Exploring System Challenges of Ultra-Low Latency Solid State Drives

Sungjoon Koh, Changrim Lee, Miryeong Kwon and Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory, Yonsei University,
{*skoh, crlee, mkwon*}*@camelab.org, and mj@camelab.org*

Abstract

We quantitatively characterize performance behaviors of a real ultra-low latency (ULL) SSD archive by using a real 800GB Z-SSD prototype, and analyze system-level challenges that the current storage stack exhibits. Specifically, our comprehensive empirical evaluations and studies demonstrate i) diverse performance analyses of the ULL SSD, including a wide range of latency and queue examinations, ii) I/O interference characteristics, which are considered as one of the great performance bottlenecks of modern SSDs, and iii) efficiency and challenge analyses of a polling-based I/O service (newly added into Linux 4.4 kernel) by comparing it with conventional interrupt-based I/O services. In addition to these performance characterizations, we discuss several system implications, which are required to take a full benefit of ULL SSD in the future.

1 Introduction

The state-of-the-art solid state drives (SSDs) begun to offer extremely high bandwidth by connecting with an on-chip processor controller, such as a memory controller hub (MCH) or processor controller hub (PCH) via PCI Express (PCIe) buses. For example, NVMe SSDs (e.g., Intel 750 [10]) provide read and write bandwidths as high as 2.4 GB/s and 1.2 GB/s, respectively, which are approximately 4.4× and 2.4× higher than the performance of a conventional SSD, which resides on an I/O controller hub (ICH). Several industry prototypes [5, 20] promise to deliver high performance, ranging from 8.4 hundred thousand IOPS to one million IOPS. Thanks to the high bandwidth, the high-end NVMe SSDs are widely used for data-intensive applications and server systems as a disk drive cache [7, 18], burst buffer [17], and in-memory computing storage [26].

Even though the bandwidth of modern high-end SSDs almost reaches the maximum performance that PCIe buses can deliver, unfortunately, their system-level traverse latency is still far different from that of other memory technologies or fast peripheral devices, such as DRAM and GPGPU. To bridge the latency disparity between the processors and SSDs, new flash memory based archives, called Z-SSD, get more attention from both industry and academia. This new type of SSDs can provide *ultra-low latency* (ULL), which has a great potential to bring storage close to computational components [11, 23, 15]. Specifically, the new flash medium that ULL SSDs employ is a revised version of vertically-stacked 3D NAND flash (48 layers) whose memory read latency for a page is 3 μs [19], which is 8× faster than the fastest page access latency of modern multi-level cell flash memory. ULL SSDs are expected to satisfy high service-level agreements (SLA) and quality of services (QoS) while offering high storage capacity, compared with other types of new memory, such as resistive memory (ReRAM) [8].

Industry articles uncover the low-level device performance of ULL SSDs, but unfortunately, it is difficult to foresee their actual performance by taking into account diverse system execution parameters. In addition, analyzing system-level challenges that should be addressed to take full advantages of ULL SSDs is non-trivial since these new types of SSDs are unfortunately unavailable in a public market yet. In this work, we characterize performance behaviors of a real 800GB Z-SSD prototype and analyze system-level challenges in integrating ULL SSDs into the current software storage stack. The main observations and system-level analyses of this paper can be summarized as follows:

• *Performance and system implications.* We observe that the ULL SSD shortens the read latency and write latency of a high-end NVMe SSD [10] by 36% and 57%, on average, respectively. Specifically, even though systems increase I/O queue depths, the ULL SSD provides a sustainable ultra-low latency (for both average and long tail) while the average and 99.999% latencies of high-end NVMe SSD seriously degrade. Our analysis also shows that the current rich queue mechanism of NVMe

3D NAND	BiCS	V-NAND	Z-NAND
# layer	48	64	48
t_R	45μs	60μs	3μs
t_{PROG}	660μs	700μs	100μs
Capacity	256Gb	512Gb	64Gb
Page Size	16KB/Page	16KB/Page	2KB/Page

