Energy Implications of IO Interface Design Choices

Sidharth Sundar, William Simpson, Jacob Higdon, Caeden Whitaker, Bryan Harris, Nihat Altiparmak
Dept. of Computer Science & Engineering, University of Louisville
{sidharth.sundar,william.simpson.3,jacob.higdon,caeden.whitaker,bryan.harris.1,nihat.altiparmak}@louisville.edu

ABSTRACT

With the availability of high performance storage technology, there is extra pressure on the efficiency of IO interfaces. In addition to the popular POSIX synchronous, POSIX asynchronous, and Linux asynchronous (*libaio*) IO interfaces, there are two recent interfaces, *spdk* and *io_uring*, that are increasingly attracting attention with their high performance asynchronous designs. While providing high performance IO is crucial, it is also essential to do so in an energy-aware manner. In this paper, we study the energy implications of IO interface design choices and how these choices impact a system's energy consumption. Our empirical evaluation using a power meter, an ultra-low latency storage device, and various workload behaviors including single and multiple thread scenarios allow us to lay out the most energy efficient design choices, with the goal of yielding energy-aware high-performance storage stack designs.

CCS CONCEPTS

• **Software and its engineering → Secondary storage**; • **Information systems → Storage power management**.

KEYWORDS

IO interface, energy efficiency

ACM Reference Format:
Sidharth Sundar, William Simpson, Jacob Higdon, Caeden Whitaker, Bryan Harris, Nihat Altiparmak. 2023. Energy Implications of IO Interface Design Choices. In *15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '23), July 9, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3599691.3603411

HotStorage '23, July 9, 2023, Boston, MA, USA

ACM ISBN 979-8-4007-0224-2/23/07…$15.00
https://doi.org/10.1145/3599691.3603411

Table 1: IO Interface Design Choices

API	Execution	Behavior	Submission	Completion
	kernel user	sync async	syscall spoll	int cpoll
posix-sio	● ○	● ○	● ○	● ●
posix-aio	● ●	○ ●	● ○	● ○
libaio	● ○	○ ●	● ○	● ○
io_uring	● ○	● ●	● ●	● ●
spdk	○ ●	○ ●	○ ○	○ ●

○ unsupported, ● supported

1 INTRODUCTION

IO interfaces allow user applications to issue IO requests to a secondary storage device. Various IO interface designs with different characteristics are available across many platforms; however, their characteristics are mainly shaped by the following four interrelated design choices listed in Table 1: (i) kernel vs. user space execution; (ii) synchronous vs. asynchronous behavior; (iii) system call vs. polling based IO submission; and (iv) interrupt vs. polling based IO completion. IO interface design choices are expected to have different impacts on system components such as CPU and memory, with unique energy implications. This impact generally depends on the number of interrupts, system calls, and context switches they trigger; the number of submission/completion polls they execute; the overhead of their runtime system, if any; and their unique usage of file system, page cache, IO scheduling, and IO driver functionalities.

While providing high performance disk IO is crucial, it also matters to do so in an energy-aware manner since today's data centers consume as much electricity as a city [8]. From embedded systems, mobile devices, and personal computers to servers and clusters used in data centers, IO interfaces are heavily used to enable disk IO in a variety of computer systems. However, efficiency of a storage stack is generally measured in terms of its performance, whereas its energy implications are commonly neglected. In this paper, we measure IO performance per unit energy, and study energy implications of IO interface design choices. Since low latency is the common driving factor of new interface design choices, we use an ultra-low latency (ULL) SSD in our experiments, allowing us to make further analysis for single-digit microsecond IO latencies.

Our empirical evaluation using a ULL SSD, a power meter, and various workload behaviors indicate that by eliminating the need for a system call to submit an IO request, IO submission polling mechanism reduces the performance

gap between kernel bypass and kernel-based interface designs to 1–2 microseconds. However, IO submission polling also yields worse energy efficiency in general, especially in multithreaded scenarios as it requires an additional core to be dedicated by the kernel for each thread. In addition, while a kernel bypass design achieves the best performance and energy efficiency for small requests, system call and interrupt-based kernel-space mechanisms generally yield better energy efficiency for larger requests due to excessive polling cost and inability to handle interrupts outside of the kernel. Finally, interrupt based IO completion is found to be crucial for achieving the best energy efficiency for larger request sizes, especially in multithreaded scenarios.

