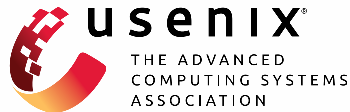


Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs

Gyusun Lee, Seokha Shin, and Wonsuk Song, *Sungkyunkwan University;*
Tae Jun Ham and Jae W. Lee, *Seoul National University;*
Jinkyu Jeong, *Sungkyunkwan University*

https://www.usenix.org/conference/atc19/presentation/lee-gyusun

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs

Gyusun Lee[†], Seokha Shin[*][†], Wonsuk Song[†], Tae Jun Ham[§], Jae W. Lee[§], Jinkyu Jeong[†]
[†]*Sungkyunkwan University,* [§]*Seoul National University*
{gyusun.lee, seokha.shin, wonsuk.song}@csi.skku.edu, {taejunham, jaewlee}@snu.ac.kr, jinkyu@skku.edu

Abstract

Today's ultra-low latency SSDs can deliver an I/O latency of sub-ten microseconds. With this dramatically shrunken device time, operations inside the kernel I/O stack, which were traditionally considered lightweight, are no longer a negligible portion. This motivates us to reexamine the storage I/O stack design and propose an *asynchronous* I/O stack (AIOS), where synchronous operations in the I/O path are replaced by asynchronous ones to overlap I/O-related CPU operations with device I/O. The asynchronous I/O stack leverages a lightweight block layer specialized for NVMe SSDs using the page cache without block I/O scheduling and merging, thereby reducing the sojourn time in the block layer. We prototype the proposed asynchronous I/O stack on the Linux kernel and evaluate it with various workloads. Synthetic FIO benchmarks demonstrate that the application-perceived I/O latency falls into single-digit microseconds for 4 KB random reads on Optane SSD, and the overall I/O latency is reduced by 15–33% across varying block sizes. This I/O latency reduction leads to a significant performance improvement of real-world applications as well: 11–44% IOPS increase on RocksDB and 15–30% throughput improvement on Filebench and OLTP workloads.

1 Introduction

With advances in non-volatile memory technologies, such as flash memory and phase-change memory, ultra-low latency solid-state drives (SSDs) have emerged to deliver extremely low latency and high bandwidth I/O performance. The state-of-the-art non-volatile memory express (NVMe) SSDs, such as Samsung Z-SSD [32], Intel Optane SSD [12] and Toshiba XL-Flash [25], provide sub-ten microseconds of I/O latency and up to 3.0 GB/s of I/O bandwidth [12, 25, 32]. With these ultra-low latency SSDs, the kernel I/O stack accounts for a large fraction in total I/O latency and is becoming a bottleneck to a greater extent in storage access.

[*]Currently at Samsung Electronics

One way to alleviate the I/O stack overhead is to allow user processes to directly access storage devices [6, 16, 27, 28, 49]. While this approach is effective in eliminating I/O stack overheads, it tosses many burdens to applications. For example, applications are required to have their own block management layers [49] or file systems [15, 43, 49] to build useful I/O primitives on top of a simple block-level interface (e.g., BlobFS in SPDK). Providing protections between multiple applications or users is also challenging [6, 16, 28, 43]. These burdens limit the applicability of user-level direct access to storage devices [49].

An alternative, more popular way to alleviate the I/O stack overhead is to optimize the kernel I/O stack. Traditionally, the operating system (OS) is in charge of managing storage and providing file abstractions to applications. To make the kernel more suitable for fast storage devices, many prior work proposed various solutions to reduce the I/O stack overheads. Examples of such prior work include the use of polling mechanism to avoid context switching overheads [5, 47], removal of bottom halves in interrupt handling [24, 35], proposal of scatter/scatter I/O commands [37, 50], simple block I/O scheduling [3, 24], and so on. These proposals are effective in reducing I/O stack overheads, and some of those are adopted by mainstream OS (e.g., I/O stack for NVMe SSDs in Linux).

