Studying the Effects of Asynchronous I/O on HPC I/O Patterns

Arnav Gupta†, Druva Dhakshinamoorthy†, Arnab K. Paul
BITS Pilani, K. K. Birla Goa Campus, India
{*f20212092, f20220131, arnabp*}*@goa.bits-pilani.ac.in*

***Abstract*—High-performance computing (HPC) applications have long faced the issue of I/O as a bottleneck. This work analyzes how read/write ratios, I/O block sizes, and sequential versus random access patterns affect a workload's I/O characteristics when using POSIX synchronous I/O (psync), and the relatively new *io_uring* asynchronous interface for single-node performance. Based on our findings, we introduce a preliminary interceptor that intercepts POSIX synchronous I/O calls and replaces them with *io_uring* operations. Initial results demonstrate performance gains with the IOR benchmark, indicating the potential of our approach.**

I. Introduction

High-Performance Computing (HPC) workloads have experienced exponential growth in workload sizes and performance demands in recent years. With the rise of data-intensive applications like machine learning, the I/O performance of HPC environments remains a persistent bottleneck for HPC workloads [1]. Various approaches have been developed to address this, including but not limited to parallel file system optimizations [2], sophisticated caching mechanisms [3], and optimizations like alternating between direct and buffered modes of I/O operations [4]. While these approaches deal with distributed I/O optimization, recent studies indicate that ($\sim$30%) of HPC workloads on supercomputers run on a single node [5]. Moreover, the focus on prefetching data to local memory and disk from the remote parallel file system, warrants the need to optimize the I/O performance at a node-level.

Recent efforts in the Linux community have focused on developing performant asynchronous methods like io_uring [6], to perform I/O at a kernel level. Studies have shown the performance of such asynchronous methods on local file systems with varying I/O interfaces [7]. However, the fundamental I/O interface used by most HPC applications has remained as POSIX, creating a gap in the literature to study the performance of the recent advancements in asynchronous POSIX I/O for HPC I/O patterns.

This paper aims to investigate the performance of HPC I/O patterns using asynchronous I/O, primarily the *io_uring* interface, as compared to synchronous POSIX I/O. We study the impact of read versus write intensity, I/O block sizes, and sequential versus random I/O accesses. Our initial analysis identifies scenarios where *io_uring* outperforms the traditional POSIX I/O calls [8]. Leveraging these insights, we design an interceptor that intercepts I/O calls from applications and dynamically selects between the synchronous POSIX I/O and the asynchronous *io_uring* interfaces.

We show the efficacy of our interceptor using the IOR benchmark on a single node. The findings and the proof-of-concept interceptor hint at the potential for improving I/O performance at the single and multi-node level in HPC environments through targeted optimizations and intelligent adoption of asynchronous I/O interfaces.

† These authors contributed equally to this work.

II. Background and Related Work

The landscape of I/O operations has changed significantly with the release of asynchronous methods, including Linux AIO (libaio) [8], [9] and *io_uring* [6]. Libaio allows applications to submit I/O requests to the kernel and continue execution without waiting for I/O completion. However, to show any significant performance benefit, libaio requires the use of the O_DIRECT flag when opening files. This restriction limits its effectiveness for buffered I/O operations and makes it less flexible for general-purpose use compared to newer alternatives like *io_uring*.

A. How does io_uring work?

As shown in Figure 1, *io_uring* utilises a pair of ring buffers shared between the user space and the kernel space. The aim is to reduce the overhead of context switching required to perform individual I/O operations. The ring buffers; namely the Submission Queue (SQ) and the Completion Queue (CQ) are used to submit I/O operations and retrieve the results of the operation respectively. Memory-mapped buffers allow direct access from user and kernel space, therefore preventing extra copy operations. Once the ring buffers are initialized, I/O requests are filled at the tail of the SQ as Submission Queue Entries (SQE). The application notifies the kernel once these SQEs are filled and continues with other operations asynchronously. The kernel processes the requests, starting with the head of the SQ, and fills the tail of the CQ with the results of the I/O operation as Completion Queue Entries (CQE). The application in the user space then consumes this response, starting from the head of the CQ. The shared memory architecture minimizes context switches and data copying between the kernel and user space. System call overhead is greatly reduced, with notifying the kernel of SQEs being the only system call being used. The ring structure also performs batching, allowing the application to submit multiple I/O requests in a single batch.

