Understanding Modern Storage APIs: A systematic study of libaio, SPDK, and io_uring

Diego Didona, Jonas Pfefferle
Nikolas Ioannou, Bernard Metzler
IBM Research Europe
Zurich, Switzerland
{ddi,jpf,nio,bmt}@ibm.zurich.com

Animesh Trivedi
VU Amsterdam
Amsterdam, Netherlands
a.trivedi@vu.nl

ABSTRACT

Recent high-performance storage devices have exposed software inefficiencies in existing storage stacks, leading to a new breed of I/O stacks. The newest storage API of the Linux kernel is io_uring. We perform one of the first in-depth studies of io_uring, and compare its performance and dis-/advantages with the established libaio and SPDK APIs. Our key findings reveal that (i) polling design significantly impacts performance; (ii) with enough CPU cores io_uring can deliver performance close to that of SPDK; and (iii) performance scalability over multiple CPU cores and devices requires careful consideration and necessitates a hybrid approach. Last, we provide design guidelines for developers of storage intensive applications.

ACM Reference Format:
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 *The 15th ACM International Systems and Storage Conference (SYSTOR '22), June 13–15, 2022, Haifa, Israel.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3534056.3534945

1 INTRODUCTION

Modern non-volatile memory (NVM) storage technologies, like Flash and Optane SSDs, can support down to single digit μsecond latencies, and up to multi GB/s bandwidth with millions of I/O operations per second (IOPS). CPU performance improvements have stalled over the past years due to various manufacturing and technical limitations [8]. As a result, researchers have put considerable effort into identifying new CPU-efficient storage APIs, abstractions, designs, and optimizations [2, 3, 11, 13, 15, 19, 22, 25, 26, 30, 31]. One specific API, io_uring, has drawn much attention from the community due to its versatile and high performance interface [5, 15, 16, 18, 27, 34]. io_uring was introduced in 2019 and has been merged in Linux v5.1. It brings together many well established ideas from the high performance storage and networking communities, such as asynchronous I/O, shared memory-mapped queues, and polling (Section 2) [9, 10, 31, 32].

With the addition of io_uring, Linux now has multiple ways of accessing a storage device. In this paper, we look at Linux Asynchronous I/O (libaio) [6, 24], the Storage Performance Development Kit (SPDK) from Intel® [13], and io_uring [15, 17, 18]. These APIs have different parameters, deployment models, and characteristics, which make understanding their performance and limitations a challenging task. The use of the io_uring API and its performance has been the focus of recent studies [7, 28, 33, 36]. However, to the best of our knowledge, there is no systematic study of these APIs that provides design guidelines for the developers of I/O intensive applications. There has also been an extensive body of work in studying system call overhead [29], implementing better interrupt management for I/O devices [30], leveraging polling for fast storage devices [38], using I/O speculation for μsecond-scale devices such as NVMe drives [35], and improving the performance of the Linux block layer in general [3, 39, 40]. These works are orthogonal to ours, since they explore designing new storage stacks, while we focus on the performance characteristics of state-of-the-art APIs that are readily available in Linux.

Our main contributions include (i) a systematic comparison of libaio, io_uring, and SPDK, that evaluates their latency, IOPS, and scalability behaviors; (ii) a first-of-its-kind detailed evaluation of the different io_uring configurations; and (iii) design guidelines for high-performance applications using modern storage APIs. Our key findings reveal that:

SYSTOR '22, June 13–15, 2022, Haifa, Israel

ACM ISBN 978-1-4503-9380-5/22/06…$15.00
https://doi.org/10.1145/3534056.3534945

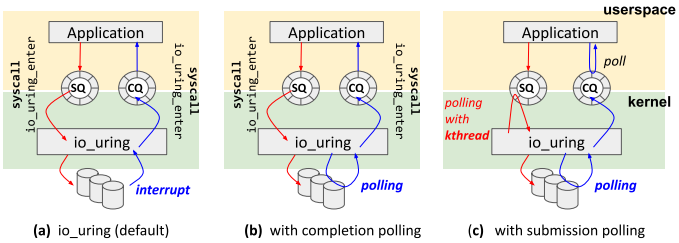


Figure 1: `io_uring` I/O modes considered in this paper.