2 IO INTERFACE DESIGN CHOICES

***posix-sio* [3]** is the Linux-native synchronous IO interface, which includes the traditional POSIX standard *read*/*write* system calls utilizing (kernel) buffered IO by interfacing with the page cache. Various versions of *read*/*write* system calls exist, including *pread*/*pwrite* that accepts a file offset, *readv*/*writev* that performs reads/writes in a vector format using multiple buffers, and *preadv2*/*pwritev2* (a Linux-only extension) that also accepts flags to modify behavior, such as completion polling (cpoll). The common characteristics of all these interfaces are that they rely on a single system call and that they are blocking. In other words, they halt the execution of the requesting process until the IO is completed. Synchronous interfaces are generally easier to understand and use by programmers and are thus more popular. However, their blocking behavior can cause slowdowns for certain applications and high performance storage devices such as ULL SSDs [10, 11]. Asynchronous IO is an alternative to traditional synchronous IO, where rather than stalling a process, an IO request is submitted and the associated completion is received later, allowing the submitting process to continue execution in the meantime. There are multiple asynchronous IO interfaces available with different design choices, including *posix-aio* [2], *libaio* [1], *io_uring* [4], and *spdk* [15].

posix-aio is the POSIX standard asynchronous IO interface, where *aio_read*/*aio_write* are asynchronous analogs to the *read*/*write* system calls. However, *aio_read*/*aio_write* are not system calls, they are GNU C library (*glibc*) functions. *posix-aio* is actually a user space implementation of asynchronous IO, where *glibc* performs synchronous IO through *posix-sio* using a user space thread pool. An *aiocb* struct is used to store information about IOs, where requests can be set up to provide notification about completion by delivery of a signal. Alternatively, IO completion status can also be checked with calls to *aio_error*, and upon completion a call to *aio_return* will return the final IO completion status. The main advantages of *posix-aio* are its POSIX portability and usage of page cache. However, its user space runtime system for thread pool management is expected to have further performance and energy implications.

libaio is the Linux-native asynchronous IO interface. In *libaio*, IO requests are stored in IO command blocks (*iocb*), which are then mapped to IO contexts (*aio_context_t*). These contexts are used to connect submission queues (created using the *io_setup* system call) to their corresponding return values, stored in event structs (*io_event*). Since *libaio* is a true asynchronous IO implementation fully supported by the kernel, all IO requests are handled and scheduled by the kernel. However, *libaio* has the major limitation that it can only operate asynchronously as intended if files are opened with the *O_DIRECT* flag. Since the page cache is bypassed with *O_DIRECT*, this approach may cause slowdowns for workloads that would otherwise benefit from caching, unless caching is implemented at an alternative layer. In addition, *libaio* requires two system calls per IO, one for submission and one to retrieve completion.

io_uring is a relatively new Linux IO interface (added in 2019, kernel 5.1) that provides buffered asynchronous IO using two ring buffers located in memory shared between user and kernel space: one for submission events and one for completion events. One motivation behind using this shared-memory approach is to eliminate memory copying between user and kernel space. After setting up the shared memory region and adding submission requests to the ring buffer, *io_uring* has two methods to submit IOs to the kernel. First, it can perform a regular system call (syscall) with the *io_uring_enter* function, which (unlike *libaio*) can both submit and reap completed IOs in a single system call. Alternatively, *io_uring* can establish a kernel side submission polling (spoll) thread that continuously checks the submission ring for new requests, thus eliminating the system call. Similar to the latency benefit of completion polling (cpoll) [14], submission polling is also expected to help reduce latency, potentially with energy implications. It supports both polling (cpoll) and interrupt (int) based completion.

