iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 12 **Issue:** II **Month of publication:** February 2024

DOI: https://doi.org/10.22214/ijraset.2024.58460

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 12 Issue II Feb 2024- Available at www.ijraset.com

Evaluating the Performance of SPDK-based io uring and AIO Block Device

Gaurav Chawda[1], Gauri Vijaykar[2], Yash Kalavadiya[3], Siddhesh Kotkar[4], Dr. Girish Potdar[5]
[1, 2, 3, 4]*Student,* [5]*Associate Professor, Dept. of Computer Technology PICT, Pune*

Abstract: ***In the context of contemporary data processing and storage, this article examines the performance of asynchronous I/O interfaces that are provided by the Linux Kernel. Specifically, this study utilizes an SPDK-based block device module of io uring and AIO which offers features like request queueing and lockless queues.***
Our benchmarks comparing io uring and aio performance revealed an interesting dichotomy in their handling of random I/O patterns. Both approaches exhibited similar performance trends for random-read operations, showcasing a clear benefit from increasing the IO queue depth. This suggests both methods efficiently leverage parallel processing to boost read throughput. However, the story flips for random-write operations. Unlike read workloads, increasing queue depth for writes yielded no performance improvement. This increase significantly inflated latency, introducing undesired delays.
Index Terms: IO URING, LIBIO, IO-Engines, Storage Per-formance Development Kit

I. INTRODUCTION

The relentless ascent of Solid State Drive (SSD) technology, particularly Non-Volatile Memory has ushered in an era of unparalleled storage performance. However, unlocking the full potential of this hardware necessitates advancements in the software interfaces responsible for harnessing its power. This is where three key solutions emerge: SPDK, libaio, and io uring [1,2,3].

The Storage Performance Development Kit (SPDK) stands out as a groundbreaking initiative that aims to revolutionize the way we interact with Non-Volatile Memory Express (NVMe) devices. It offers a high-performance, user-space driver opti-mized specifically for NVMe-based solid-state drives (SSDs), unlocking significant performance gains and addressing lim-itations commonly encountered in traditional kernel-based approaches. The user-space driver supports features like zero-copy, asynchronous operations, and lockless NVMe driver [4].

Several studies have demonstrated that user-space drivers built upon the Storage Performance Development Kit (SPDK) achieve significantly higher performance compared to kernel-based alternatives like libaio and io uring, particularly under diverse system workloads [1].

SPDK also offers a versatile block device layer that abstracts underlying storage devices, empowering applications to inter-act with them efficiently and exploit their full performance potential. This layer goes beyond the standard kernel-based block device drivers, providing additional features such as cre-ation of involuted I/O pipelines, request queuing, and multiple lockless queues for maintaining outstanding I/O requests [7].

For this article, we will utilize SPDK's block device layer with libaio and io uring.

Linux kernel and its libraries provide different mechanisms for asynchronous I/O. One of the early and notable libraries is libaio. Introduced in kernel version 2.6, it offered one of the first dedicated asynchronous APIs for storage devices [1].

libaio offers two crucial system calls: io submit and io getevents. These calls enable non-blocking, unbuffered I/O (O DIRECT flag), bypassing the system's I/O cache for potentially significant performance gains [1].

io uring, introduced in Linux kernel version 5.1, is a powerful and versatile asynchronous I/O interface designed to address the limitations of traditional methods like read(), write(), and aio * functions. It consists of two queues namely
'submission queue' and 'completion queue' which is shared by kernel and user space [6].

io uring boasts remarkable flexibility by offering no restric-tions on request types. Whether initiating a file read or write operation, each request is encapsulated as a submission queue entry (SQE) and subsequently appended to the submission queue's tail [2,6].

It relies on two key system calls amoung many: io uring setup andio uring enter. The former establishes the foundation, initializing the submission and completion queues that manage I/O requests. On the other hand, io uring enter

TABLE I
BENCHMARK ENVIRONMENT SETUP

CPU	Intel(R) Core(TM) i7-9750H CPU @2.60GHz, Hyper-threading disabled
Memory	16GB, DDR4
Storage	256GB NVMe SSD, SKHynix HFS256GD9TNG-L3A0B
OS	Arch Linux, x86 64, 6.7.0-arch3-1
SPDK	v24.01

acts as the conductor, orchestrating both the submission of new requests and the retrieval of completed ones, seamlessly handling asynchronous I/O operations.
Due to the efficient polling design, io uring's performance can get close to the SPDK's user-space driver given it uses double the cores as SPDK [1].