In our work, we identify new unexplored opportunities to further optimize the I/O latency in storage access. The current I/O stack implementation requires many operations to service a single I/O request. For example, when an application issues a read I/O request, a page is allocated and indexed in a page cache [36]. Then, a DMA mapping is made and several auxiliary data structures (e.g., `bio`, `request`, `iod` in Linux) are allocated and manipulated. The issue here is that these operations occur synchronously before an actual I/O command is issued to the device. With ultra-low latency SSDs, the time it takes to execute these operations is comparable to the actual I/O data transfer time. In such case, overlapping those operations with the data transfer can substantially reduce the end-to-end I/O latency.

To this end, this paper proposes an *asynchronous* I/O

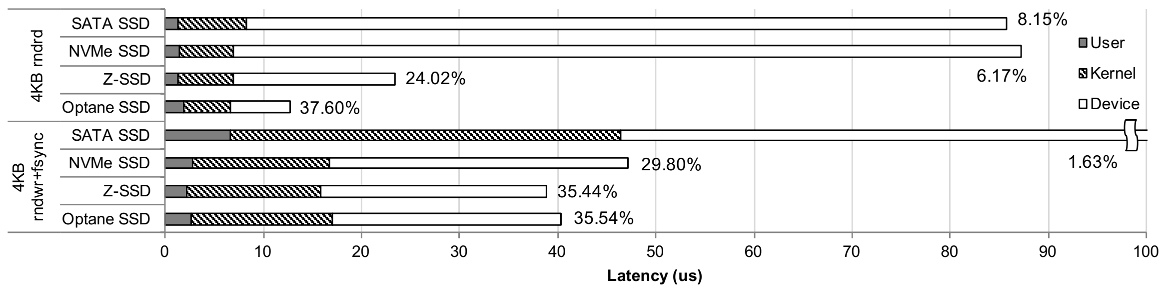


Figure 1: I/O latency and its breakdown with various storage devices. The numbers beside each bar denote the relative fraction of kernel time in the total I/O latency.

stack (AIOS), a low-latency I/O stack for ultra-low latency SSDs. Through a careful analysis of synchronous, hence latency-critical, system call implementations (i.e., `read()` and `fsync()`) in the Linux kernel, we identify I/O-related CPU operations that can be overlapped with device I/O operations and modify the Linux kernel to execute such CPU operations while the I/O device is processing a request. To further reduce the CPU overhead, we also introduce a lightweight block I/O layer (LBIO) specialized for NVMe-based SSDs, which enables the kernel to spend considerably less time in the block I/O layer. Our evaluation demonstrates that AIOS achieves up to 33% latency reduction for random reads and 31% latency reduction for random writes on FIO benchmarks [2]. Also, AIOS enables various real-world applications (e.g., a key-value store, database) to achieve higher throughput. Our contributions are summarized as follows:

- We provide a detailed analysis of the Linux kernel I/O stack operations and identify CPU operations that can overlap with device I/O operations (Section 2).
- We propose the lightweight block I/O layer (LBIO) specialized for modern NVMe-based SSD devices, which offers notably lower latency than the vanilla Linux kernel block layer (Section 3.1).
- We propose the asynchronous I/O stack for read and fsync paths in which CPU operations are overlapped with device I/O operations, thereby reducing the completion time of the read and fsync system calls (Section 3.2 and 3.3).
- We provide a detailed evaluation of the proposed schemes to show the latency reduction of up to 33% for random reads and 31% for random writes on FIO benchmarks [2] and substantial throughput increase on real-world workloads: 11–44% on RocksDB [10] and 15–30% on Filebench [40] and OLTP [18] workloads (Section 4).

2 Background and Motivation

2.1 ULL SSDs and I/O Stack Overheads

Storage performance is important in computer systems as data should be continuously supplied to a CPU to not stall the pipeline. Traditionally, storage devices have been much slower than CPUs, and this wide performance gap has existed for decades [26]. However, the recent introduction of modern storage devices is rapidly narrowing this gap. For example, today's ultra-low latency (ULL) NVMe SSDs, such as Samsung's Z-SSD [32], Intel's Optane SSD [12], and Toshiba's XL-Flash [25], can achieve sub-ten microseconds of I/O latency, which is orders of magnitude faster than that of traditional disks.