spdk was developed by Intel as an alternative to kernel-based IO mechanisms, where the kernel is bypassed entirely using a user-space storage stack alternative. This approach has obvious advantages like eliminating system calls, switching between user/kernel space, and user/kernel space memory copying. However, it requires kernel functionalities such as the NVMe driver to be re-implemented, and if needed by applications, the protection, caching, and scheduling features of the kernel as well. The Linux kernel uses dedicated submission queues for each core to reduce the locking overhead (*blk-mq* [6]), and similarly, *spdk* creates a static number of threads and assigns each individual thread a submission-completion queue pair from the device

controller. This way, *spdk* achieves an asynchronous IO interface, where IO requests are submitted and a separate thread polls for completion, upon which a callback function is invoked to return data to the user. Apart from the obvious drawbacks of bypassing the kernel, there are several key limitations of *spdk*. First, it implements a lockless architecture to minimize overhead; however, as a result it forces applications to control locking if multiple threads access the same submission/completion queue pair. Furthermore, *spdk* cannot allocate more queue pairs than the NVMe device supports (31 in our case), limiting the number of applications that can use *spdk* simultaneously. Finally, *spdk* requires binding the NVMe device to its own drivers, unbinding them from the kernel NVMe driver and prohibiting other IO interfaces from accessing the device.

3 IMPACT OF IO INTERFACE DESIGNS

3.1 Experimental Setup and Methodology

We used an Intel Optane (P4801X) ULL SSD as our storage device and attached it to a Dell PowerEdge R230 with a single-socket Intel Xeon E3-1230 quad-core (8 threads, 3.4 GHz) CPU and 64 GB of RAM as the host device. We used the *xfs* file system and installed Ubuntu 22.04 LTS with kernel version 5.18.19 (the latest stable release compatible with all tested IO interfaces). We measured the energy consumption of the system using an Onset HOBO UX120-018 power meter [13]. The HOBO power meter reads several attributes of the system, including voltage, current, and power levels with a resolution of once per second. We used the Flexible IO tester [5] (*fio*, v3.31) for workload generation, and measured the IO performance per unit energy, in IOs per joule, by synchronizing *fio*'s performance data with the power meter's energy consumption data. We ran each workload for two minutes, with 60 seconds of ramp time to enable the system to reach a steady state before recording data. Furthermore, to keep experimental conditions consistent, we used the *O_DIRECT* flag (no page cache) and used the default IO scheduler, *none*. Finally, for interfaces that support batching, we used the default batch size of one. We repeated each experiment five times and present the mean of replicates.

3.2 Workloads

In order to cover various workload behaviors, we generated microbenchmarks with 4 KB, 16 KB, and 128 KB random read and random write requests using *fio*, and ran experiments for both single and multithreaded scenarios. In the single thread case, we used the IO depth feature of *fio* for asynchronous IO interfaces to submit multiple outstanding requests from a single thread. In the multithreaded case, multiple threads submit IO requests to the storage device simultaneously spanning through multiple CPU cores, using both synchronous and asynchronous IO interfaces. In both scenarios, we used IO depths (total number of outstanding requests) of 1, 2, 4, 8, 16, and 31, where 31 is the maximum number of simultaneous IO paths *spdk* supports for our storage device,* enough to saturate it.

Table 2: Latency (Single Thread, 4 KB, IO Depth = 1)

			read (µs)		write (µs)		avg (%) slower
Interface	**Subm.**	**Comp.**	**50th**	**99th**	**50th**	**99th**	**than *spdk***
spdk	—	cpoll	**6.23**	**7.47**	**8.44**	**10.40**	—
posix-sio	syscall	cpoll	8.69	9.06	**11.11**	**12.24**	27.52%
io_uring	syscall	cpoll	8.65	**9.03**	14.50	15.93	46.18%
io_uring	spoll	cpoll	**7.89**	9.05	12.38	22.33	52.30%
posix-sio	syscall	int	11.68	12.36	14.08	15.64	67.54%
io_uring	spoll	int	8.72	11.15	13.34	25.32	72.69%
libaio	syscall	int	12.35	13.06	16.33	21.87	94.21%
io_uring	syscall	int	12.73	13.45	16.92	23.61	102.97%
posix-aio	syscall	int	19.10	28.89	21.60	34.46	220.15%

3.3 Latency Impact of IO Interface Designs

To analyze the latency impact of IO interface design choices, we generate 4 KB read and write workloads from a single thread with an IO depth of 1 and measure the median (50th percentile) and tail (99th percentile) latencies as shown in Table 2. In the last column, we calculate an average slowdown percentage of each design choice compared to *spdk*, and sort the table by this value.