II. PERFORMANCE EVALUATION

Instead of the widely used fio tool, we opted for SPDK's bdevperf for benchmarking the performance of SPDK's uring block device against its aio counterpart. The advantage of bdevperf lies in its light footprint, minimizing latency overhead within the I/O path [8].
Our investigation focused on two common I/O patterns: random read (randread) and random write (randwrite). To ensure consistency, we fixed the block size at 4096 bytes during bdev creation and mirrored it in the benchmark's I/O size setting. Additionally, a benchmark time of 180 seconds was chosen for robust results.
We commenced by setting the I/O depth (queue depth) to 1, simulating a single outstanding request queue. Subsequently, we incrementally increased the queue size to emulate scenarios with parallel workloads.

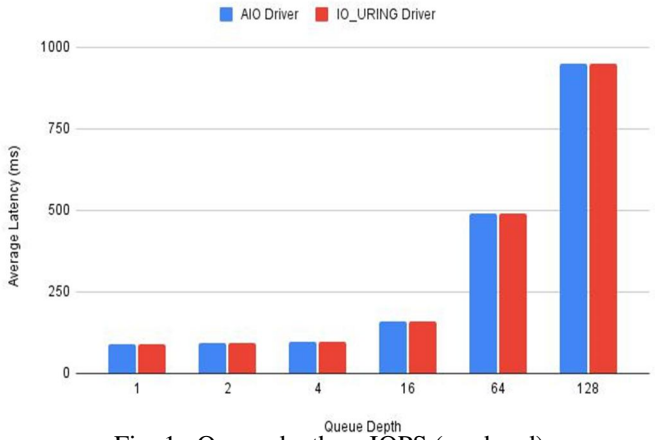


Fig. 1. Queue depth vs IOPS (randread)

Figure 1 depicts the performance of two block device types (io uring and AIO) under a 4KiB random read workload,measured in IOPS (higher is better) across various queue depths.

We observe a clear trend: increasing queue depth leads to significant performance gains, with the most dramatic improvement of 144% occurring at a queue depth of 16. However, further increases in queue depth yield only minor performance benefits.
Interestingly, both block device types exhibit similar per-formance characteristics across the workloads, with io uring bdevs demonstrating a slight edge at low queue depths.

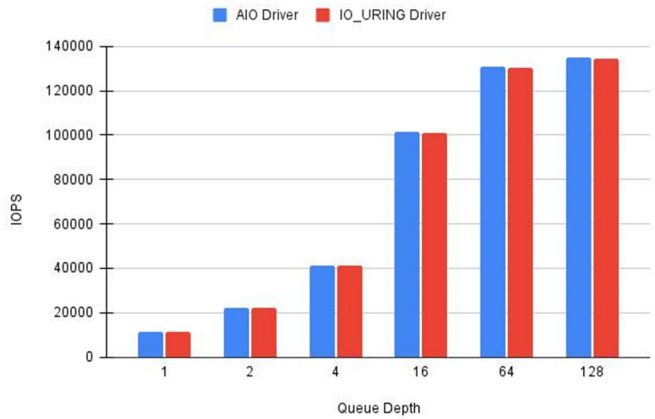


Fig. 2. Queue depth vs average latency (randread)

Figure 2 depicts a bar graph that illustrates the relationship between latency (with lower values signifying better perfor-mance) and queue depths.
Examining the graph reveals that the latency between queue depths 1, 2, and 4 remains consistent, hovering around 90 microseconds. However, for queue depths of 16, 64, and 128, a significant rise in latency is observed.
Particularly, at a queue depth of 64, the latency is increased by approximately 210%. Interestingly, both types of block devices exhibit comparable latency behavior.