CPU	2x Intel® Xeon® E5-2630, 2.2GHz, 10 cores/socket, hyper-threading disabled, with Spectre and Meltdown patches, intel_pstate=disable, intel_idle.max_cstate=1
DRAM	128 GiB, DDR4
Storage	20 Intel® DC P3600 400GB NVMe SSDs (10x2 on 2 NUMA nodes), spec: 320KIOPS (randread), 30KIOPS (randwrite), preconditioned as in [4]
Software	Ubuntu20, kernel 5.13.0-051300-generic, fio 3.28, SPDK commit af31c4c

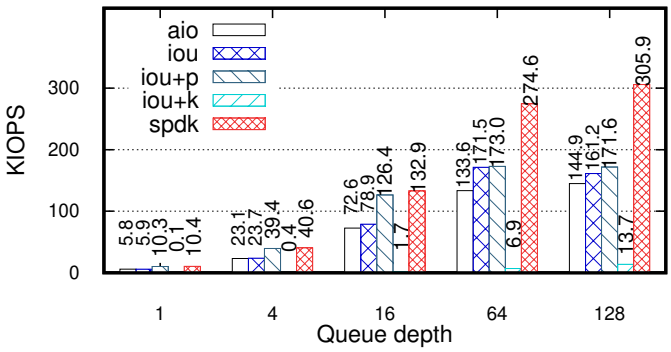
Table 1: Benchmarking environment.

• **Not all polling methods are created equal.** We evaluate different polling mechanisms. SPDK offers a single user-space polling mechanism for both submission and completion, while `io_uring` offers two options that can be enabled independently: polling for completion, and kernel-thread polling for submission. We observe that polling can be both the key to achieving high performance and the cause of order-of-magnitude performance losses (Section 3.1).
• **`io_uring` is close to SPDK.** `io_uring` with kernel polling can deliver performance close to SPDK (within 10%), thanks to the elimination of system calls from the I/O path. However, this performance needs twice as many CPU cores as SPDK (Section 3.2).
• **Performance scalability warrants careful considerations.** When not enough CPU cores are available, `io_uring` with kernel polling can lead to a collapse of performance. Hence, a hybrid and scale-aware approach needs to be taken for the selection of the API and its parameters (Section 3.3).

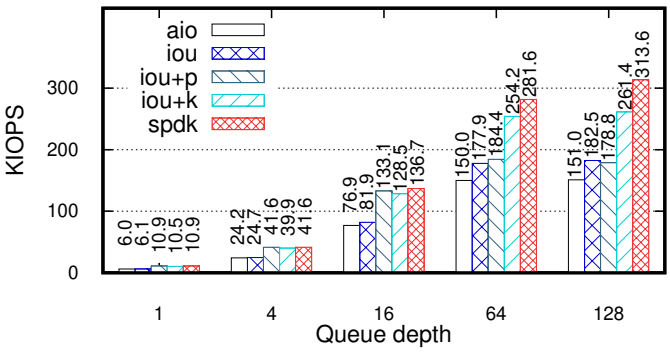
2 LIBAIO, SPDK, IO_URING: A PRIMER

`libaio`. The Linux asynchronous API allows applications to interact with any block device (HDD, SATA SSDs, and NVMe SSDs) in an asynchronous fashion [6, 24]. The main benefits of `libaio` are its ease of use, flexibility, and high performance compared to the traditional blocking I/O APIs. The design of `libaio` revolves around two main system calls: `io_submit` to submit I/O requests to the kernel, and `io_getevents` to retrieve the completed I/O requests. The main limitation of `libaio` is its per I/O performance overhead [20, 33], which stems from relying on two system calls per I/O operation, using interrupt-based completion notifications, and copying meta-data [15, 33]. Moreover, `libaio` only supports unbuffered accesses (i.e., with O_DIRECT) [15, 18].
SPDK. Introduced by Intel in 2010, SPDK [13] is the *de facto* high-performance API in Linux, used by many projects [14, 20, 23, 37, 41]. SPDK implements a zero-interrupt, zero-copy, poll-driven NVMe driver in user space. PCIe registers are mapped to user space to configure submission (SQs) and completion (CQs) queues shared between a device and an application. I/O requests are submitted to the SQs and completions are polled from the CQs without the need for interrupts or system calls. The downsides of SPDK are its increased complexity and reduced scope of usability with respect to `libaio`: SPDK does not support Linux file system integration, and cannot benefit from many kernel storage services such as access control, QoS, scheduling, and quota management.
`io_uring`. `io_uring` aims to bridge the gap between the ease of use and flexibility of `libaio` and the high performance of SPDK. `io_uring` (i) implements a shared memory-mapped, queue-driven request/response processing framework; (ii) supports POSIX asynchronous data accesses both on direct and buffered I/O; (iii) works with different block devices (e.g., HDDs, SATA SSDs and NVME SSDs) and with any file system (and files). `io_uring` achieves low meta-data copy and system call overhead by implementing two ring data structures that are mapped into user space and shared with the kernel. The *submission* ring contains the I/O request posted by the application. The *completion* ring contains the results of completed I/O requests. The application can insert and retrieve I/O entries by updating the head/tail pointers of the rings atomically, without using system calls.