We can quickly observe from the table that the kernel bypass design of *spdk* yields the lowest IO latency of all cases, as it achieves a median read latency of 6.23 µs. By eliminating the need for a system call to submit an IO request, kernel-side submission polling clearly helps reduce the latency gap between kernel bypass and kernel-based solutions. For median read latency, this latency gap is reduced to 1–2 µs when system calls are eliminated using kernel-side submission polling (*io_uring* with sp & cp). It should be noted that for small requests, this still poses a significant overhead, being 26% slower (7.89 µs) than *spdk* (6.23 µs) for 4 KB reads. It should also be noted that the submission polling mechanism of *io_uring* does not help much in the tail. In addition, *io_uring* has a slower write performance compared to the traditional synchronous *posix-sio*.

Considering overall performance, ***posix-sio* with completion polling yields the closest performance to *spdk***, being 27% slower on average across mean and tail latencies. This is surprising as one may expect *io_uring* to be closest to *spdk* due to its incorporation of both submission and completion polling. The reason *posix-sio* wins over *io_uring* lies in *io_uring*'s slower write performance, which is not stated in any previous work to the best of our knowledge.

Finally, *posix-aio*'s user space asynchronous implementation results in the highest IO latency in all cases as it has both

*Our device has 31 pairs, plus one reserved for administration.