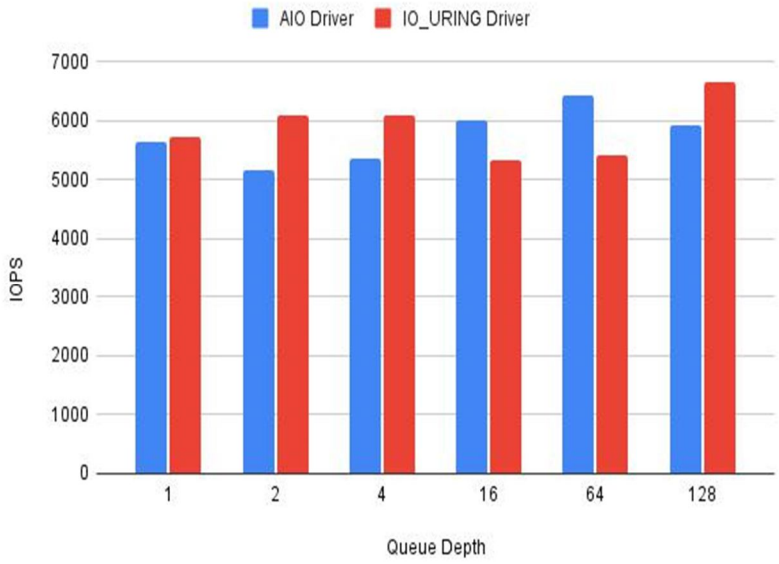


Fig. 3. Queue depth vs IOPS (randwrite)

Figure 3 illustrates the IOPS metric for the randwrite I/O pattern, revealing an interesting contrast to the readrand workload. While increasing queue depth led to a clear rise in IOPS for readrand, the same behavior is not observed for randwrite. Instead, IOPS remain relatively stable at around 5500, regardless of queue depth.
Furthermore, when using fewer queue depths, io uring exhibits marginally better performance than aio. It's also worth noting that the range of IOPS observed in randwrite workloads is considerably smaller compared to readrand workloads.

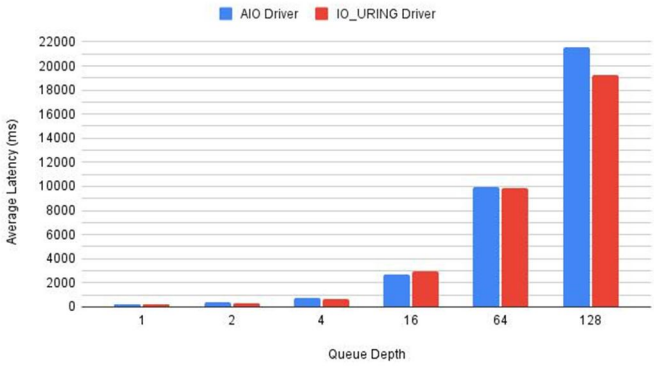


Fig. 4. Queue depth vs average latency (randwrite)

Figure 4 paints a striking picture of how queue depth dramatically impacts latency for randwrite I/O patterns. The data reveals a staggering contrast between a queue depth of 1 and 128, with block device latency soaring from a mere 175 microseconds to a shocking 20,000 microseconds.

Interestingly, io uring consistently outperforms aio in terms of latency across almost all queue depths. At the maximum queue depth of 128, io uring boasts a 10% latency advantage over aio. However, the true story unfolds in the magnitude of latency increases. While both methods experience latency growth with rising queue depth, io uring exhibits a signif-icantly smaller hike. Notably, the biggest performance hit occurs at a queue depth of 64, with latency ballooning by a whopping 272%.

III. CONCLUSION

SPDK's bdevperf benchmark provides valuable insights into the performance characteristics of io uring and AIO SPDK block devices. For random read workloads, both devices offer similar performance, with significant gains achieved by increasing queue depth. However, latency increases for higher queue depths. For random write workloads, IOPS remain stable, but latency increases significantly, especially for AIO. Overall, io uring slighly outperforms AIO in terms of both IOPS and latency.

These findings suggest that both io uring and aio are promising options for applications requiring high performance I/O, particularly for read-heavy workloads. However, careful consideration should be given to queue depth settings to avoid latency spikes, especially for write-intensive workloads.

REFERENCES

[1] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a system-atic study of LIBAIO, SPDK, and IO-Uring. In Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR '22). Association for Computing Machinery, New York, NY, USA, 120–127. doi: 10.1145/3534056.3534945

[2] Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io uring. In Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23). Asso-ciation for Computing Machinery, New York, NY, USA, 35–45. doi: 10.1145/3578353.3589545

[3] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. Proc. VLDB Endow. 16, 9 (May 2023), 2090–2102. doi: 10.14778/3598581.3598584