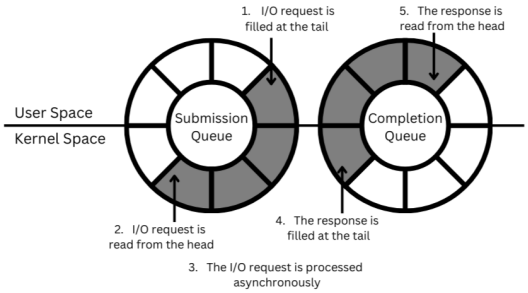
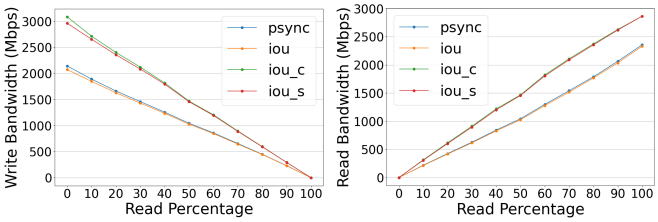


Fig. 1: Design of *io_uring* ring buffers.



(a) Write bandwidth for varying read percentage.

(b) Read bandwidth for varying read percentage.

Fig. 2: FIO: Varying read percentage in mixed read-write workload.

The basic implementation requires notifying the kernel with a system call after filling the SQ with entries. For applications chasing the last bit of performance, *io_uring* offers different modes, namely 'IO_POLLING' and 'SQ_POLLING', the latter being kernel-side polling. In IO_POLLING, the application actively polls for I/O completion status by checking the CQ rather than relying on interrupts, commonly referred to as Completion Queue polling. SQ_POLLING, widely referred to as Submission Queue polling configures the kernel to poll for entries in the SQ. This removes the overhead of the '*io_uring_enter()*' system call that is used to notify the kernel about SQEs. It significantly reduces latency while there is a cost of increased CPU utilisation. This mode can be useful for applications with high request submission rates.

B. Related Work

In the literature, there is only one most relevant work that focuses on systematically characterizing the performance and overheads of various storage I/O stacks and APIs available in the Linux ecosystem. Ren et al. [7] conducts a comprehensive study comparing POSIX I/O, asynchronous I/O (libaio), *io_uring*, and SPDK on high-performance storage hardware, specifically Intel Optane SSDs. Their study underscore the importance of careful consideration when selecting I/O APIs and emphasize the potential benefits of optimizing existing syscalls, which aligns with our hypothesis of intelligently intercepting psync (synchronous POSIX) calls and replacing them with *io_uring* operations. While Ren et al. provide valuable comparative data, our work builds upon these insights to analyze HPC I/O patterns and propose a novel practical interceptor to leverage the potential benefits of *io_uring* while still utilising psync for sections of workloads that perform better with synchronous I/O, thereby offering a performance improvement without necessitating the rewrite of the application with the *io_uring* interface.

III. Analysis

This section aims to assess the I/O performance of the *io_uring* and POSIX synchronous (psync) interfaces by varying the I/O patterns. Specifically, we aim to vary the read-write ratio, I/O block size, and sequential-random I/O access pattern at a single node-level using the FIO benchmark.

A. Experimental Setup

1) Workload: Flexible I/O Tester (FIO) [10] is a widely utilized benchmarking tool for assessing the performance of storage systems. Designed to simulate a variety of I/O workloads, it is also highly configurable, enabling the emulation of a wide range of workload patterns.

2) TestBed: The FIO runs are conducted on a system with a 12th Gen Intel(R) Core(TM) i7-12700, 2 x 32 GB G Skill Intl 5200 MHz memory and Samsung SSD 980 PRO 1TB SSD. We run single-threaded tests on a 5 GB file, comparing psync and *io_uring* calls. 8 polling queues are used on the SSD for benchmarking.