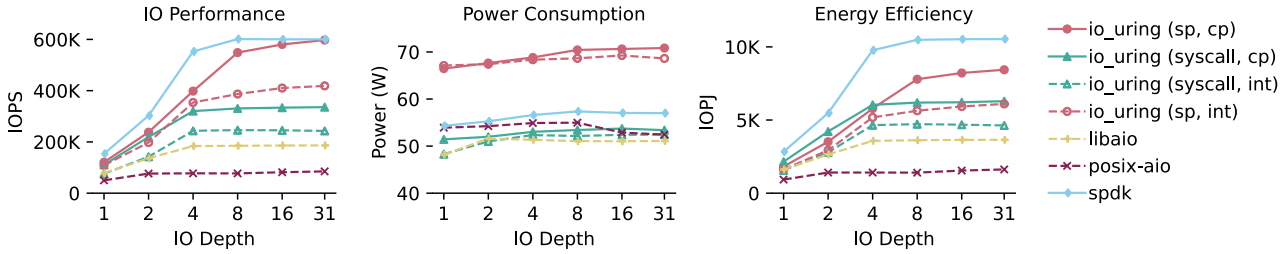

Figure 1: Single Thread, 4 KB, Read

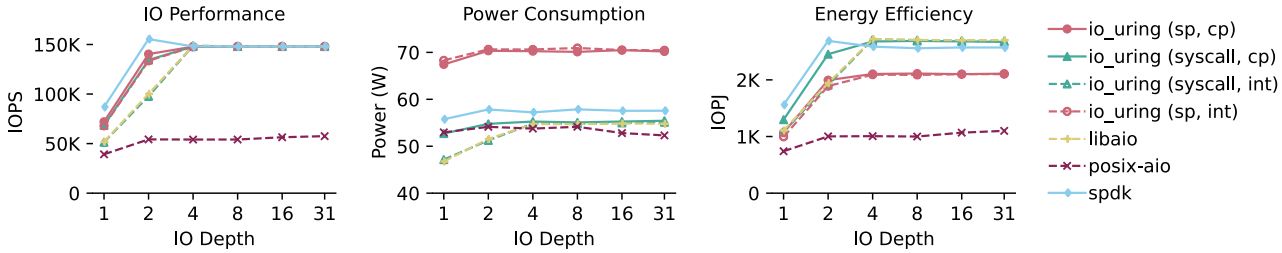

Figure 2: Single Thread, 16 KB, Read

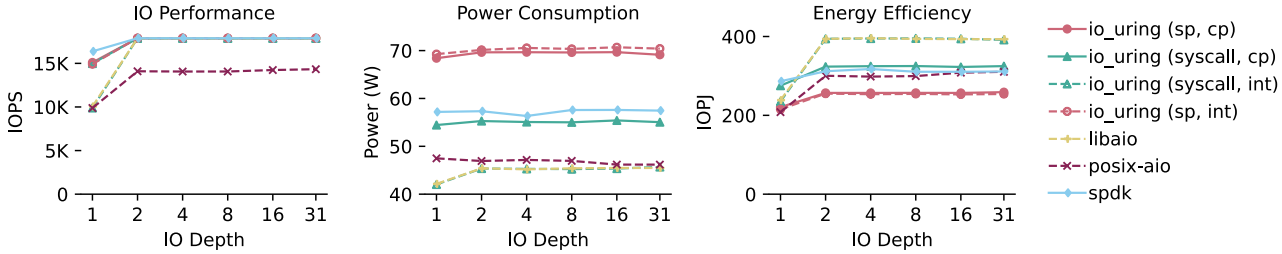

Figure 3: Single Thread, 128 KB, Read

the cost of its thread pool implementation and its reliance on an interrupt based synchronous IO interface underneath. In addition, *fio* checks for completion in user space by continuously polling the *aio_error* return value, reducing *posix-aio*'s performance even further.

3.4 Energy Impact of IO Interface Designs

3.4.1 Single Thread. In single thread experiments, we generate IO requests from a single CPU core using asynchronous IO interfaces. The results are provided in Figures 1, 2, and 3 for 4 KB, 16 KB, and 128 KB read requests, respectively. The *x*-axis indicates the IO depth (number of outstanding requests) and the *y*-axis indicates the IO performance (IOPS), power (W), and energy-efficiency (IOPJ) for the *left*, *middle*, and *right* figures, respectively. We repeated the same experiments for writes as well but did not observe any significant changes in behavior. Write results are provided for the multithreaded case in Section 3.4.2. Also, since *posix-sio* is not capable of issuing more than one outstanding IO request at a time due to its synchronous behavior, it is instead only evaluated as part of the multithreaded experiments in Section 3.4.2.

The kernel bypass design of *spdk* achieves the best energy efficiency in the single thread case for small request sizes. This is possible thanks to *spdk*'s high IO performance with a moderate power usage as shown in Figure 1. However, its energy-efficiency advantage starts disappearing for larger requests sizes, as shown in Figure 2 for 16 KB requests, and completely disappears for even larger requests sizes, as shown in Figure 3 for 128 KB requests. **For larger request sizes, system call and interrupt-based kernel space implementations including *io_uring* and *libaio* achieve the best energy efficiency.** The main culprit for the energy inefficiency of the kernel bypass design of *spdk* for larger request sizes is its polling based completion. As request size gets larger, latency increases. As a result, the cost of polling for completions becomes more significant, rendering kernel bypass designs energy-inefficient as they cannot utilize interrupts without the help of the kernel. This is a significant energy limitation of the kernel bypass techniques for larger request sizes.

As also indicated in Section 3.3 for the IO depth of 1, submission polling helps reduce the performance gap between kernel bypass and kernel-based IO interface deigns, where we see this gap being completely closed at the largest IO depth as shown in Figure 1 (*left*). However, the performance advantage of submission polling disappears as request size

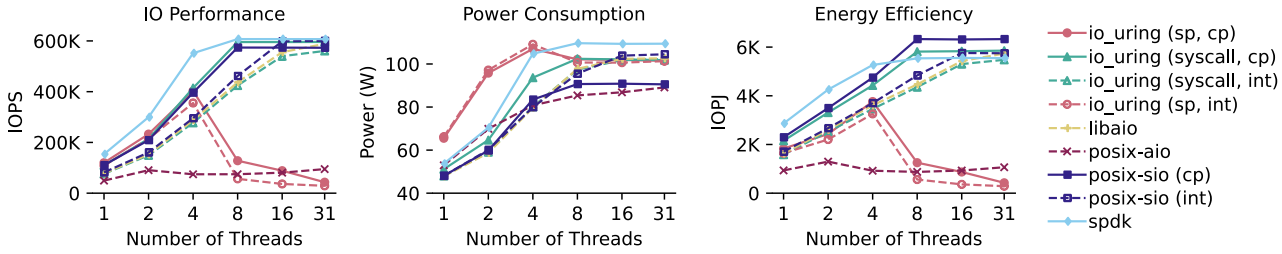

Figure 4: Multithreaded, 4 KB, Read

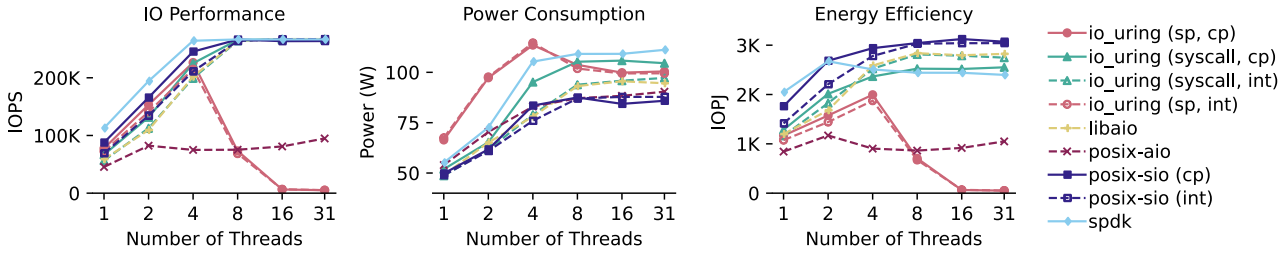

Figure 5: Multithreaded, 4 KB, Write

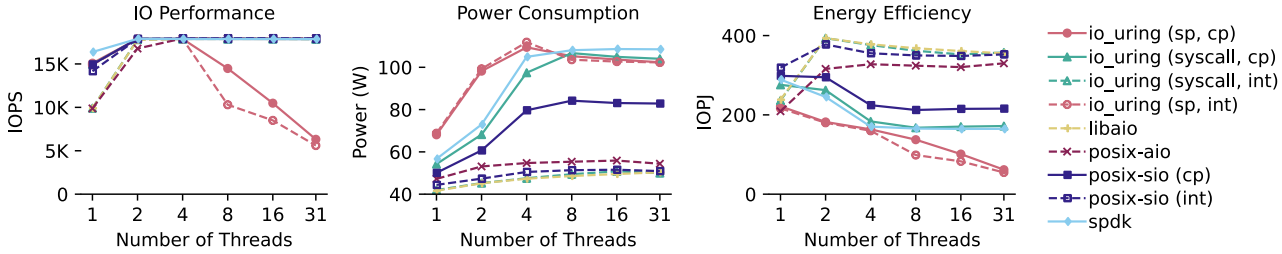

Figure 6: Multithreaded, 128 KB, Read

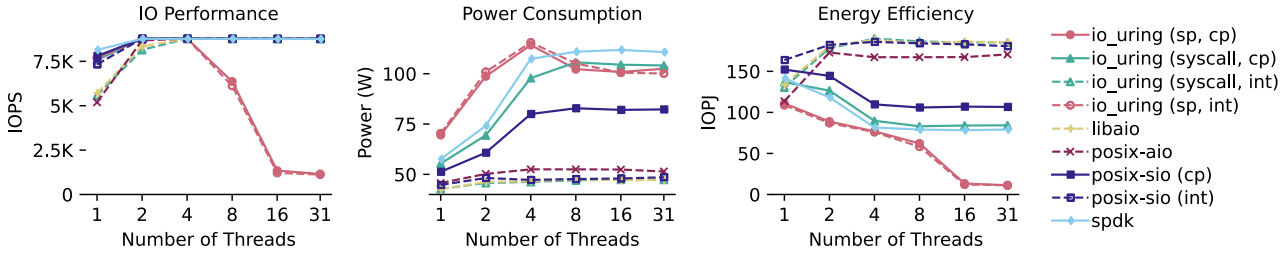

Figure 7: Multithreaded, 128 KB, Write

gets larger (Figures 2 and 3, *left*). On top of that, due to its added power consumption cost (Figures 1, 2, and 3, *middle*), **polling based IO submission is generally less energy efficient than system call based IO submission**. The only exception to this is the larger IO depths of small requests as shown in Figure 1 (*right*), where submission polling mechanisms yield better energy efficiency than system calls in the single thread case as the number of system calls increase. However, this behavior does not repeat in the multithreaded case as we discuss in the next section. The main reason for the energy inefficiency of submission polling lies in its added CPU cost, where submission polling requires an additional core to be dedicated by the kernel for each thread. This can clearly be seen in the CPU utilization graph shown in Figure 8 (*left*) for the single thread case, where submission polling based interfaces use 2 cores, while others use up to one core. The impact of this behavior is even more dramatic in the multithreaded case shown in Figure 8 (*right*).

3.4.2 Multithreaded. Multithreaded results are provided in Figures 4, 5, 6, and 7 for 4 KB read, 4KB write, 128 KB read, and 128 KB write workloads, respectively. 16 KB results are not shared as they did not provide any additional insight. For multithreaded experiments, we generate IO flows from multiple threads, where the x-axis indicates the number of threads (each with IO depth of 1), and the y-axis indicates the IO performance (IOPS), power (W), and energy efficiency (IOPJ) for the *left*, *middle*, and *right* figures, respectively.

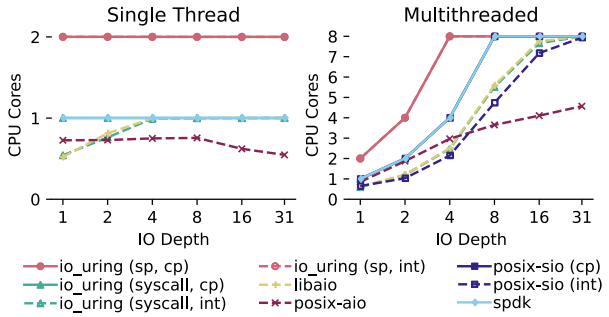

Figure 8: CPU Utilization, 4 KB, Read