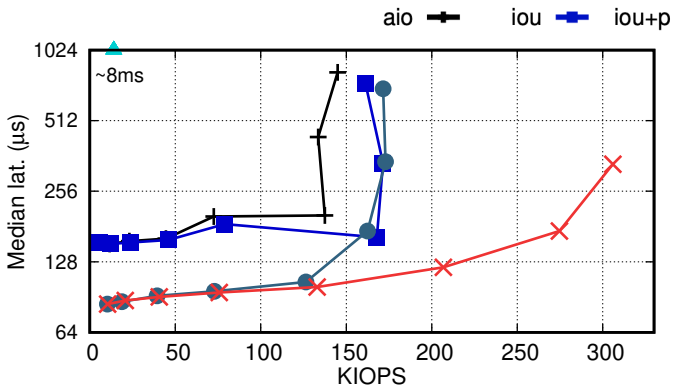
`io_uring` can perform I/O in different ways. Figure 1 provides a visual representation of the different `io_uring` I/O modes, which we describe below and evaluate in Section 3. By default, the application *notifies* the kernel about new requests in the submission ring using the `io_uring_enter` system call. As the completion ring is mapped in user space, the application can check for completed I/O by polling, without issuing any system calls. Alternatively, the same `io_uring_enter` system call can be used to *wait* for completed I/O requests. `io_uring_enter` supports both an interrupt-driven (Figure 1a) and a polling-based (Figure 1b) I/O completion. The `io_uring_enter` system call can be used to submit new I/O requests and at the same time wait for completed I/O requests. This allows reducing the number of system calls per I/O. `io_uring` further supports an operational mode that requires no system calls. In this mode, `io_uring` spawns a kernel thread (one per `io_uring` context) that continuously polls the submission ring for new I/O requests (Figure 1c).



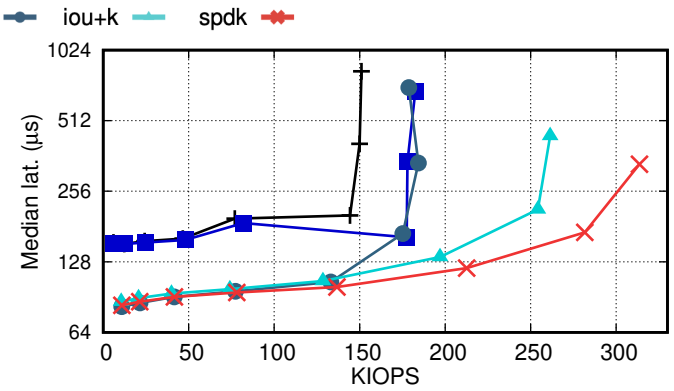
(a) Queue depth vs. IOPS, one core *(x-axis in log scale).*



(b) Queue depth vs. IOPS, two cores *(x-axis in log scale).*



(c) IOPS vs. median latency, one core *(y-axis in log scale).*



(d) IOPS vs. median latency, two cores *(y-axis in log scale).*

Figure 2: IOPS (top) and median latency (bottom) achieved by a single `fio` job accessing one drive with one core (left) or two cores (right). In the latency plots, the different IOPS values are obtained using the queue depth as independent variable, which can lead to curves with the shape of a 'hook' (more details are provided in Section 3.1).