B. Analysis Results

We use POSIX synchronous I/O (psync) and the different modes of asynchronous *io_uring*, namely vanilla *io_uring* (iou), *io_uring* with submission queue polling (iou_s) and *io_uring* with completion queue polling (iou_c).

1) Varying Read/Write ratios: We vary the read percentage from 0% to 100% at 256KB block size. We notice the maximum performance difference between the I/O engines at this block size. As seen in Figures 2a and 2b, *io_uring* polling methods consistently outperform psync and *iou*. The gap grows larger when the write percentage increases in Figure 2a indicating an increase in the number of write operations, and in Figure 2b, where the gap in read bandwidth grows larger when the number of read operations increase. Polling reduces the system call overhead by reducing the number of system calls that are performed in the entire I/O operation, though it comes at a cost of CPU utilisation.

2) Varying I/O Block Sizes: We vary the I/O request block size from 4 KB to 64 MB and measure the write bandwidth. As shown in Figure 3, a consistent performance advantage for *io_uring* polling methods is seen across most block sizes. The bandwidth difference grows wider as the block sizes increase, becoming relatively stable once they cross 4MB. The *iou_c* line doesn't extend past 256KB. This is because the engine crashes for block sizes greater than 256KB. This can be a system configuration error or a lack of support for polling larger block sizes which needs to be further investigated.

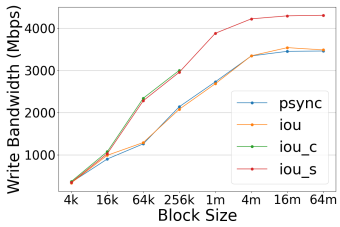
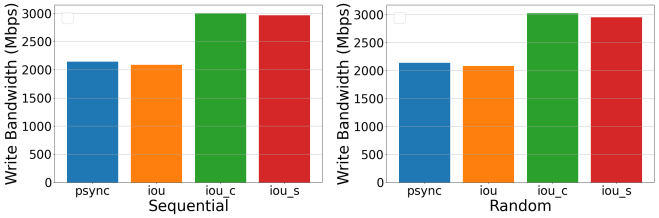


Fig. 3: FIO: Write bandwidth for varying I/O block sizes.



(a) Sequential. (b) Random.

Fig. 4: FIO: Write bandwidth for Sequential vs Random.

3) Random versus Sequential Access: We run FIO in both random and sequential access mixed read-write workload for 256KB and measure the write bandwidth. For the sequential and random access workload, as depicted in Figures 4a and 4b, *iou_s* and *iou_c* demonstrate a clear performance advantage over the other methods. Psync and *iou* perform similarly, with psync having a slight edge. This similarity in performance between the random and sequential workload can be attributed to the high-spec SSD used for the benchmarks.

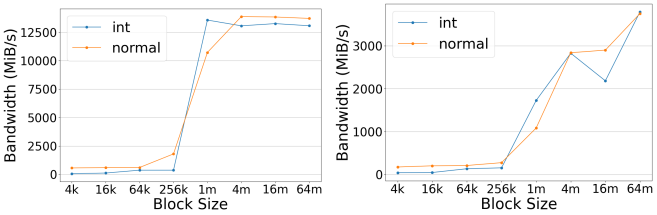
IV. Proof-of-Concept Interceptor

A. Design

The interceptor operates by being compiled as a shared library and loaded using the *LD_PRELOAD* environment variable. This allows interception and modification of function calls made by an application, facilitating custom implementations without modifying the source code and operating at the runtime linker level. When an application issues a read or write call, the interceptor captures this call and creates a Submission Queue Entry (SQE) in the SQ, submits it to the kernel, and continues execution without waiting for the I/O operation to complete. The kernel processes the SQEs asynchronously and places the results in the CQ. The framework then waits for the operation to complete by monitoring the CQ for Completion Queue Entries (CQEs). Once the CQE is available, the result of the I/O operation is retrieved and returned to the application.