With such ultra-low latency SSDs, the kernel I/O stack [11, 38] no longer takes a negligible portion in the total I/O latency. Figure 1 shows the I/O latency and its breakdown for 4 KB random read and random write + `fsync` workloads on various SSDs[1]. The figure shows that ultra-low latency SSDs achieve substantially lower I/O latency than conventional SSDs. Specifically, their device I/O time is much lower than that of the conventional SSDs. On the other hand, the amount of time spent in the kernel does not change across different SSDs. As a result, the fraction of the time spent in the kernel becomes more substantial (i.e., up to 37.6% and 35.5% for the read and write workloads, respectively).

An I/O stack is composed of many layers [17]. A virtual file system (VFS) layer provides an abstraction of underlying file systems. A page cache layer provides caching of file data. A file system layer provides file system-specific implementations on top of the block storage. A block layer provides OS-level block request/response management and block I/O scheduling. Finally, a device driver handles device-specific I/O command submission and completion. In this paper, we target two latency-sensitive I/O paths (`read()` and `write()+fsync()`) in the Linux kernel and Ext4 file system with NVMe-based SSDs, since they are widely adopted system configurations from the mobile [13] to enterprise [33, 51].

2.2 Read Path

2.2.1 Vanilla Read Path Behavior

Figure 2 briefly describes the read path in the Linux kernel. Buffered read system calls (e.g., `read()` and `pread()` without `O_DIRECT`) fall into the VFS function (`buffered_read()` in the figure), which is an entry point to the page cache layer.

Page cache. Upon a cache miss (Line 3-4), the function

[1]The detailed evaluation configurations can be found in Section 4.1. Note that all the tested NVMe SSDs feature a non-volatile write cache, hence showing low write latency.

```
1  void buffered_read(file, begin, end, buf) {
2   for (idx=begin; idx<end; idx++) {
3    page = page_cache_lookup(idx)
4    if (!page) {
5     page_cache_sync_readahead(file, idx, end)
6     page = page_cache_lookup(idx)
7    } else {
8     page_cache_async_readahead(file, idx, end)
9    }
10   lock_page(page)
11   memcpy(buf, page, PAGE_SIZE)
12   buf += PAGE_SIZE
13  }
14 }
15
16 void page_cache_readahead(file, begin, end) {
17  init_list(pages)
18  for (idx=begin; idx<end; idx++) {
19   if (!page_cache_lookup(file, idx)) {
20    page = alloc_page()
21    page->idx = idx
22    push(pages, page)
23   }
24  }
25  readpages(file, pages)
26 }
27
28 void ext4_readpages(file, pages) {
29  blk_start_plug()
30  for (page : pages) {
31   add_to_page_cache(page, page->idx, file)
32   lba = ext4_map_blocks(file, page->idx)
33   bio = alloc_bio(page, lba)
34   submit_bio(bio)
35  }
36  blk_finish_plug()
37 }
```

Figure 2: Pseudocode for Linux kernel read path.

page_cache_sync_readahead() is called, in which missing file blocks are read into the page cache. It identifies all missing indices within the requested file range (Line 18-19), allocates pages and associates the pages with the missing indices (Line 20-21). Finally, it requests the file system to read the missing pages (Line 25).

File system. File systems have their own implementation of readpages(), but their behaviors are similar to each other. In Ext4 ext4_readpages() inserts each page into the page cache (Line 30-31), retrieves the logical block address (LBA) of the page (Line 32) and issues a block request to the underlying block layer (Line 33-34).

Linux batches several block requests issued by a thread in order to increase the efficiency of request handling in the underlying layers (also known as *queue plugging* [4]). When blk_start_plug() is called (Line 29), block requests are collected in a plug list of the current thread. When blk_finish_plug() (Line 36) is called or the current thread is context-switched, the collected requests are flushed to the block I/O scheduler.