Table 1: Comparisons of 3D flash characteristics [3].

may be over-kill for ULL SSDs; 6∼8 queue entries are good enough to maximize the bandwidth of ULL SSDs, on average, and even in the worst case, it consumes only 16 queue entries, which is well aligned with light queue mechanisms, such as native command queue (NCQ). We also observe that the ULL SSD minimizes many I/O interferences between reads and writes, which are considered as a great performance bottleneck of conventional SSDs. Since flush operations or metadata writes that modern file systems must handle are usually intermixed with user requests, the performance of many SSDs are unfortunately not as much promising as SSD vendors promote, in real-life storage stack [24, 13, 14].

• *Storage stack analyses.* Our study reveals that, while still high-performance NVMe SSDs may not require a polling-based I/O completion routine (supported from Linux 4.4), ULL SSDs can take an advantage of polling (instead of using an interrupt). However, it raises several system-level challenges as the polling routine fully occupies one or more heavy CPU cores. Specifically, it consumes CPU cycles more than 99% of the total I/O execution (based on 1 core). In addition, the load/store instructions that access system memory consume more than 95% of the total pipeline slots for just polling the I/O completion. Even though the polling-based NVMe I/O service can shorten the latency for the most case, we also observe that the current polling technique should be optimized further; it can potentially hurt overall system performance as its overheads have a great impact on making five nine latency (99.999%) much longer than that of the interrupt-based I/O service (as high as 115%).

2 Background

Ultra-low latency. Modern SSDs can satisfy high bandwidth requirements with various architectural supports, such as internal parallelism, I/O queueing/scheduling and DRAM buffers. However, shortening the latency for a basic unit of I/O operation requires low-level memory design changes and device updates. New flash memory, called *Z-NAND*, leverages single-level cell (SLC) based 3D flash design but optimizes several I/O circuitries, such as a default I/O page size and DDR interfaces, to support the low latency of flash accesses and data transfer delays, respectively [19]. Table 1 summarizes the device-level characteristics of three different state-of-the-art 3D flash technologies: i) Bit Cost Scaling (BiCS) 3D flash

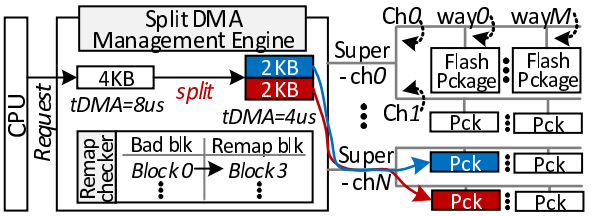
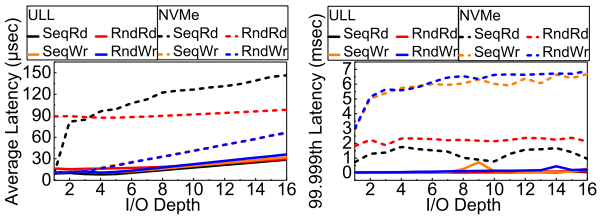


Figure 1: ULL SSD internals and Split DMA.

[25] ii) Vertically stacked (V-NAND) 3D flash [16] and iii) ULL-tailored flash (Z-NAND) [3]. Z-NAND uses 48 stacked word-line layer, which offers 3μs and 100μs latencies for a read operation and a write operation, respectively. The write latency of Z-NAND is shorter than that of BiCS and V-NAND by 6.6 and 7 times, respectively, while its read latency is 15∼20 times faster than those of such two modern 3D flash technologies. Even though the storage capacity and page size of Z-NAND are smaller than those of BiCS/V-NAND, ULL SSDs can offer a bigger storage volume with shorter latency by putting more Z-NAND packages into their device as a scale-out solution.

Split-DMA and super-channel architectures. The storage architecture of high-end SSDs consists of multiple system buses, referred to as *channel*, each employing many flash packages via multiple datapaths called *way* [4]. Similarly, ULL SSDs adopt this multi-channel and multi-way architecture, but further optimizes the datapaths and its channel-level striping method. As described in Table 1, the basic I/O unit of Z-NAND (i.e., page) is much smaller than that of other flash technologies, which in turn can enable device-level and flash-level parallelism techniques [12] to serve a host request with finer granule operations. Specifically, ULL SSDs split a 4KB-sized host request into two operations and issue them to two different channel simultaneously, as shown in Figure 1. These two channels always handle flash transactions together as a pair of system buses, which is referred to as a *super-channel*. To efficiently manage data transfers and flash transactions upon the super-channel architecture, ULL SSDs exploit a circuit that automatically adjusts data-flow and manages a timing skew on individual channels [3]. This circuit, called, split-DMA management engine, can reduce the read access time thereby tailoring ULL further. One of the concerns behind the super-channel architecture is to manage wear-out blocks (i.e., bad blocks) in different channels. Since two operations spread across a pair of system buses (within a super-channel), flash firmware can waste the storage space if the bad blocks appear in a specific channel. To address these challenges, the split-DMA management engine also employs a remap checker to automatically remap the physical block addresses between a bad block and a clean block. The remap checker exposes this semi-



(a) Average latency. (b) 99.999th latency.

Figure 2: Latency analysis of NVMe SSD and ULL SSD.

virtual address space to the flash firmware, such that the storage space can be fully utilized with the super-channel technique.

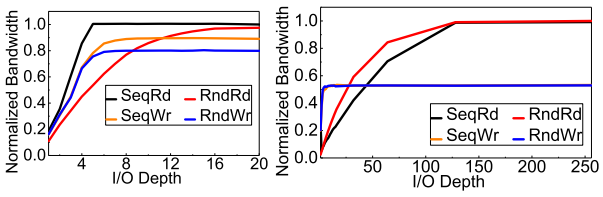
Suspend/resume support. To reduce the read latency more, Z-SSDs also apply a suspend/resume DMA technique [3]. While the split-DMA management engine can take advantages of shorter latency with a finer granule data access, it may not be able to immediately serve a read if a super-channel is busy to transfer the data associated with to a write, which was issued in an earlier time. This resource conflict can increase the latency of such read, which waits for a service in a device-level pending queue. The suspend/resume DMA method of ULL SSDs pauses a write service in progress for the super-channel that the read targets by storing an exact point of the flash write operation into a controller-side small buffer. The suspend/resume DMA technique then issues a read command to its target flash package without waiting for resolving the resource conflict. Once the read has been successfully committed, which takes only a few cycles, the engine resumes the write again by restoring the context (e.g., the stored write point). Thus, this suspend/resume mechanism can maximize resource utilizations of multi-way of super-channels and reduce overall latency of ULL SSDs, thereby satisfying diverse levels of QoS and SLA.

3 Performance Characterizations

3.1 Methodology

Benchmarks. To characterize NVMe and ULL SSDs, we use a Flexible I/O Tester (FIO v2.99) as our microbenchmark suite[1]. We set a O_DIRECT flag for all evaluations to bypass page caches and directly serve I/O requests to/from the underlying SSDs. In this test, we also use Linux native AIO (libaio) [2] as an I/O engine to generate asynchronous block I/O requests. Even though we test the SSDs with different block I/O sizes, ranging from 4KB to 32KB, for specific evaluations such as performance analysis and CPU utilization, we configure the default block size as 4KB. On the other hand, synchronous preadv2/pwritev2 (pvsync2) is enabled by the I/O engine to analyze the system impacts brought by different types of I/O completion methods.

Device configuration and profilers. We use a testbed that employs a 4 GHz, 4-core Intel i7 processor (i7-4790K) and 16GB DDR4 DRAM. All OS modules and executables are stored and executed from external 1TB HDD [21]. We evaluate an NVMe SSD by using Intel 750 SSD [10] as it is only the commercially available device that uses PCIe interface (without a M.2 bridge) with a standard NVMe protocol. We evaluate a 800GB Z-SSD prototype as an ULL SSD. In this test, all the SSDs are connected to the host of the testbed via PCIe 4x 3.0 lanes. For this study, we use Ubuntu 16.04 LTS and Linux kernel 4.14.10, which contains the most recent and stable version of NVMe storage stack. Lastly, we use collectd (5.5.1 [6]) for the analysis of CPU cycles, and Intel Vtune Amplifier 2018 [27] is used for measuring memory bounds that the NVMe storage stack exhibits.



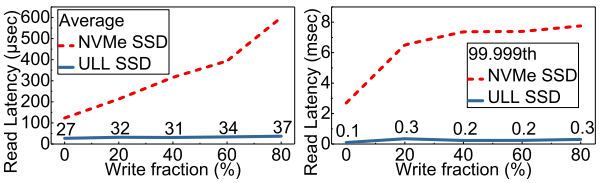
(a) ULL SSD. (b) NVMe SSD.

Figure 3: Bandwidth analysis (normalized to max B/W).

3.2 Performance analysis

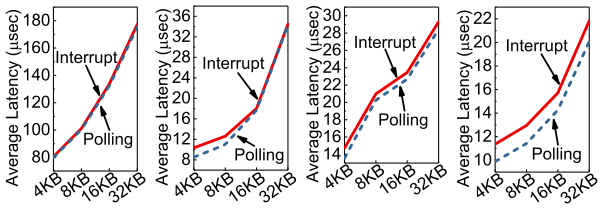
Overall latency. Figure 2a shows overall latency characteristics of ULL SSD and NVMe SSD with varying I/O depths, ranging from 1 to 16. With a small number of I/O queues (1∼4), the latency of NVMe SSD for write evaluations is around 13μs, which is slightly worse than that of ULL SSD; ULL SSD offers 12.6μs and 11.4μs for reads and writes, respectively. The reason why NVMe SSD can provide much shorter write latency than actual flash execution time is to cache and/or buffer the data over their large size of internal DRAM. However, NVMe SSD cannot hide the long latency imposed by low-level flash in cases of random reads (88.5μs), which is 4.6 times slower than ULL SSD (16.1μs). This is because the low locality of random reads enforces the internal cache to frequently access the underlying flash media, which makes NVMe SSD expose actual flash performance to the host. Similarly, as the queue depth increases, the execution time characteristics of NVMe SSD significantly get worse, and its latency increases as high as 66.6μs and 98.4μs for random writes and reads, respectively. In contrast, ULL SSD provides reasonably sustainable performance even in the test with a large number of I/O queues.

Long-tail latency. This performance difference between NVMe SSD and ULL SSD is more notable when we examine their long tail latency characteristics. Figure 2b analyzes five-nines (99.999%) latency for NVMe SSD and ULL SSD. The results of five-nines latency evaluation show the pure performance of each SSD's low-level flash and characteristics by removing the latency



(a) Average latency. (b) 99.999th latency.

Figure 4: I/O interference analysis.



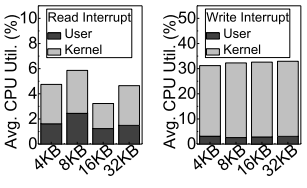
(a) NVMe SSD (reads). (b) NVMe SSD (writes). (c) ULL SSD (reads). (d) ULL SSD (writes).

Figure 5: Latency comparison (interrupt vs. polling).

hiding impacts brought by internal DRAM cache/buffer. For example, even though the average latency of random/sequential writes is better than reads, in this long tail latency evaluation, writes are worse than even random reads by 2.6×, on average. More specifically, NVMe SSD increases five-nines latency of reads and writes than the average latency of reads and writes by 29767 times and 167214 times, on average, respectively. The reason behind this significant performance degradation is that most of the NVMe SSD's architectural supports cannot take an advantage for the five-nines latency due to many systemic challenges such as resource conflicts, insufficient internal buffer size to accommodate all incoming I/O requests, and heavy internal tasks (e.g., garbage collection). In contrast, ULL SSD takes much shorter latencies ranging from a few μs to hundreds μs for both reads and writes. In contrast to NVMe SSD, the backend flash media of ULL SSD offers ultra-low latency by changing memory design and device-level tailoring. This low-level optimizations and Z-NAND characteristics not only reduce the latency of I/O accesses that head to the underlying flash but also offer a better opportunity to maximize the efficiency of multi-channel and multi-way architectural support.

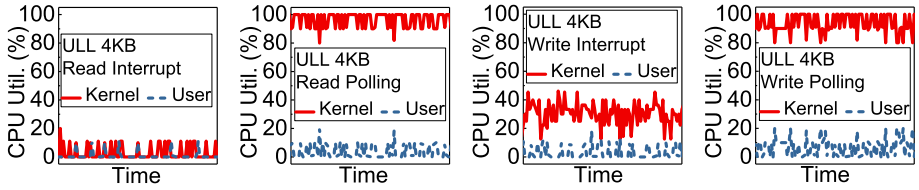
Queue analysis. Figure 3 shows bandwidth utilizations of NVMe SSD and ULL SSD with varying I/O depths. While one can expect that the level of parallelism can increase (thereby higher bandwidth) as the queue depth increases, the performance of NVMe SSD cannot reach the maximum performance by scheduling many 4KB-sized I/O requests. Specifically, NVMe SSD only utilizes their bandwidth compared with the total performance capacity by 53% for writes. Interestingly, unlike the previous read latency evaluations, the NVMe bandwidth of random reads with a higher queue depth (more than 128) outperforms that of all other I/O patterns. This is because, with more I/O requests scheduled in the queue, SSDs can easily find out a set of flash media that can simultaneously serve multiple requests (by spreading the requests across different flash dies in parallel). Thus, random and sequential reads of NVMe SSD can offer the maximum bandwidth (1.8GB/s). In contrast, the bandwidth utilization of ULL SSD bumps against the maximum bandwidth for all read scenarios, and even for sequential and random writes, ULL SSD utilizes the bandwidth by 89% and 80%, on average, respectively. Importantly, ULL SSD needs only 6 queue entries for sequential accesses, and even in the worst case, 16 queue entries are sufficient to achieve the peak bandwidth of ULL SSD. We believe that the rich queue mechanism and software-based protocol management of NVMe (which are managed by an NVMe driver and/or Linux blk-mq) are very reasonable design strategies to maximize modern high performance SSDs as the SSDs require many I/O resources for higher parallelism. However, once the latency gets shorter with new flash technology (like ULL SSD), we believe that the rich queue and protocol specification are overkilled; future systems may require to have a lighter queue mechanism and simpler protocol such as NCQ [9] of SATA [22].

I/O Interference impact. Figure 4 analyzes the degree of I/O interference when reads and writes are intermixed. For this analysis, we randomly read data from NVMe SSD and ULL SSD by sporadically issuing writes between the read requests. In addition, we increase the write fractions of total I/O executions, ranging from 20% to 80% in an attempt to see different I/O interference behaviors. As shown in Figure 4a, the average read latency of NVMe SSD linearly increases as the write fraction in intermixed workloads increases. When 20% writes are interleaved with reads, they seriously interfere the I/O service of reads, which can make the read latency worse than read only execution by 1.7 times (90μs). There are two root causes. First, a write operation of a conventional flash (at memory-level) takes a significantly longer time than that of a read operation (26× at most), and it blocks the subsequent read services. Second, the data transfers for writes (4KB) also occupy a specific channel for around 660μs, which prevent issuing a read command to the target memory over that channel. In contrast, we observe that ULL SSD exhibits almost sustainable latency irrespective of varying write operations interleaved with reads. These I/O interference characteristics are also captured by in our five-nines latency analysis. As shown in Figure 4b, while the five-nines read latency of NVMe SSD increases as high as 6.5ms even with 20% sporadic writes, ULL-SSD maintains it under 350μs. Since modern file systems and OS kernels require to periodically write metadata or perform journaling, even if a user application only intensively reads a chunk of data, we be-



(a) Read. (b) Write.

