

# When Poll is More Energy Efficient than Interrupt

Bryan Harris and Nihat Altiparmak

Dept. of Computer Science & Engineering

University of Louisville

{bryan.harris.1,nihat.altiparmak}@louisville.edu

## ABSTRACT

Polling is commonly indicated to be a more suitable IO completion mechanism than interrupt for ultra-low latency storage devices. However, polling's impact on overall energy efficiency has not been thoroughly investigated. In this paper, contrary to common belief, we show that polling can also be more energy efficient than interrupt. To do so, we systematically investigate the energy efficiency of all available Linux IO completion mechanisms, including interrupt, classic polling, and hybrid polling using a real ultra-low latency storage device, a power meter, and various workload behaviors. Our experimental results indicate that although hybrid polling provides a good trade-off in CPU utilization, it is the least energy efficient, whereas classic polling is the most energy efficient for low latency IO requests. To the best of our knowledge, this is the first paper classifying polling as more energy efficient than interrupt for a real secondary storage device, and we hope that our observations will lead to more energy efficient IO completion mechanisms for new generation storage device characteristics.

## CCS CONCEPTS

• **Information systems** → **Storage power management**;  
**Storage class memory**.

## KEYWORDS

IO completion, energy efficiency

### ACM Reference Format:

Bryan Harris and Nihat Altiparmak. 2022. When Poll is More Energy Efficient than Interrupt. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22)*, June 27–28, 2022, Virtual Event, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3538643.3539747>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*HotStorage '22, June 27–28, 2022, Virtual Event, USA*

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9399-7/22/06...\$15.00

<https://doi.org/10.1145/3538643.3539747>

## 1 INTRODUCTION

With the most recent advancements in data storage technology, a new category of Solid-State Drives (SSDs) have emerged. These devices are referred to as Ultra-Low Latency (ULL) SSDs and are broadly classified as providing data access in less than 10  $\mu$ s [17]. Various vendors including Intel, Samsung, and Toshiba have representative ULL SSDs [3, 4, 20], where Intel's latest generation of the Optane SSD is advertised to deliver read IO in 5  $\mu$ s and write IO in 6  $\mu$ s [6]. ULL IO performance providing sub-10  $\mu$ s data access latency renders the performance of traditional, interrupt-based IO completion mechanism questionable. Both industry and academia suggested replacing interrupts with polling based IO completion methods for improved latency in such devices [11, 13, 15, 19, 22, 25–27], where polling has also been supported by the Linux kernel since version 4.4. However, one must also consider the relationship between IO performance and power consumption, as power saving methods may not be worth the resulting loss in IO performance.

Despite greater performance, polling is commonly believed to be more costly and less energy efficient than interrupt since polling wastes CPU cycles. The primary assumption behind this is that reduced CPU usage directly correlates to reduced power consumption. Therefore, with kernel version 4.10, Linux introduced a hybrid polling mechanism, which sleeps the task before starting to poll so that less CPU cycles are wasted [13].

In this paper, we study the energy implications of the three IO completion mechanisms available in Linux, including interrupts, classic polling, and hybrid polling techniques, specifically for ULL disk IO. Our empirical evaluation using a real ULL device, a power meter, various workload behaviors, and the most recent longterm Linux kernel relies on IO performance measured per energy unit, bytes transferred per joule. Considering both performance and energy in a single metric, we make observations laying out the most energy efficient IO completion mechanisms. We hope that our observations and analysis can lead to more energy efficient storage stack designs in the future.

## 2 IO COMPLETION IN LINUX KERNEL

In this section, we outline the working mechanisms of available Linux IO completion mechanisms for the most commonly used Linux-native synchronous IO interface.