One interesting observation that we can make in the multithreaded case is the high energy consumption of the kernel bypass design of *spdk* as well as the interfaces that rely on submission polling mechanism, especially for increasing number of threads (Figures 4, 5, 6, and 7, *middle*). This behavior is also correlated with their CPU core usage shown in Figure 8 (*right*), where both designs quickly reach 8 cores (maximum in our system). Since this added power consumption is not sufficiently compensated for by their IO performance, both kernel bypass and submission polling based designs generally end up being less energy efficient than the traditional *posix-sio* in multithreaded scenarios, especially for higher number of threads.

For larger request sizes (Figures 6 and 7), *spdk* and submission polling based interfaces are among the worst for energy efficiency. Considering all request sizes (Figs. 4–7), ***posix-sio* generally provides the best energy efficiency in the multithreaded case, using its polling version for smaller requests and its interrupt version for larger requests.** It is important to note that *posix-sio* is not capable of issuing more than one request at a time due to its synchronous behavior, whereas the system call based *io_uring* (reads) or *libaio* (writes) are alternative energy-efficient designs in multithreaded scenarios requiring asynchronous capability.

Finally, as also observed in the single thread case, **interrupt based IO completion is crucial to achieving the best energy efficiency for all interfaces when the request size gets larger.** Therefore, it is very important to design polling based IO completion mechanisms carefully. Figures 6 and 7 (*right*) clearly show for both reads and writes, respectively, how the current polling based interfaces yield the worst energy efficiency as they cannot selectively use interrupts, motivating dynamic approaches.

4 RELATED WORK

In addition to the traditional IO interfaces *posix-sio* [3], *posix-aio* [2], and *libaio* [1], two newer interfaces recently became available: *spdk* [15] and *io_uring* [4]. A previous work related to these interfaces is xNVMe [12], which provides a user space abstraction layer for submitting IO requests in an interface independent way so that changing IO interface in the application does not require refactoring the code. There is also an extensive body of work on improving the performance of the storage stack [9, 10, 16, 17]. However, none of these works focus on IO interface design specifically, and their analyses focus on the performance improvement or penalty of the introduced method, as in xNVMe.