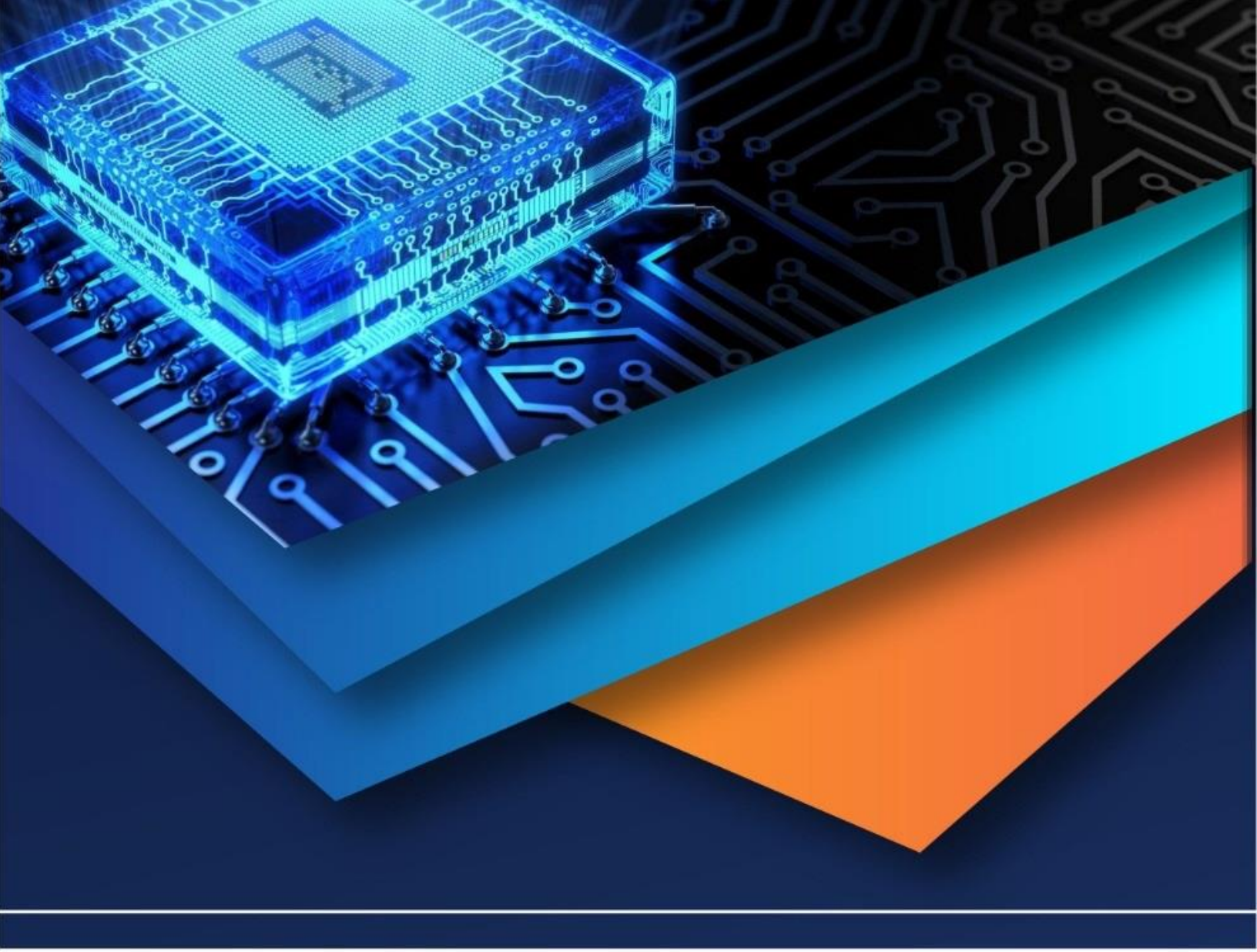
[4] Z. Yang et al., "SPDK: A Development Kit to Build High Performance Storage Applications," 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, China, 2017, pp. 154-161, doi: 10.1109/CloudCom.2017.14.

[5] Youngjin Yu, Dongin 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 (2014), 6:1–6:48. doi: 10. 1145/2619092

[6] Benjamin Block. "An Introduction to the Linux Kernel Block I/O Stack" https://chemnitzer.linux-tage.de/2021/media/programm/folien/165.pdf (accessed Feb 15, 2024).

[7] SPDK. "Block Device User Guide" https://spdk.io/doc/bdev.html (ac-cessed Feb 15, 2024).

[8] Karol Latecki. "SPDK NVMe BDEV Performance Report Release 21.01'' https://ci.spdk.io/download/performancereports/SPD nvme perf report 2101.pdf (accessed Feb 15, 2024)



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY