical core | # of FIO threads / logical core | # of FIO threads in the system | runs/# SSD sets for merged results on disjoint SSD sets |
|---|---|---|---|---|---|
| Fig. 13(a) | 4 | 2 | 2 | 64 | 1 |
| Fig. 13(b) | 2 | 1 | 1 | 32 | 2 |
| Fig. 13(c) | 1 | 1 | 1 | 16 | 4 |
| Fig. 13(d) | 1 FIO thread on the entire system | | | | 64 |

deviation of the worst-case latency improves from 1,644 to 4 with all the system optimizations.

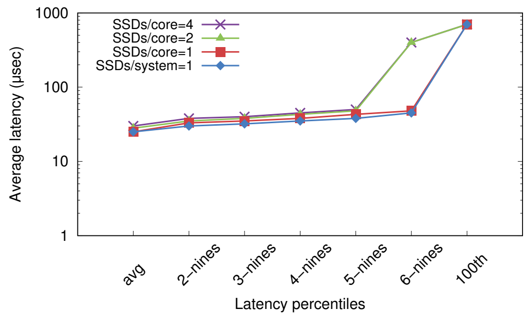
## G. Balance between # of CPU cores and # of SSDs

Do we have a good balance between number of CPU cores and number of SSDs? This might be the most important question for those who are responsible for designing storage servers comprised of multiple CPU cores and NVMe SSDs. For example, given the number of CPUs, some might want to determine the best number of NVMe SSDs suitable for their AFA systems (and vice versa). If we put so many SSDs to AFA systems, it would provide higher capacity with better TCO, but customers may suffer from poor I/O latency. In order to answer this question, we carry out a set of experiments to see the impact of CPU utilization on SSD latency profiles.

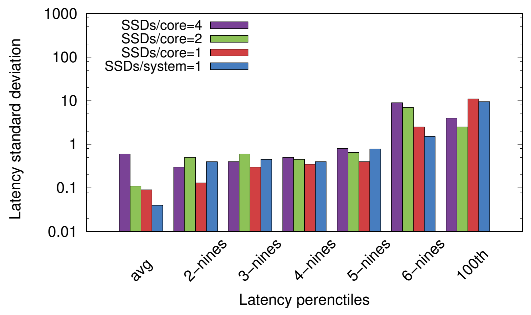
Fig. 13 compares the latency distributions according to the number of SSDs per physical CPU core. Table II summarizes the details of experimental setups corresponding to Fig. 13, which includes the number of SSDs per physical CPU core and other numbers (e.g., the number of IRQ handlers per logical CPU cores). Some test configurations require multiple test runs to obtain results for all the 64 SSDs. Each test runs a different number of FIO threads (e.g., 64 FIO threads in Fig. 13(a) and 16 FIO threads in Fig, 13(c)), creating a different amount of read traffic to the AFA. This, however, does not cause biased results. As mentioned in Section III, in order to focus on analyzing latency distributions between CPUs and SSDs, FIO was configured to generate 4KB random reads with a single queue. Even with 64 FIOs, the aggregate throughput issued by FIO threads is limited to 8.3 GB/s, which are lower than 16 GB/s of the PCIe uplink and 108 GB/s of the underlying 64 SSDs (1.7 GB/s each). Consequently, the AFA is able to serve incoming I/Os promptly regardless of a number of FIO threads running. Note that Fig. 13(a) is identical to Fig. 9. In addition, Fig. 14 illustrates the average and standard deviations of read latency percentiles collected from various SSD setups.

We can see that latency distributions are quite similar among these test cases. Our allocation of 4 SSDs to a CPU core yields close results to the system workload with 1 SSD per CPU core, except that it delivers a higher latency at the 6-nineth percentile. When the number of SSDs allocated to a physical CPU reduces to 1, its performance is almost the same as running one FIO thread on the entire system.

Fig. 13 confirms the validity of running latency profile workload over multitudes of SSDs as long as CPU utilization is kept at sufficiently low level. It is a straightforward solution to decide the best combination of CPUs and SSDs, but its drawback is that it could seriously underutilize available CPU cores. This asks us to optimize system software based on what we have learned from a series of the experiments.