There is only one previous work so far that systematically analyzes IO interface characteristics [7], in which the authors provide performance analysis for *libaio*, *spdk*, and *io_uring* using 4 KB read-only workloads issued on a flash-based SSD. We observed similar performance trends; however, our main focus in this paper is the energy implications of IO interfaces by investigating not only *libaio*, *spdk*, *io_uring*, but also *posix-sio* and *posix-aio* design choices. In addition to 4 KB reads, we also analyzed write and multithreaded workloads in various request sizes using an ULL SSD allowing us to make cost analysis for single-digit microsecond IO latencies. To the best of our knowledge, no other work has so far investigated the energy implications of IO interface design choices.

5 DISCUSSION AND CONCLUSION

In this paper, we investigate the energy implications of IO interface design choices using a power meter and an ultra-low latency storage device. Our experiments indicate that kernel bypass designs provide the lowest latency; however, they can also have serious energy implications for larger request sizes as they cannot utilize interrupt based completion. Kernel-side submission polling is a promising method to close the performance gap between kernel-based and kernel bypass designs, as it can reduce the latency difference to 1–2 µs. However, as in any polling mechanism, if not carefully designed, it can also incur a serious energy penalty. For better energy efficiency, it is crucial to design polling based IO submission and completion with careful consideration of poll duration. Shorter poll durations can yield more energy efficient designs than system call based submission and interrupt based completion mechanisms for smaller request sizes, while longer poll durations can negatively impact energy efficiency for larger request sizes or multithreaded scenarios. Hybrid IO submission and completion mechanisms capable of selectively using the most appropriate method have the potential to provide energy-aware high performance storage stack designs.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2012. Linux Asynchronous I/O Explained. https://www.fsl.cs.sunysb.edu/~vass/linux-aio.txt.

[2] 2021. aio - POSIX asynchronous I/O overview. https://man7.org/linux/man-pages/man7/aio.7.html.

[3] 2021. read(2) - Linux manual page. https://man7.org/linux/man-pages/man2/read.2.html.

[4] Jens Axboe. 2019. Efficient IO through io_uring. https://kernel.dk/io_uring.pdf.

[5] Jens Axboe. 2022. *Flexible I/O Tester, Version 3.31.* https://github.com/axboe/fio.

[6] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '13)*. ACM, New York, NY, USA, Article 22, 10 pages. https://doi.org/10.1145/2485732.2485740

[7] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and Io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 120–127. https://doi.org/10.1145/3534056.3534945

[8] Bryan Harris and Nihat Altiparmak. 2019. Monte Carlo Based Server Consolidation for Energy Efficient Cloud Data Centers. In *11th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2019)*. Sydney, Australia, 263–270. https://doi.org/10.1109/CloudCom.2019.00046

[9] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. 2017. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 345–358. https://www.usenix.org/conference/fast17/technical-sessions/presentation/kim-sangwook

[10] Gyusun 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-gyusun

[11] Alberto Lerner and Philippe Bonnet. 2021. *Not Your Grandpa's SSD: The Era of Co-Designed Storage Devices.* Association for Computing Machinery, New York, NY, USA, 2852–2858. https://doi.org/10.1145/3448016.3457540

[12] Simon A. F. Lund, Philippe Bonnet, Klaus B. A. Jensen, and Javier Gonzalez. 2022. I/O Interface Independence with XNVMe. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 108–119. https://doi.org/10.1145/3534056.3534936

[13] Onset Computer Corporation 2017. *HOBO Plug Load Logger (UX120-018) Manual.* Onset Computer Corporation. https://www.onsetcomp.com/sites/default/files/resources-documents/17838-E%20MAN-UX120-018.pdf

[14] Jisoo Yang, Dave B. Minturn, 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.

[15] 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.

[16] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. 2014. Optimizing the Block I/O Subsystem for Fast Storage Devices. *ACM Trans. Comput. Syst.* 32, 2, Article 6 (Jun 2014), 48 pages. https://doi.org/10.1145/2619092

[17] 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