Figure 6: CPU utilization.



(a) Read w/ interrupt. (b) Read w/ polling. (c) Write w/ interrupt. (d) Write w/ polling.

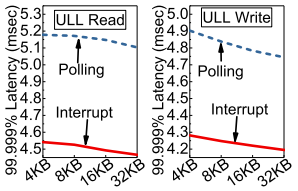
Figure 7: Time series analysis of CPU utilization under 4KB requests.

lieve that ULL SSD is a more desirable solution in many data-intensive solutions.

4 I/O Completion Methods and Challenges

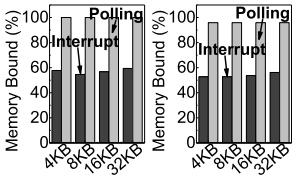
Overall latency comparison. Figure 5a and 5b show the latency difference of NVMe SSD when polling-based and interrupt-based I/O services are applied in the current NVMe storage stack. While polling gets attention from industry and academia as a promising solution to expose the true latency of PCIe based SSDs, one can observe from the figure that polling has no performance impact with the modern high-end SSD technology. Specifically, the latency difference of reads and writes brought by the interrupt-based and polling-based I/O services is less than 0.9% and 8.2%, on average, respectively. In contrast, as shown in Figure 5c and 5d, polling is better than interrupt in cases of ULL SSD; the read and write latency of ULL SSD is respectively 13.6 μs and 9.9 μs (for 4KB-sized requests), while that of interrupt is 14.7 μs and 11.4 μs, respectively. Even though polling shortens the read and write latency by only 7.5% and 13.2% on average, respectively, we believe that the polling mechanism in the NVMe stack can bring an advantage for latency sensitive applications, and the benefit will be more notable with future SSDs that employ further memory-level design changes or different materials such as resistive random access memory (ReRAM). To appropriately exploit the polling method in future systems, we believe that there exists several system-level challenges that the conventional NVMe storage stack should address; we will analyze the challenges shortly.

CPU utilization analysis. Figures 6a and 6b compare the CPU utilizations of two systems, each employing the interrupt-based and polling-based I/O services, respectively. In addition, Figure 7 shows the dynamics behind such CPU utilization analyses in details. For this evaluation, we measure the CPU usages by randomly reading 4KB-sized data from ULL SSD with those two different I/O completion methods. As shown in the figures, the interrupt-based I/O services only take 3.2% and 1.5% of total CPU cycles for kernel mode and user mode, on average, respectively. However, the polling-based I/O services show a completely different trend on the CPU utilization analysis. While the CPU cycles consumed by polling at the user-level are similar to those of interrupt, such CPU cycles (of polling) at the kernel-level accounts



(a) Read. (b) Write.

Figure 8: 99.999th latency of ULL SSD.



(a) Read. (b) Write.

Figure 9: Memory bound analysis.

for 97% of the entire application execution, on average. This is because the NVMe driver that implements the polling service does not relax the CPU and keeps iterating looking up NVMe's completion queue (CQ) head/tail pointers and checking all phase tags within the CQ's entries. For writes, interrupt consumes around 25% CPU cycles of the total execution at kernel level. The reason why the interrupt I/O services on writes require more computations than the services on reads is that, as the latency of writes at a device-level is longer than its read latency, the CPU is released from a running task, but used for other I/O operations such as queue aggregation and management. On the other hand, the polling-based I/O services on writes consume all CPU cycles (almost 100%) to check up the head/tail pointers and phase tag information of CQ entries. We believe that, even though polling can shorten the device-level latency, allocating an entire core to poll the I/O completions can hurt the overall system performance as it prevents several computational tasks from running on the host system.

Memory requirements. Figure 9 compares the amount of memory instructions generated from the interrupt-based and polling-based I/O services, called *memory bound*. The interrupt-based I/O services generate memory accesses around 57% and 54% for reads and writes, respectively, which is reasonable by considering the fact that the NVMe stack contains multiple NVMe commands and data in its system memory. However, the load and store memory instructions of polling, respectively, take account of 100% and 96% of total executed instructions for I/O services, which are higher than those of interrupt by 43% and 42%, respectively. There are two reasons behind this high memory overhead. First, the polling method implemented in the NVMe storage stack requires checking up all CQ entries as it needs to synchronize the head and tail pointers with the underly-

ing ULL SSD for each polling iteration. Second, during the execution of polling method, the data consistency should be also maintained. Thus, when polling refers and changes the CQ status, the system should use spin locks for every polling operation. These two behaviors introduce many memory instructions, which significantly consume meaningless CPU cycles (as described earlier) and waste energy on working memory accesses. We believe that these memory access patterns of polling should be overhauled in the future NVMe storage stack. For example, the number of spin lock accesses can be reduced by introducing a shared-nothing structure or selectively using polling-based on I/O access patterns in the future.

Five-nines latency. Figures 9a and 9b show five-nines latency for reads and writes with two different I/O completion methods, respectively. In contrast to the previous observations, the long tail latency of polling for reads and writes are worse than those of interrupt by 12.5% and 11.4%, on average, respectively, which should be also addressed in the future NVMe storage stack. This is because the key functions of polling (cf. `nvme_poll` and `blk_poll`) require acquiring spin locks when they process CQ(s), which will be iterated until the target request is completed. Specifically, as spin locks are used for polling NVMe queues, polling does not release and relax CPU with a high possibility. This prevents other I/O requests from a service, and unfortunately, polling holds everything without a context switch until the I/O completion reported. Even though the worst-case latency exhibited by the internal tasks of ULL SSD, such as garbage collection, is shorter than that of other high-end SSDs, such latency is still significantly longer than normal operation latency, which in turn makes polling a critical performance bottleneck in five-nine latency analysis.

5 Conclusion

We analyzed the performance behaviors of ULL SSDs and brought several system-level challenges of the current storage stack to take full advantages of ULL devices. Specifically, we observed that ULL SSDs can offer their maximum bandwidth with only a few queue entries, which contradicts with the design direction of the current rich NVMe queue. While it is beneficial to employ a polling-based I/O completion routine for ULL SSDs, system-level overheads delivered by polling, such as high CPU cycles and frequent memory accesses, incur many CPU stalls and consume more system power than those of interrupt-based storage systems.

6 Acknowledgement

The authors thank Samsung's Jaeheon Jeong (President), Jongyoul Lee (Vice President), Se-Jeong Jang and Joo-Young Hwang for their engineering sample donations and technical support. This paper is mainly supported by Yonsei Future Research Grant (2017-22-0105) and NRF 2016R1C1B2015312. The work is also supported in part by DOE DEAC02-05CH11231, IITP-2018-2017-0-01015, NRF2015M3C4A7065645. S. Koh and C. Lee equally contribute to this work. M. Jung is the corresponding author.

References

[1] AXBOE, J. Flexible io tester. *https://github.com/axboe/fio* (2017).

[2] BHATTACHARYA, S. Linux asynchronous i/o design: Evolution & challenges.

[3] CHEONG, W., ET AL. A flash memory controller for 15μs ultra-low-latency ssd using highspeed 3d nand flash with 3μs read time. *ISSCC*.

[4] ESHGHI, K., AND MICHELONI, R. Ssd architecture and pci express interface. In *Inside solid state drives (SSDs)*. 2013.