(a) Average latency



(b) Standard deviation

Fig. 14: Comparison of different number of SSDs per physical CPU core

## V. Implications on System Software

Our work could provide some insights in designing systems software for a large number of high performance flash storage devices. We discuss three ideas in this section.

**Better CPU Scheduling Algorithms:** The current Linux kernel's CPU scheduler is not designed to efficiently schedule I/O-bound and CPU-bound processes on all-flash array systems shared by many cores. As discussed in Section IV-B, CPU-bound jobs often interfered with I/O bound jobs, degrading overall I/O latency. Moreover, as pointed out in Section IV-C, the Linux's CPU scheduler allocated CPU-bound processes on CPU cores being used by IO-bound jobs, even though there were sufficient CPU cores reserved for CPU-bound processes. Currently, we fixed the problems above by manually assigning the highest priority to I/O-bound jobs and isolating them from CPU-bound jobs. However, the ultimate solution that addresses the aforementioned issues is to develop a better CPU scheduling algorithm which can properly schedule CPU- and IO-bound jobs across multiple CPU cores to minimize interferences among them.

**Better IRQ Handling Considering CPU Affinity:** In our experimental results, we observed that the handling of interrupt requests caused high I/O latency. Unfortunately, the allocation of IRQ handers on improper CPU cores was frequently observed with the current CPU scheduling algorithms. For example, even in the latest Linux kernel, IRQ handlers were often executed on a remote CPU that required additional context switches, along with cache pollution. Our experimental results suggested that it was necessary to devise a better IRQ allocation algorithm.

Astute readers might notice, similar issues related to interrupt handling were already observed in network-intensive systems [29]. Some storage researchers also reported that interrupt handling was becoming a serious bottleneck in the I/O stack [8], [11]. They suggested various ways of addressing the problem – using a dedicated CPU core to handle interrupt signals [29] or using polling instead of interrupt [8]. Even though more quantitative evaluations are necessary, it is unclear whether or not those previous attempts would be still valid for AFA systems. For example, it is unsure if one or few dedicated CPUs would sufficiently handle an interrupt storm from hundreds of high-speed SSDs. Using polling might reduce I/O latency, but I/O throughput inevitably deteriorates (which is another important requirement from customers) [11]. We leave those questions open to future work.

**Better Housekeeping Protocols:** In Section IV-E, we disabled a SMART feature of NVMe SSD's firmware, so as to suppress regular I/O traffic for exchanging reliability information between the host and underlying hardware. Even though disabling the SMART feature was feasible to understand its negative effects on performance, it couldn't be an option for production systems that should maintain underlying storage devices reliably. In order to take advantage of a low latency of AFA systems and to keep monitoring the health conditions of the hardware at the same time, it is required to develop a new housekeeping protocol with negligible communication overheads.

## VI. Conclusion

In this paper, we presented several optimization techniques for all-flash array systems (AFA). We found out that default kernel configurations did not work efficiently with high-performance AFA systems, exhibiting high read-latency fluctuations and exceptionally long tail latency. With the system optimizations for workload process priority and CPU isolation/affinity of NVMe device driver's IRQ handlers, we were able to obtain vastly improved latency from a large number of SSDs. We further improved the long tail latency by making experimental SSD firmware changes to remove periodic latency spikes within SSDs.

We expect that our experimental environments built for testing a large number of NVMe SSDs would be useful in several aspects. With the ability to make tens or even hundreds of times more measurements over a given time, one can carry out data driven engineering analyses on a regular basis. For instance, it is more cost-effective to detect and root cause latency outliers from daily SSD firmware builds. Even more, what we learned from this study helps us build better system software. As an example, one can implement a new CPU scheduler with more agility leveraging the more current and extensive SSD latency data sets.

Our future work includes prototyping new CPU schedulers and I/O load balancers, exploring all-flash array performance implications in NUMA architecture, and applying the profiling framework to full-fledged applications. In addition, in this study, we haven't analyzed the details of how garbage collection affects latency of all-flash array systems. In the near future, we will assess latency distributions in used (non-FOB) SSD states and will analyze their impacts on performance.

## Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (No. 2017R1E1A1A01077410) and the DGIST R&D Program of the Ministry of Science, ICT and Future Planning (18-EE-01). This work was done when Young Tack Jin and Dr. Sungjoon Ahn were with SK Hynix Memory Solutions (SKHMS). (Corresponding author: Sungjin Lee)

## References

[1] N. E. Workgroup, "Nvm express specification revision 1.2," 2014.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[3] J. Aas, "Understanding the linux 2.6. 8.1 cpu scheduler," *Retrieved Oct*, pp. 1–38, 2005.

[4] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: a facebook messages case study." in *FAST*, vol. 14, 2014.

[5] H. H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance modeling and analysis of flash-based storage devices," in *Proceedings of the Symposium on Mass Storage Systems and Technologies*, 2011, pp. 1–11.

[6] P. Desnoyers, "Empirical evaluation of nand flash memory performance," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 50–54, 2010.

[7] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the International Joint Conference on Measurement and Modeling of Computer Systems*, 2009, pp. 181–192.

[8] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt." in *Proceedings of the USENIX conference on File and Storage Technologies*, vol. 12, 2012.

[9] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: introducing multi-queue ssd access on multi-core systems," in *Proceedings of the international systems and storage conference*, 2013.

[10] B. Caldwell, "Improving block-level efficiency with scsi-mq," *arXiv preprint arXiv:1504.07481*, 2015.

[11] A. Wu, "Methods to achieve low latency and consistent performance," in *Flash Memory Summit*, 2015.

[12] D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system iops on low-cost, commodity hardware," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*, 2013.

[13] M. Oh, H. Eom, and H. Y. Yeom, "Enhancing the i/o system for virtual machines using high performance ssds," in *Proceedings of the international conference on Performance Computing and Communications*, 2014, pp. 1–8.

[14] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework," in *Proceedings of the Symposium on Mass Storage Systems and Technologies*, 2015, pp. 1–11.

[15] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield, "Strata: High-performance scalable storage on virtualized non-volatile memory," in *Proceedings of the USENIX conference on File and Storage Technologies*, 2014, pp. 17–31.

[16] A. M. Caulfield and S. Swanson, "Quicksan: a storage area network for fast, distributed, solid state disks," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013, pp. 464–474.

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

[18] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "Sdf: Software-defined flash for web-scale internet storage systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 471–484.

[19] B. Kim, J. Kim, and S. H. Noh, "Managing array of ssds when the storage device is no longer the performance bottleneck," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2017.

[20] D. Zheng, R. Burns, and A. S. Szalay, "Optimize unsynchronized garbage collection in an ssd array," *arXiv preprint arXiv:1506.07566*, 2015.

[21] O. C. Project, 2016. [Online]. Available: http://www.opencompute.org/

[22] C. Petersen, "Introducing lightning: A flexible nvme jbof," 2016. [Online]. Available: https://code.facebook.com/posts/989638804458007/

[23] B. Leitao, "nvme-format(1)," 2016. [Online]. Available: https://github.com/linux-nvme/nvme-cli/blob/v0.7/Documentation/nvme-format.txt

[24] SNIA, "Snia sss pts-e (solid state storage performance test specification enterprise) v1.1," 2013.

[25] E. Kim, "Ssd performance – a primer: An introduction to solid state drive per-formance, evaluation and test," 2013. [Online]. Available: http://www.snia.org/sites/default/files/SNIASSSI.SSDPerformance-APrimer2013.pdf

[26] Microsoft, "Storscore," 2016. [Online]. Available: https://github.com/Microsoft/StorScore

[27] LTTng, "Lttng documentation v2.7," 2015. [Online]. Available: http://lttng.org/docs/

[28] M. Enterprise and L. Brindley, "Red hat enterprise mrg 1.3 tuna user guide," 2010.

[29] Red Hat, "Red hat enterprise linux performance tuning guide," 2017. [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-cpu-irq.html