3 PERFORMANCE EVALUATION

In this section, we compare the performance of `libaio`, `io_uring`, and SPDK, using `fio` [1] as the workload generator. We used `fio` because of (i) its flexibility in generating I/O workloads; (ii) its low-overhead I/O path; and (iii) its full support for SPDK and different `io_uring` configurations. We configured `fio` to perform random data reads at the granularity of 4KiB using unbuffered I/O. We chose a read-only workload because it allows higher IOPS on our drives with respect to a mixed workload or a write-only one [12]. The higher IOPS allowed a better evaluation of the scalability trends of the different APIs and of the effects of the overhead per I/O operation (e.g., system calls). We used the default values for the workload generation parameters. We also used the default configuration parameters for each API, except for `io_uring`, which we evaluated under three different configurations: (i) `iou`: uses `io_uring_enter` to submit new I/O, and uses `io_uring_enter` with interrupts to wait for completed requests if none are available upon submission (default in `fio`, Figure 1a); (ii) `iou+p`: same as `iou` except that it uses polling instead of interrupts in `io_uring_enter` (`hipri` parameter in `fio`, Figure 1b); (iii) `iou+k`: uses the kernel poller thread for I/O submission, and uses polling to become aware of completed I/O (`sqthread_poll` parameter in `fio`, Figure 1c) – `iou+k` has zero system call overhead per I/O. Table 1 describes our benchmarking setup.

3.1 Understanding Polling

We first measured the performance of the three libraries when running a single `fio` job that targets a single NVMe drive, using multiple queue depths (from 1 to 128). We ran this experiment in two variants, with one or two CPU cores,[1] placed on the same NUMA node as the drive. Figure 2a and Figure 2b report the IOPS obtained for different queue depth values with one core and two cores, respectively. Figure 2c and Figure 2d report the median latencies corresponding to

[1]Other cores disabled using `/sys/devices/system/cpu/cpuN/online`

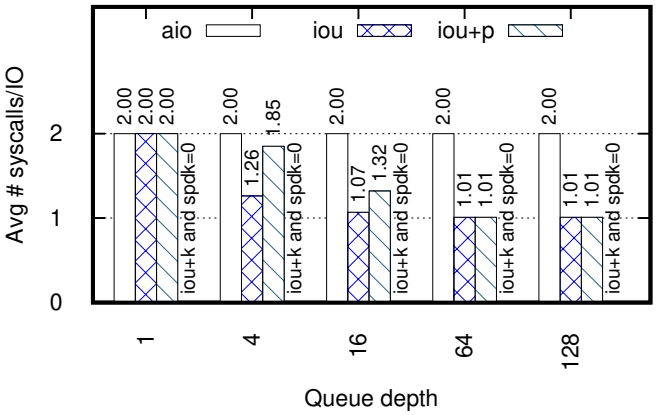


Figure 3: Average #syscalls per I/O *(x-axis in log scale).*

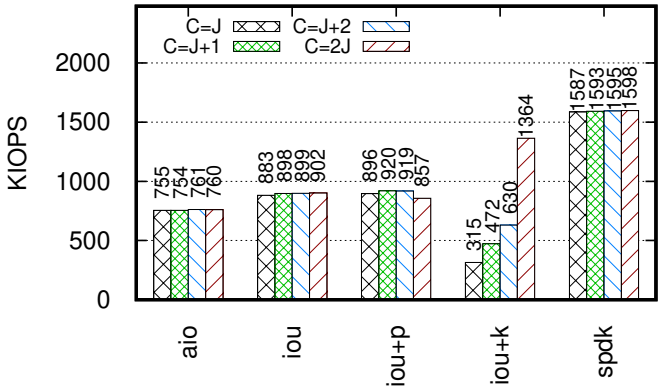


Figure 4: Throughput varying #cores (C) with 5 jobs (J).

the IOPS values obtained on one core and two cores, respectively. We also ran the tests on three cores without observing any significant difference. We note that some libraries exhibit latency functions with the shape of a 'hook'. This is due to the fact that, for these libraries, increasing the queue depth past the saturation point leads to a latency increase and a slight decrease in IOPS, due to increased overhead. Figure 3 reports the average number of system calls per I/O operation (which is the same on one and two cores). The results lead to three primary observations.

First, with a single core, `iou+k` suffers a catastrophic performance loss delivering only 13 KIOPS, i.e., one order of magnitude less than the other APIs. In this configuration, the `fio` thread and the kernel poller thread share the single CPU in mutual exclusion. The kernel thread takes up a significant share of the CPU cycles (50% according to our `perf` tracing), which leads to delays in processing the I/O requests in `fio`. The median latency of `iou+k` is 8 msec, i.e., one or two orders of magnitude worse than that of the other two APIs. The median latency of `iou+k` does not vary with throughput, because it is determined by the interleaving dynamics described earlier, rather than by queueing effects as it is the case for the other libraries. With two cores (one for `fio` and one for kernel thread), the performance of `iou+k` recovers completely, being second only to that of SPDK: the maximum throughput of `iou+k` is 18% lower than SPDK's, and the median latency of `iou+k` is, up to 200 KIOPS, equal or within 10% of SPDK.