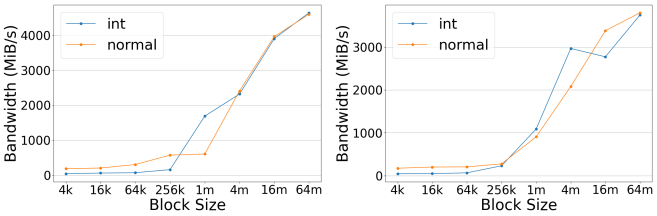
B. Evaluation Setup

Interleaved Or Random (IOR) [11], a widely used synthetic benchmark for measuring the performance of parallel storage systems is used on the same node as III-A2. IOR is configured with 16 segments, varying block sizes, and different I/O characteristics (random write, sequential write, sequential read-write).



(a) Read bandwidth. (b) Write bandwidth.

Fig. 5: IOR: Sequential read-write workloads.



(a) Random write workload. (b) Sequential write workload.

Fig. 6: IOR: Write bandwidth for sequential and random write.

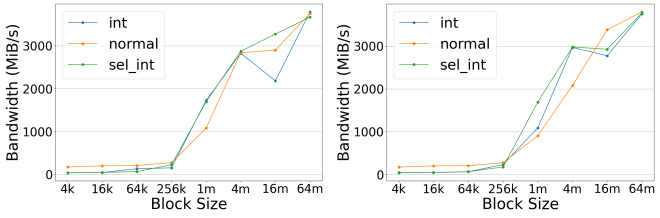
C. Evaluation

We tune the framework to selectively intercept calls and observe when psync performs better than *iou* and vice versa. Currently, the framework does not support the *iou_c* and *iou_s* modes which are expected to deliver significant performance improvements over *iou*.

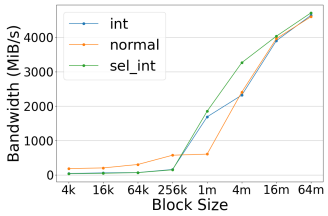
1) Sequential Workloads: Figures 5a and 5b present the results for read and write bandwidth when our interceptor is used for a sequential workload with an equal ratio of reads to writes, over I/O block sizes varying from 4KB to 64MB. For reads (Figure 5a), our interceptor (*int*) performs similarly to psync. Performance for higher block sizes surpasses the bandwidth on a single SSD due to caching and aggregation [12]. The overhead of an increased number of system calls reduces as the I/O block sizes increase. However, the reduction is matched to the increase in the interception pre-processing overhead. In the case of writes (Figure 5b), *int* consistently outperforms psync I/O (*normal*).

2) Sequential and Random Write Workloads: We focus mostly on the write performance. Figures 6a and 6b demonstrate the write bandwidth when the workloads have random and sequential writes respectively over block sizes varying from 4KB to 64MB. The graphs demonstrate the efficacy of our *int*, showing improvements for specific portions of workloads. Once the block sizes exceed 64m, *int* performs equally as *normal*.

Figures 5 and 6 show the performance of the interceptor when every single I/O call is intercepted. We understand how psync compares with asynchronous I/O, albeit with some overhead due to the usage of *LD_PRELOAD*. We observe that the read performance of *iou* is similar to psync, while there are specific portions of write-heavy workloads that would benefit from interception.



(a) Mixed read write workload. (b) Sequential write workload.



(c) Random write workload.

Fig. 7: IOR: Write Bandwidth for selective interception.

The interceptor is accordingly tuned to selectively intercept only write calls. Figures 7a, 7b and 7c compare the performance of *normal*, *int*, and selective interception *(sel_int)* of write calls for a mixed read-write, sequential and random write workload over varying block sizes. We notice that *sel_int* is the best for mixed read-write workload and the random write workload for all block sizes. For sequential write workloads *sel_int* is better till 8m block size. Thereafter, psync is better. These results are also consistent with the FIO runs for sequential and random workloads.

V. Conclusion and Future Work

This work shows the efficiency of using asynchronous I/O (io_uring) over traditional POSIX synchronous calls when dealing with single-node I/O performance in HPC environments. We also develop a proof-of-concept interceptor that intelligently intercepts the required POSIX I/O calls and accordingly translates those to *io_uring* calls. The evaluation of the interceptor on the IOR benchmark on a single-node shows the naive interceptor's ability to outperform traditional I/O methods across diverse HPC scenarios and the selective interceptor's ability to outperform the naive interceptor and traditional psync over a large range of workloads. This hints at its potential to significantly enhance I/O performance in real-world HPC applications with multiple nodes, particularly those involving hybrid read-write patterns, varying block sizes, and sequential access. Currently, the framework does not support *iou_c* and *iou_s* modes which should result in a significant performance improvement as noticed in the FIO runs.