## 2.1 Interrupt

When a storage device completes a request, its controller raises an interrupt by making a request on an interrupt request line (IRQ), which is caught by the CPU. The CPU performs a context switch, saving its current state, and jumps to an interrupt handler routine listed in a vector table. After completion of the handler, the CPU clears the interrupt and restores itself to the stored state, resuming its previous operation. Use of interrupts is preferable for high IO latency (such as HDDs) since it allows other work to be performed while waiting. However, the overhead of switching and handling interrupts can be significant if the interrupt occurs too quickly after the request. If the rate of interrupts is exceedingly high, it can even overload the system (livelock) causing more damage than benefit [7, 14].

## 2.2 Classic Polling

In *classic polling*, the kernel continuously queries the device's completion queues for completed requests without switching tasks and handles completions immediately. Block IO polling has been supported since Linux 4.4. There are numerous other performance benefits to avoiding context switching, such as not polluting hardware and memory caches (TLB), reducing CPU power state transitions, and reduced interrupt handling overhead. This results in not only lower latency for individual IOs, but overall greater IO throughput [25]. However, continuously polling for IO completion uses 100% of a core, which puts significant pressure on CPU resources.

## 2.3 Hybrid Polling

Instead of continuously polling for IO completion, *hybrid polling* (available since Linux 4.10) sleeps the task before starting to poll [13]. This has two modes: *fixed*, in which a user-specified sleep time (with microsecond resolution) can be assigned for all polled requests, and *automatic*, where the kernel sleeps for half the mean device service time. The block layer maintains statistics on service times separately for eight request sizes (512 B through 64 KB) and read/write types, collected once over a 100 ms period, triggered by the first IO request. For example, if the IO is expected to complete in 8  $\mu$ s, the kernel sleeps for 4  $\mu$ s before polling [2].

## 3 ENERGY EFFICIENT IO COMPLETION

In this section, we first demonstrate how classic and hybrid polling achieve their design goals, then we look into the additional costs associated with polling and interrupt, and finally present our energy efficiency analysis.

### 3.1 Methodology and Experimental Setup

Our test system is a Dell PowerEdge R230 with a single-socket Intel Xeon E3-1230 v5 quad-core CPU (3.4 GHz) and 64 GB RAM. Our test storage device is an Intel Optane 900P

SSD, a PCIe 3 ULL SSD capable of sub-10  $\mu$ s read latency. Our system also contains a Dell-certified flash SSD used only as the OS disk, and not used for IO workloads in the experiments. We used Ubuntu 20.04 and upgraded the Linux kernel to version 5.15.16 (the latest longterm release). CPU C-states are enabled, as this is a basic power saving feature and the default for many systems and distributions.

In order to measure the power consumption of the entire system, we used an in-line power meter (Onset HOBO UX120-018 [21]) connected to the sole power supply of our test system. This power meter is entirely self contained; it measures voltage, current, power, etc., once a second and records to embedded memory. Only after experiments have completed is the data copied from the meter via USB. The power meter's real-time clock and host's system clock are synchronized to the same NTP server.

Our microbenchmark workloads use *fio*, the Flexible IO tester [10] version 3.29, to generate a constant stream of synchronous random reads using the *preadv2* function. This is the only API through the traditional IO path that supports completion polling, which requires using the RWF\_HIPRI and O\_DIRECT flags, thus avoiding the page cache. We used the *ext4* file system with the "*none*" IO scheduler, the default for many distributions. We ran an array of microbenchmarks similar to other authors [14, 24] using request sizes from 1 KB through 128 KB and a number of threads from 1 through 64. Since we used synchronous requests, the number of threads is also the total number of outstanding requests (IO depth).

In order to record a sufficient number of power measurements from the power meter, we ran each workload for a fixed two minutes. In order to avoid any initial effects such as caching, we used a warm up ("ramp time") of 10 seconds. All results presented are the average of 10 replicates.

### 3.2 Goals of Polling

First, we verify that classic and hybrid polling achieve their intended design goals of improved performance over interrupts and, for hybrid polling, reduced CPU utilization over classic polling.