Second, SPDK delivers the best performance in every configuration. With just one core, SPDK achieves 305 KIOPS versus the 171 KIOPS and 145 KIOPS of the best `io_uring` alternative and `libaio`, respectively. With two cores, SPDK achieves 313 KIOPS, vs the 260 KIOPS and 150 KIOPS of `iou+k` and `libaio`, respectively. Moreover, SPDK is the only library capable of saturating the bandwidth of the drive, while all other approaches are CPU-bound. Part of this efficiency can be traced down to SPDK's optimized software stack with zero system call overhead, zero-copy and polling-based I/O (see Figure 3). Despite embracing the same polling-based approSPDK'sach, `iou+k` cannot achieve the same performance as SPDK. `iou+k`, in fact, runs polling on two threads, the application one and the kernel one, both accessing the same shared variables and data structures, which incur overhead from atomic accesses, memory fences and cache invalidations. SPDK, instead, implements polling with the application in the same thread, allowing higher resource efficiency. As an example, on two cores and with a queue depth of 16 (where `iou+p` and SPDK have similar throughput), `iou+p` experiences a cache miss rate of 5% versus 0.6% for SPDK (`cache-misses` counter in `perf`).

Third, regardless of the number of cores, `iou+p` achieves performance that is comparable with SPDK for low to medium throughput values (up to $\approx$ 150 KIOPS). This result is explained by the fact that, at low queue depths, the system call overhead in `iou+p` is not yet so high as to be a bottleneck, and hence the polling implemented by `iou+p` is as effective as the polling implemented by SPDK. At higher queue depths, however, the system call overhead becomes the bottleneck for `iou+p`, leading to performance that is worse than SPDK's. A similar dynamic can also be observed with `iou` and `libaio`. Up to a queue depth of 16, they achieve very similar throughput (79 KIOPS and 72 KIOPS, respectively) and median latency (185 μsec and 190 μsec, respectively). However, as the depth increases (> 16 in Figure 2a and Figure 2b), the higher CPU efficiency of `iou`, which incurs fewer system calls per I/O operation than `libaio`, helps to deliver better performance (182 KIOPS of peak throughput versus 151 KIOPS on two cores).

Interestingly, up to queue depth=16 `iou` incurs fewer system calls per I/O on average than `iou+p`, despite achieving worse latency than `iou+p`. This happens because, at low queue depths, there is a higher probability that after submitting all the I/O requests, `fio` has to wait for at least one I/O

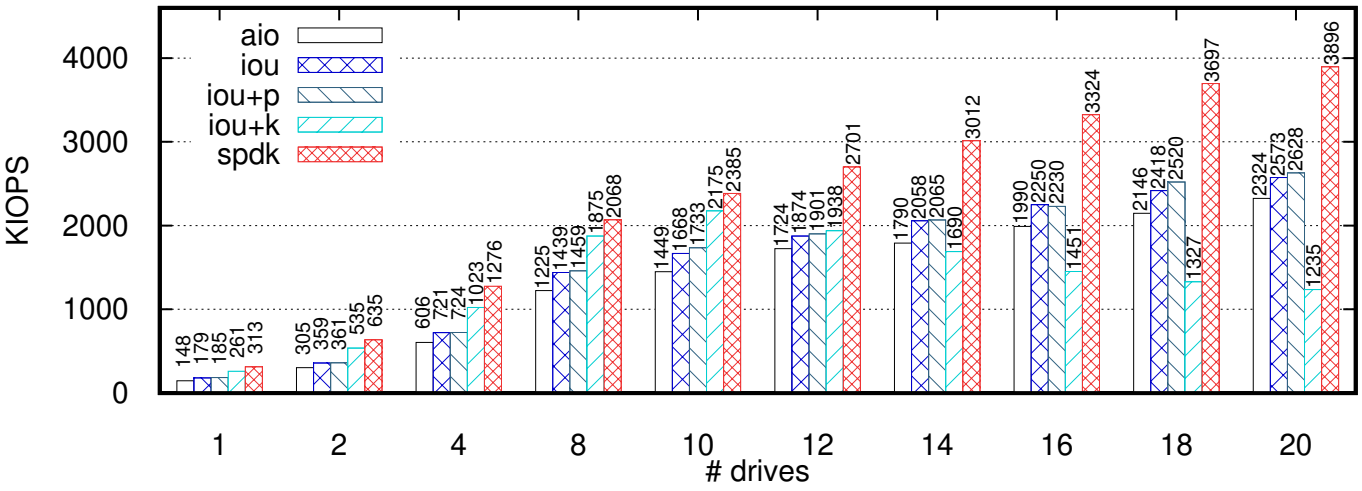


Figure 5: Performance scalability over multiple NVMe devices and cores (1 fio job per device, 20 core system).

request to be completed. Then, the delay caused by the interrupt handler in iou allows for processing more completions at once, at the expense of latency. iou+p, instead, reaps a completed request as soon as it is available, thus potentially missing out on opportunities to batch. As the queue depth increases, both approaches converge to an average of one system call per I/O operation, and iou+p achieves an higher throughput by eschewing the interrupt handling overhead.

We note that the results that we have reported differ from other experimental results reported online with a single physical core [16]. This discrepancy is due to the fact that such previous results have been obtained with more powerful SSDs and CPUs, an optimized benchmarking tool, and an experimental Linux kernel version [21].

3.2 Different CPU-to-drive ratios

In light of the results discussed so far, we studied the performance of the APIs using more than one drive. In particular, we aimed to observe how many CPUs per drive iou+k needs, in the general case, to obtain the best performance and avoid the performance degradation of the one core scenario described earlier. We ran a test in which fio runs $J = 5$ jobs, each accessing a distinct drive, with a queue depth of 128, and we enabled a different number of cores C on the machine. We set $C = J, J + 1, J + 2, J * 2$. We used $J = 5$ because it corresponds to the largest setting such that all the experiments could run in a single NUMA domain (10 cores per domain). We remark that iou+k spawns one kernel poller thread per fio job. Figure 4 reports the results of our experiments.

The results indicate that iou+k is the only library that benefits significantly from higher CPU-to-drive ratios. The other libraries only marginally benefit from additional available CPUs. iou+k, in particular, needs twice as many CPUs as drives to achieve the highest throughput, indicating that each polling kernel thread needs a dedicated CPU to achieve the best performance. When iou+k can run with a dedicated core per kernel polling thread, it achieves throughput that is only $\approx 15\%$ lower than SPDK, $\approx 45\%$ higher than iou+p and iou, and $\approx 80\%$ higher than libaio. These values suggest that iou+k can achieve remarkable performance without needing a complete rewrite of an application, as is the case with SPDK. iou+k, however, can be the worst-performing solution if the number of extra cores is not optimal. In our case with 5 jobs, allocating just two extra cores to iou+k leads to throughput that is $\approx 20\%$ lower than libaio, and allocating no extra cores leads iou+k's throughput to plummet to less than half the throughput of libaio.

These results shed light on the inherent provisioning costs incurred by iou+k to achieve high performance, which were overlooked by previous experimental results that did not take into account the CPU-to-drive ratio when analyzing the performance achievable by iou+k [16].

3.3 Scalability

We now present the results obtained when running the different libraries on a number of drives varying from 1 to 20 to study their scalability. We configured fio to run J jobs, with J ranging from 1 to 20, each accessing a different drive. In light of the results presented so far, we ran the experiments with $C = 2J$ cores (up to a maximum of $C = 20$, which is the number of physical cores available on our machine). We uniformly spread the drives and cores across the two NUMA domains of the machine when $J > 1$. We ran the tests with a queue depth of 128 to measure close to the peak throughput achievable by the APIs. Figure 5 reports the results of the experiments.

The results showcase the implications for scalability of the dynamics that we have described in the previous sections. SPDK achieves the best performance across the board, and the second best performing library depends on the number of `fio` jobs executing and the number of cores available.

As long as $J \leq 10$, `iou+k` can allocate a separate core to each kernel polling thread, achieving linear scalability and throughput that is between 9% and 16% lower than SPDK's, between 27% and 45% higher than `iou` and `iou+p` (which perform very similarly), and between 50% and 76% higher than `libaio`. As soon as the number of jobs J is such that $2J > 20$, however, the kernel polling threads and the application threads start to interleave their executions on the limited number of cores, leading to a gradual performance degradation of `iou+k`. In our setting, $J = 12$ is the point where the performance of `iou+k` crosses those of `iou+p` and `iou`. With $J = 14$, `iou+k` becomes the worst performing library, with throughput that is 44% lower than SPDK's, 18% lower than `iou`'s and `iou+p`'s, and even 5% lower than `libaio`'s. When $J = 20$, `iou+k`'s throughput is less than one third of that achieved by SPDK, and roughly half of that achieved by `libaio` and the other two `iou` variants.