As part of future work, we aim to analyse the effect of asynchronous I/O performance on HPC workloads running on a large number of nodes with more varied benchmarks. We will also improve the design of the interceptor to build an end-to-end framework consisting of an interception layer, an I/O manager, and a communication channel. The interception layer, implemented as a shared library loaded via LD_PRELOAD, will transparently capture I/O system calls from applications and send them via the communication channel to the I/O manager. The I/O manager, running as a standalone daemon, will aggregate I/O operations from multiple applications, process them using *io_uring*, as and when deemed optimal, in the most appropriate mode, and return results to the respective applications. This centralized approach will enable efficient batching of I/O requests and optimal utilization of *io_uring*'s capabilities. The communication manager will facilitate low-latency, high-bandwidth inter-process communication between the interception layer and the I/O manager. We will evaluate various inter-process communication mechanisms such as sockets, pipes, and shared memory to determine the most efficient method for our use case. We also aim to optimise the internal *io_uring* working by finding the optimal block size and queue depth for each workload, further improving the I/O performance. Further analysis of workloads will be done to identify the specific portions which benefit from the interception.

Acknowledgment

This work uses resources in the Data, Systems and HPC (DaSH) Lab at BITS Pilani, KK Birla Goa Campus India. The lab resources are sponsored by SERB, Govt. of India Start-up Research Grant - SRG/2023/002445 and additional grants in BITS Pilani - BBF/BITS(G)/FY2022-23/BCPS-123, GOA/ACG/2022-2023/Oct/11, and BITS CDRF - C1/23/173.

References

[1] M. Isakov, E. d. Rosario, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, and M. A. Kinsy, "Hpc i/o throughput bottleneck analysis with explainable local models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2020.

[2] A. K. Paul, S. Neuwirth, B. Wadhwa, F. Wang, S. Oral, and A. R. Butt, "Tarazu: An adaptive end-to-end i/o load-balancing framework for large-scale parallel file systems," *ACM Trans. Storage*, vol. 20, apr 2024.

[3] A. Khan, A. K. Paul, C. Zimmer, S. Oral, S. Dash, S. Atchley, and F. Wang, "Hvac: Removing i/o bottleneck for large-scale deep learning applications," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 324–335, 2022.

[4] Y. Qian, M.-A. Vef, P. Farrell, A. Dilger, X. Li, S. Ihara, Y. Fu, W. Xue, and A. Brinkmann, "Combining buffered I/O and direct I/O in distributed file systems," in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, (Santa Clara, CA), pp. 17–33, USENIX Association, Feb. 2024.

[5] J. L. Bez, A. M. Karimi, A. K. Paul, B. Xie, S. Byna, P. Carns, S. Oral, F. Wang, and J. Hanley, "Access patterns and performance behaviors of multi-layer supercomputer i/o subsystems under production load," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pp. 43–55, 2022.

[6] J. Axboe, "Efficient io with io_uring." https://kernel.dk/io_uring.pdf, 2019.

[7] Z. Ren and A. Trivedi, "Performance characterization of modern storage stacks: Posix i/o, libaio, spdk, and io_uring," CHEOPS '23, (New York, NY, USA), p. 35–45, Association for Computing Machinery, 2023.

[8] Michael Kerrisk, *Linux manual pages*, 6.9.1 ed.

[9] P. Bhattacharya, Pratt and Morgan, "Asynchronous i/o support in linux 2.5," *Proceedings of the Linux Symposium*, 2003.

[10] J. Axboe, "Flexible i/o tester (fio)." https://github.com/axboe/fio. Accessed: 2024-07-13.

[11] I. Lee, "Hpc io benchmark repository." https://github.com/hpc/ior, 2024. Accessed: 07-13-2024.

[12] "Samsung 980 pro 1 tb ssd." https://semiconductor.samsung.com/consumer-storage/internal-ssd/980pro/.