**Observation 1.** *Both classic and hybrid polling show improved performance over interrupts.*

The motivation of using polling-based IO completion for ULL IO is to achieve improved performance over interrupts [25]. Our first observation confirms that both classic and hybrid polling achieve this goal with our Optane SSD. Table 1 compares the average latency and throughput (IOPS) using 4 KB random read workloads. For a single thread, classic polling has the lowest latency of 8.1  $\mu$ s, a 2- $\mu$ s improvement over interrupts. Hybrid polling is not far behind, with only a slight 0.4  $\mu$ s latency cost over classic polling. This improved latency for both polling methods corresponds to

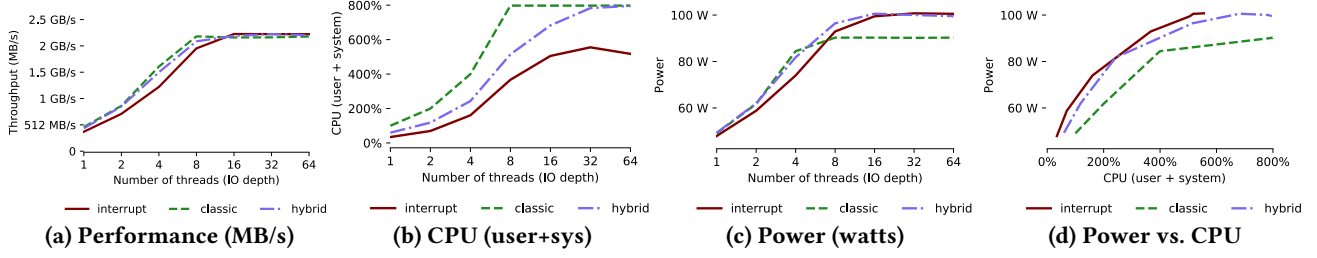


Figure 1: Performance and basic cost comparison

Table 1: Basic performance comparison (4 KB reads)

IO Completion	1 Thread		8 Threads	
	Latency	IOPS	Latency	IOPS
Classic polling	8.1 $\mu$ s	120,470	13.6 $\mu$ s	571,659
Hybrid polling	8.5 $\mu$ s	113,846	13.9 $\mu$ s	546,968
Interrupt	10.1 $\mu$ s	96,722	15.0 $\mu$ s	511,895

an increased throughput over interrupts of roughly 24K IOPS and 17K IOPS more for classic and hybrid polling, respectively. Even at the saturation point of eight threads these improvements remain. Figure 1a further illustrates these performance trends with bandwidth (MB/s), where both polling methods outperform interrupts until device saturation.

**Observation 2.** *Hybrid polling uses more CPU than interrupts and less than classic polling.*

Although classic polling outperforms interrupts, it comes at an obvious cost to CPU usage. The motivation of hybrid polling is to use less CPU than classic polling by sleeping for half the expected latency before polling begins [13]. We directly observed this reduction in CPU usage, as shown in Figure 1b. As intended, hybrid consistently uses less CPU than classic polling, but this usage is still higher than interrupts. Polling for half the latency still requires more CPU than sleeping for the entire latency using an interrupt. However, as we show next, hybrid polling has other costs in addition to CPU usage.

### 3.3 Costs of Polling

Here we observe the costs of polling, first in terms of power consumption and then we investigate additional costs.

**Observation 3.** *CPU utilization does not directly correspond to power consumption of the entire system.*

Figure 1c shows the overall power consumption of our test system for the three IO completion methods. If one assumes that power consumption is directly related to CPU usage (Fig. 1b), then he would expect power (Fig. 1c) to show similar trends. Notice that hybrid polling has a surprisingly high power consumption. Unlike its CPU usage, hybrid polling’s power can exceed even that of classic polling. Moreover, classic polling can use even less power than interrupts.

These results clearly warn us against making quick assumptions about power consumption based solely on CPU usage. In order to outline this better, Figure 1d directly compares CPU usage (Fig. 1b) to power consumption (Fig. 1c). Power consumption of the three completion methods does not scale equally based on CPU usage, indicating that there must be other factors affected by the choice of completion method that also contribute significantly to power. In other words, simply reducing CPU usage does not necessarily correspond to power savings. Next, we explore the reasons behind this by looking at other system software costs.

**Observation 4.** *Hybrid polling triggers as many context switches as interrupts, while classic polling triggers none.*

Context switching refers to the switching of a processor from one task to another, and it is known to be a costly operation as it requires the execution of the CPU scheduler, storing the CPU context to memory for the task that is being scheduled-out, loading the CPU context from memory for the task that is being scheduled-in, and flushing TLB entries [18]. Using `getrusage(2)`, `fio` records the number of context switches for its worker threads. Figure 2a shows the rate of context switches as a function of IO depth. First, we can notice that classic polling has no noticeable context switch cost. On the other hand, both hybrid polling and interrupts have context switch rates directly correlated to their throughput. If we divide Fig. 2a by IOPS, we have the average number of context switches *per IO*, shown in Figure 2b. This clearly indicates that both hybrid polling and interrupts have, on average, one context switch per IO. Next, we observe the costs associated with these context switches for hybrid polling as increased load & store operations.

**Observation 5.** *Hybrid polling has a high cost in load & store operations associated with context switches.*

While interrupt has its cost of context switching and classic polling has its cost of polling, hybrid polling has the combined costs of both. Here we measure the number of load/store operations, and further attribute them to high-level operations in the kernel. Polling requires continuously checking for completion through memory-mapped device controller registers. Context switching requires loads and stores by the CPU scheduler and the dispatcher as contexts

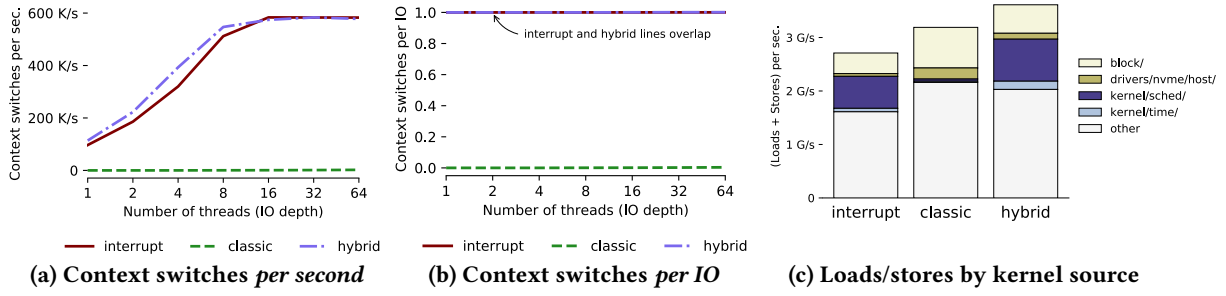


Figure 2: Additional costs of polling

are switched in and out of processors and the scheduler’s data structures are modified. If these memory values do not change frequently, most of these load/store operations will be handled by the CPU cache rather than DRAM. Even if polling or context switching do not require continuous DRAM operations, they can add considerable cost to the CPU die that is not accounted for by process CPU utilization alone. In order to measure these costs, we recorded the number of loads and stores using Intel’s VTune profiler [5] while running our *fio* workloads. Furthermore, we attribute loads/stores to specific portions of the kernel by looking at per-function information from the profiler. However, since the Linux kernel contains thousands of source files with thousands of functions, we map function names to source files using *cscope* [1], and aggregate functions into sections based on the directory of the kernel source tree in which they are defined: `block/` is the block layer, `drivers/nvme/host/` is the NVMe driver, `kernel/sched/` is the CPU scheduler, etc.

Let us examine in depth a case of 4 KB reads after saturation (32 threads), shown in Table 2. Values shown are sums of the number of load and store operations, averaged over the two-minute workload and rounded to the nearest million per second. To emphasize specific operations in further detail, we share a breakdown into selected functions below the source directory grouping. We illustrate the main sections of Table 2 with Figure 2c.

First, let us look at the cost of *polling*; notice that the cost from the block layer (`block/`) is lowest for interrupts, highest for classic, and with hybrid roughly halfway between the two. The most significant individual function in this section is `blk_poll`, which performs the polling for IO completion. As one may expect, hybrid polling performs half the `blk_poll` accesses as classic since it sleeps for half the expected device latency. The `blk_poll` function internally calls `nvme_poll` of the NVMe driver to query the completion queues, and so we also see similar contributions to the cost from `drivers/nvme/host/`, with more accesses for classic than hybrid.

To approximate the cost from the block multiqueue subsystem, we group the 15 functions with the prefix `blk_mq_` in the `block/` section. Although both polling methods have greater cost than interrupts, there is little difference between

Table 2: Load/store operations (millions per second)

Source	Interrupt	Classic	Hybrid
<code>block/</code>	386	759	533
<code>blk_poll</code>	0	219	70
<code>blk_mq_*</code> (36 functions)	111	158	156
(109 other functions)	274	383	307
<code>drivers/nvme/host/</code>	51	207	111
<code>kernel/sched/</code>	598	41	786
<code>__schedule</code>	13	0	13
<code>psi_task_change</code>	11	0	13
<code>update_load_avg</code>	14	0	22
(285 other functions)	560	41	739
<code>kernel/time/</code>	65	22	154
<code>ktime_get</code>	10	14	22
<code>*hrtimer*</code> (33 functions)	0	1	100
(56 other functions)	55	8	33
(other sources)	1611	2160	2029
<b>Total</b>	<b>2710 M/s</b>	<b>3190 M/s</b>	<b>3612 M/s</b>

classic and hybrid from `blk-mq`. Greater differences come from *context switching*, visible from the CPU scheduler `kernel/sched/`. The breakdown by function in Table 2 shows the three most significant functions. Since classic polling never voluntarily gives up its processor core, loads/stores from the scheduler is minimal. On the other hand, interrupts and hybrid both voluntarily give up their processor, and so the scheduler must switch out their contexts, schedule another task, and switch them back in when either the interrupt occurs or hybrid’s sleeper timer expires, resulting in significantly more accesses. For interrupts and hybrid polling, this cost of switching is greater than their costs from the block layer. Furthermore, the cost of hybrid polling attributed to the scheduler (786 M/s) is comparable to the costs of polling in the block layer for classic (759 M/s). Hybrid has costs associated with both polling and scheduling. Although hybrid polling may use less CPU utilization by sleeping, the load/store costs of switching its context in and out of CPU can be significant and should not be overlooked.

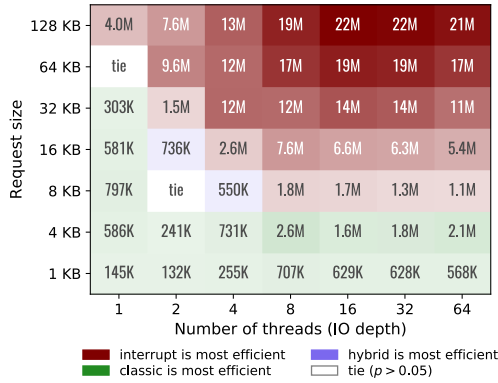
Hybrid polling uses a high resolution kernel timer (`hrtimer`) to sleep before polling, which further contributes to its load/store costs. We combine 33 functions with the substring “`hrtimer`” from `kernel/time/`, which is 100 M/s for

hybrid polling and negligible for interrupts and classic. While not as significant as other contributors, it appears to be an implementation choice that primarily affects hybrid polling.

In summary, although hybrid polling saves CPU utilization over classic polling (Obs. 2), it adds significantly to load/store operations due to the combined costs of polling and context switching. As illustrated in Figure 2c, the number of loads/stores from the task scheduler (kernel/sched/) and sleeper timer (kernel/time/) add significantly to the costs of hybrid polling over classic polling. Next, we directly compare power and performance to determine which of the three methods is the most energy efficient.

### 3.4 When Poll is More Energy Efficient

When analyzing energy efficiency, one must consider not only power consumption but also performance. We therefore use the ratio of performance (MB/s) to power (watts, or joules per second) as our metric for energy efficiency, which simplifies to *bytes per joule* (B/J).



**Figure 3: Most efficient completion method and difference in energy efficiency (B/J)**

Figure 3 shows the most energy efficient IO completion method for our tested range of request sizes and IO depths. The value in each tile is the difference in energy efficiency (bytes per joule) between the most efficient completion method and the next. If the top two methods have no significant difference ( $p > 0.05$ ) between the two sets of 10 replicates using an independent two-sample Student  $t$ -test, the tile is marked as a “tie.”

**Observation 6.** *Polling can be more energy efficient than interrupts.*

To our surprise, for requests with low IO latencies, we observed that classic polling is the most energy efficient method. This goes against the common assumption that continuously polling is more energy hungry than interrupts or hybrid polling. We believe that this observation can inform future IO completion designs for ULL disk IO. As it is clear from Fig. 3, the competition is generally between interrupts

and classic polling, motivating new hybrid IO completion mechanisms that can switch between classic polling and interrupts for better energy efficiency.

## 4 RELATED WORK

Existing energy efficiency research regarding solid-state storage has mainly focused on the design tradeoffs of flash SSDs for improved energy efficiency [12], the impact of SSD RAID configurations on server energy consumption [23], power consumption characteristics of ULL SSDs [16], and the impact of Optane SSDs on energy efficiency [14]. However, no previous work focused on system software’s impact on energy efficiency. From the performance perspective, within the last decade both industry and academia indicated the suitability of polling based IO completion methods for improved latency in ULL storage devices [11, 13, 15, 19, 22, 25–27]. Classic polling has been implemented in the Linux kernel since version 4.4 and hybrid polling since version 4.10 [13]. In addition, a selective polling/interrupt technique has also been proposed [27]. Finally, polling has also been included in alternative/custom IO interfaces such as *io\_uring* [8, 9] based on user/kernel space shared memory approach and the SPDK kernel-bypass system [26]. To the best of our knowledge, there is no previous work investigating the impact of IO completion on overall energy efficiency.

## 5 CONCLUSION AND DISCUSSION

We analyzed the energy efficiency of three IO completion mechanisms available in the Linux kernel for ULL IO: traditional interrupts, classic polling, and hybrid polling. Although both polling methods require more CPU utilization than interrupts, this does not necessarily correspond to proportionally more power consumption or less energy efficiency. In particular, hybrid polling inherits both the CPU cost of classic polling and the context switching cost of interrupt, making it the least energy efficient. Based on our experiments, classic polling is the most energy efficient for low-latency requests, and interrupt for high-latency requests. Although polling being more energy efficient than interrupt may be counter-intuitive, our result should encourage wider support for polling, as we expect ULL devices to have more widespread use and even lower latency performance. We would also like to caution against using CPU utilization alone as a shorthand for energy consumption determination, and instead encourage them to look at the power consumption of the entire system.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This research was supported in part by the U.S. National Science Foundation (NSF) under grants OIA-1849213 and CNS-2050925.