In contrast, `libaio`, `iou` and `iou+p` maintain a rather steady scalability trend, and the latter two achieve near identical performance, as already discussed in the previous sections. From $J = 14$ to $J = 20$, `iou` and `iou+p` are the second best libraries, with throughput that is 33% lower than SPDK's. For those cases, notably, `libaio` achieves throughput that is only 10% lower than `iou+p` and `iou`.

4 LESSONS AND FUTURE DIRECTIONS

Lesson 1: Not all polling methods are created equal. The unified user space polling of SPDK achieves the highest performance across all APIs, by eliminating data copies and system call overhead, but also by performing all I/O operations through a single thread context. `iou+k` also uses no system calls and minimizes data copies, but can suffer from catastrophic performance loss if not enough extra cores are available for the kernel poller threads (Figure 2). `iou+p` uses a system call-aided polling scheme and eschews the need for such extra cores. `iou+p` can achieve similar latencies as SPDK at low throughput (Figure 2b), but cannot match the SPDK's peak performance due to its higher system call overhead (Figure 3).

Lesson 2: `io_uring` can get close to SPDK. The performance and scalability of `iou+k` can be similar to SPDK's, with the crucial *caveat* that more cores than drives must be available on the machine to efficiently support kernel space polling. Our results recommend using twice as many CPU cores as the number of drives (Figure 4). `iou+p` can achieve latencies similar to SPDK under low to medium load (Figure 2b), but ultimately it cannot match the throughput and scalability of SPDK (Figure 5). Finally, `iou` is consistently the worst-performing configuration of `io_uring`, suggesting that polling is one of the key ingredients to unleashing the full potential of `io_uring`.

Lesson 3: Performance scalability needs careful considerations. In our largest experiment (20 drives), SPDK outperforms the second best approach (`iou+p`) in throughput by as much as 50%. The price to pay for these higher performance is giving up out-of-the-box Linux file support, as well as writing application logic amenable to SPDK's polling API. If support for a file system is necessary, which is the case for most applications, then `iou+k` can deliver performance within 90% of SPDK, but it utilizes twice as many cores (20 vs 10). For better performance scalability when not enough cores are available, developers can use `iou+p`, which can match the SPDK performance at low to medium queue depths (Figure 2b).

Research directions. Our study has focused on the performance of the `fio` microbenchmark on raw block devices. An interesting research direction is assessing the implications of different storage APIs on the end-to-end performance of more realistic I/O-intensive applications, like databases. Such applications are often built on top of file systems, incur extra overhead (e.g., synchronization) that can mask I/O path bottlenecks, and use optimizations such as I/O batching. Another open research avenue is identifying more efficient application designs with `iou+k`, for example, by means of a better interleaving between the application and kernel poller threads, or by sharing kernel poller threads across application threads. Finally, we note that `io_uring` supports I/O over sockets as well, hence its performance should be studied also in the context of networked applications.

5 CONCLUSIONS

We present the first systematic study and comparison between SPDK, `libaio` and the emerging `io_uring` storage APIs on top of raw block devices. Our main findings are that polling and a low system call overhead are crucial to performance, and that `io_uring` can achieve performance that is close to SPDK's, but obtaining `io_uring`'s best performance requires understanding its design and applying careful tuning.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback. Special thanks to our shepherd, Geoff Kuennig, for his careful reading and his many insightful comments and suggestions, which greatly improved the paper. Animesh Trivedi is supported by the NWO grant number OCENW.XS3.030, Project Zero: Imagining a Brave CPU-free World!

REFERENCES

[1] Jens Axboe. Accessed: 2021-12-20. The Flexible I/O tester. https://fio.readthedocs.io/.

[2] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 689–703.

[3] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *6th International Systems and Storage Conference (SYSTOR 13)*. ACM, Article 22, 10 pages.

[4] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proc. VLDB Endow.* 14, 3 (2020), 364–377.

[5] Rust docs. Accessed: 2021-12-20. Crate io_uring. https://docs.rs/io-uring/latest/io_uring/.

[6] Daniel Ehrenberg. Accessed: 2021-12-20. The Asynchronous Input/Output (AIO) interface. https://github.com/littledan/linux-aio.