[5] FADU, INC. FADU. *http://www.fadu.io/* (2017).

[6] FORSTER, F., AND HARL, S. collectd–the system statistics collection daemon, 2016.

[7] HICKEN, M. S., HOWE, S. M., SOKOLOV, D. J., SWATOSH, T., AND WILLIAMS, J. L. Disk drive cache system using a dynamic priority sequential stream of data segments continuously adapted according to prefetched sequential random, and repeating types of accesses, 2000. US Patent 6,092,149.

[8] HSU, S. T., AND ZHUANG, W.-W. Electrically programmable resistance cross point memory, 2003. US Patent 6,531,371.

[9] HUFFMAN, A., AND CLARK, J. Serial ata native command queuing. *Intel Corporation and Seagate Technology, Whitepaper* (2003).

[10] INTEL CORPORATION. Intel SSD 750 Series. *https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/750-series.html* (2015).

[11] JUNG, M. Exploring design challenges in getting solid state drives closer to cpu. *IEEE Transactions on Computers* (2016).

[12] JUNG, M. Exploring parallel data access methods in emerging non-volatile memory systems. *TPDS* (2017).

[13] JUNG, M., AND KANDEMIR, M. Revisiting widely-held expectations of ssd and rethinking implications for systems. *SIGMETRICS* (2013).

[14] JUNG, M., AND KANDEMIR, M. T. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *HPCA* (2014).

[15] KANG, Y., KEE, Y.-S., MILLER, E. L., AND PARK, C. Enabling cost-effective data processing with smart ssd. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on* (2013).

[16] KIM, C., KIM, D.-H., JEONG, W., KIM, H.-J., PARK, I. H., PARK, H.-W., LEE, J., PARK, J., AHN, Y.-L., LEE, J. Y., ET AL. A 512-gb 3-b/cell 64-stacked wl 3-d-nand flash memory. *JSSC*.

[17] LIU, N., COPE, J., CARNS, P., CAROTHERS, C., ROSS, R., GRIDER, G., CRUME, A., AND MALTZAHN, C. On the role of burst buffers in leadership-class storage systems. In *MSST* (2012).

[18] OH, Y., LEE, E., HYUN, C., CHOI, J., LEE, D., AND NOH, S. H. Enabling cost-effective flash based caching with an array of commodity ssds. In *MIDDLEWARE* (2015).

[19] SAMSUNG. Advancements in ssds and 3d nand reshaping storage market. In *Flash Memory Summit* (2017).

[20] SAMSUNG ELECTRONICS CO., LTD. PM1725 NVMe PCIe SSD. *http://www.samsung.com/semiconductor/insights/tech-leadership/pm1725-nvme-pcie-ssd/* (2015).

[21] SEAGATE TECHNOLOGY LLC. Seagate Barracuda ST1000DM003. *https://www.seagate.com/staticfiles/docs/pdf/datasheet/disc/barracuda-ds1737-1-1111us.pdf* (2011).

[22] SERIAL, A. International organization. *Serial ATA Revision 3* (2007).

[23] SESHADRI, S., GAHAGAN, M., BHASKARAN, M. S., BUNKER, T., DE, A., JIN, Y., LIU, Y., AND SWANSON, S. Willow: A user-programmable ssd. In *OSDI* (2014).

[24] WON, Y., JUNG, J., CHOI, G., OH, J., SON, S., HWANG, J., AND CHO, S. Barrier-enabled io stack for flash storage. In *FAST* (2018).

[25] YAMASHITA, R., MAGIA, S., HIGUCHI, T., YONEYA, K., YAMAMURA, T., MIZUKOSHI, H., ZAITSU, S., YAMASHITA, M., TOYAMA, S., KAMAE, N., ET AL. 11.1 a 512gb 3b/cell flash memory on 64-word-line-layer bics technology. In *ISSCC*.

[26] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).

[27] ZONE, I. D. Intel vtune amplifier, 2018. *Documentation at the URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe*.