## REFERENCES

- [1] 2012. Cscope v15.9. <http://cscope.sourceforge.net/>
- [2] 2017. Hybrid Block Polling. [https://kernelnewbies.org/Linux\\_4.10#Hybrid\\_block\\_polling](https://kernelnewbies.org/Linux_4.10#Hybrid_block_polling)
- [3] 2017. Samsung SZ985 Z-NAND SSD. [https://www.samsung.com/us/labs/pdfs/collateral/Samsung\\_Z-NAND\\_Technology\\_Brief\\_v5.pdf](https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf).
- [4] 2018. Intel Optane SSD 900P Series Product Brief. <https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-900p-brief.pdf>.
- [5] 2020. Intel VTune Profiler. <https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html>
- [6] 2022. Intel Optane SSD DC P5800X Series. <https://ark.intel.com/content/www/us/en/ark/products/201840/intel-optane-ssd-dc-p5800x-series-3-2tb-2-5in-pcie-x4-3d-xpoint.html>.
- [7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces* (1.00 ed.). Arpaci-Dusseau Books.
- [8] Jens Axboe. 2019. Efficient IO through io\_uring. [https://kernel.dk/io\\_uring.pdf](https://kernel.dk/io_uring.pdf).
- [9] Jens Axboe. 2019. Faster IO through io\_uring. [https://kernel-recipes.org/en/2019/talks/faster-io-through-io\\_uring/](https://kernel-recipes.org/en/2019/talks/faster-io-through-io_uring/). Kernel Recipes, 2019.
- [10] Jens Axboe. 2022. *Flexible I/O Tester*. <https://github.com/axboe/fio>.
- [11] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*. IEEE Computer Society, USA, 385–395. <https://doi.org/10.1109/MICRO.2010.33>
- [12] Seokhei Cho, Changhyun Park, Youjip Won, Sooyong Kang, Jaehyuk Cha, Sungroh Yoon, and Jongmoo Choi. 2015. Design Tradeoffs of SSDs: From Energy Consumption's Perspective. *ACM Trans. Storage* 11, 2, Article 8 (March 2015), 24 pages. <https://doi.org/10.1145/2644818>.
- [13] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) (*EuroSys '18*). Association for Computing Machinery, New York, NY, USA, Article 42, 13 pages. <https://doi.org/10.1145/3190508.3190524>
- [14] Bryan Harris and Nihat Altıparmak. 2020. Ultra-Low Latency SSDs' Impact on Overall Energy Efficiency. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '20)*. USENIX Association.
- [15] Hyeon-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)*. USENIX Association, Denver, CO. <https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim>
- [16] S. Koh, J. Jang, C. Lee, M. Kwon, J. Zhang, and M. Jung. 2019. Faster than Flash: An In-Depth Study of System Challenges for Emerging Ultra-Low Latency SSDs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 216–227. <https://doi.org/10.1109/IISWC47752.2019.9042009>.
- [17] Gyun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2019. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Renton, WA, 603–616. <https://www.usenix.org/conference/atc19/presentation/lee-gyun>.
- [18] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*. 2–es.
- [19] Damien Le Moal. 2016. I/O Latency Optimization with Polling. [https://events.static.linuxfound.org/sites/events/files/slides/lemoal-nvme-polling-vault-2017-final\\_0.pdf](https://events.static.linuxfound.org/sites/events/files/slides/lemoal-nvme-polling-vault-2017-final_0.pdf).
- [20] Shigeo (Jeff) Ohshima. 2018. Scaling Flash Technology to Meet Application Demands. <https://flashmemorysummit.com/English/Conference/Keynotes.html>.
- [21] Onset 2021. *HOBOTM UX120-018 Data Logger (datasheet)*. Onset. <https://www.onsetcomp.com/datasheet/UX120-018>.
- [22] S. Swanson and A. M. Caulfield. 2013. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer* 46, 8 (2013), 52–59.
- [23] Erica Tomes and Nihat Altıparmak. 2017. A Comparative Study of HDD and SSD RAID's Impact on Server Energy Consumption. In *19th IEEE International Conference on Cluster Computing (CLUSTER 2017)*. Honolulu, Hawaii. <https://doi.org/10.1109/CLUSTER.2017.103>.
- [24] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '19)*. USENIX Association, Renton, WA. <https://www.usenix.org/conference/hotstorage19/presentation/wu-kan>.
- [25] Jisoo Yang, Dave B. Minter, and Frank Hady. 2012. When Poll is Better than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) (*FAST '12*). USENIX Association, USA, 3.
- [26] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161. <https://doi.org/10.1109/CloudCom.2017.14>.
- [27] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. 2018. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA, 477–492. <https://www.usenix.org/conference/osdi18/presentation/zhang>.