[7] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *10th Conference on Innovative Data Systems Research (CIDR 20)*. www.cidrdb.org, Online Proceedings. http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf

[8] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan 2019), 48–60.

[9] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. 2018. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 17–33.

[10] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP == RDMA: CPU-efficient Remote Storage Access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 127–140.

[11] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for Microsecond Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 113–128.

[12] Intel(R). Accessed: 2022-05-02. Intel® SSD DC P3600 400GB NVMe SSDs. https://ark.intel.com/content/www/us/en/ark/products/80997/intel-ssd-dc-p3600-series-400gb-2-5in-pcie-3-0-20nm-mlc.html.

[13] Intel®. Accessed: 2021-12-20. The Storage Performance Development Kit (SPDK). https://spdk.io/.

[14] Intel®. Accessed:2022-04-26. SPDK In The News. https://spdk.io/news/.

[15] Jens Axboe. Accessed: 2021-12-20. Efficient IO with io_uring, . https://kernel.dk/io_uring.pdf.

[16] Jens Axboe. Accessed: 2021-12-20. That's it. 10M IOPS, one physical core. https://twitter.com/axboe/status/1452689372395053062.

[17] Jonathan Corbet. Accessed: 2021-12-20. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/.

[18] Jonathan Corbet. Accessed: 2021-12-20. The rapid growth of io_uring. https://lwn.net/Articles/810414/.

[19] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX Association.

[20] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, 1–15.

[21] Michael Larabel. Accessed: 2022-05-02. Axboe Achieves 8M IOPS Per-Core With Newest Linux Optimization Patches. https://www.phoronix.com/scan.php?page=news_item&px=8M-IOPS-Per-Core-Linux.

[22] 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 *USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 603–616.

[23] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Scale and Performance in a Filesystem Semi-Microkernel. In *28th Symposium on Operating Systems Principles (SOSP 21)*. ACM, 819–835.

[24] Linux Programmer's Manual. Accessed: 2021-12-20. io_submit - submit asynchronous I/O blocks for processing. https://man7.org/linux/man-pages/man2/io_submit.2.html.

[25] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. *SIGPLAN Not.* 49, 4 (Feb 2014), 471–484.

[26] Anastasios Papagiannis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2017. Iris: An Optimized I/O Stack for Low-Latency Storage Devices. *SIGOPS Oper. Syst. Rev.* 50, 2 (Jan 2017), 3–11.

[27] PingCAP-Hackthon2019-Team17. Accessed: 2021-12-20. IO-uring speed the RocksDB & TiKV. http://openinx.github.io/ppt/io-uring.pdf.

[28] Ruslan Savchenko. 2021. Reading from External Memory. arXiv:cs.DC/2102.11198

[29] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, 33–46.

[30] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. 2021. Optimizing Storage Performance with Calibrated Interrupts. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 129–145.

[31] Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Kornilios Kourtis, Ioannis Koltsidas, and Thomas R. Gross. 2018. FlashNet: Flash/Network Stack Co-Design. *ACM Trans. Storage* 14, 4, Article 30 (Dec 2018), 29 pages.

[32] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Roman Pletka, Blake G. Fitch, and Thomas R. Gross. 2013. Unified High-Performance I/O: One Stack to Rule Them All. In *14th Workshop on Hot Topics in Operating Systems (HotOS 14)*. USENIX Association.

[33] Vishal Verma, John Kariuki. Accessed: 2021-12-20. Improved Storage Performance Using the New Linux Kernel I/O Interface. https://www.snia.org/educational-library/improved-storage-performance-using-new-linux-kernel-io-interface-2019.

[34] Wander Hillen. Accessed: 2021-12-20. Preliminary benchmarking results for a Haskell I/O manager backend based on io_uring. http://wjwh.eu/posts/2020-07-26-haskell-iouring-manager.html.

[35] Michael Wei, Matias Bjørling, Philippe Bonnet, and Steven Swanson. 2014. I/O Speculation for the Microsecond Era. In *USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 475–481.

[36] WiredTiger. Accessed: 2021-12-20. Implement asynchronous IO using io_uring API. https://jira.mongodb.org/browse/WT-6833.

[37] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, Keqiang Niu, Sijie Sun, and Minyi Guo. 2020. Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure. In *USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 97–110.

[38] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When Poll Is Better than Interrupt. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. USENIX Association.

[39] 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.

[40] 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, 477–492.

[41] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-density Multi-tenant Bare-metal Cloud. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*. ACM, 483–495.