After issuing an I/O request to a storage device, the thread rewinds its call stack and becomes blocked at the function lock_page() (Line 10). When the I/O request is completed,

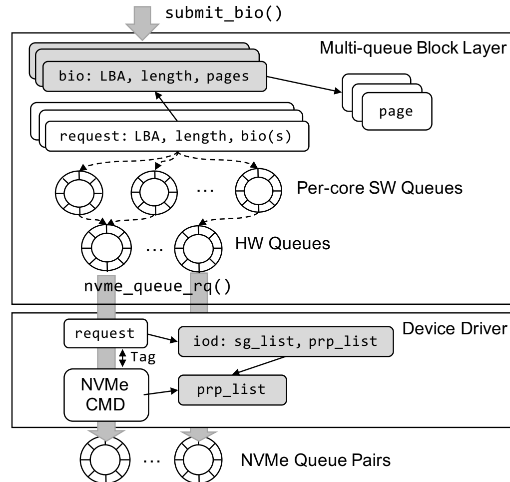


Figure 3: The overview of the multi-queue block layer. The shaded rectangles are dynamically allocated objects.

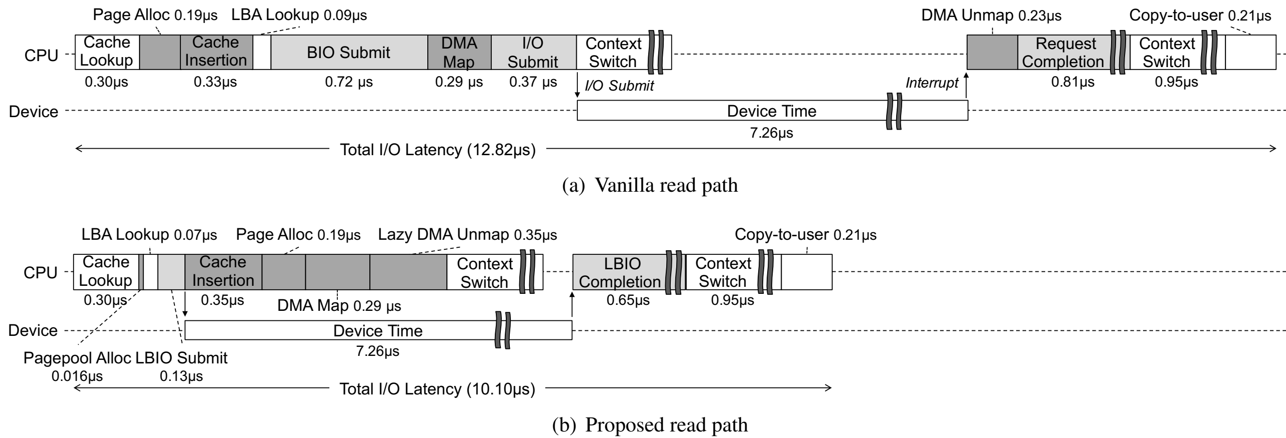
the interrupt handler releases the lock of the page, which wakes up the blocked thread. Finally, the cached data are copied to the user buffer (Line 11-12).

Block layer. Figure 3 shows the overview of the multi-queue block layer, which is the default block layer for NVMe SSDs in the Linux kernel, and the device driver layer. In the block layer, a bio object is allocated using a slab allocator and initialized to contain the information of a single block request (i.e., LBA, I/O size and pages to copy-in) (Line 33). Then, submit_bio() (Line 34) transforms the bio object to a request object and inserts the request object into request queues, where I/O merging and scheduling are performed [3, 8]. The request object passes through a per-core software queue (ctx) and hardware queue (hctx) and eventually reaches the device driver layer.

Device driver. A request is dispatched to the device driver using nvme_queue_rq(). It first allocates an iod object, a structure having a scatter/gather list, and uses it to perform DMA (direct memory access) mapping, hence allocating I/O virtual addresses (or DMA addresses) to the pages in the dispatched request. Then, a prp_list, which contains physical region pages (PRP) in the NVMe protocol, is allocated and filled with the allocated DMA addresses. Finally, an NVMe command is created using the request and the prp_list and issued to an NVMe submission queue. Upon I/O completion, the interrupt handler unmaps DMA addresses of the pages and calls a completion function, which eventually wakes up the blocked thread. While the above describes the basic operations in the read path, the roles of the block and device driver layers are identical in the write path.

2.2.2 Motivation for Asynchronous Read Path

Figure 4(a) summarizes the operations and their execution times in the read path explained in Section 2.2.1. The main problem is that a single read I/O path has many operations



(a) Vanilla read path

(b) Proposed read path

Figure 4: The operations and their execution times in the read paths during 4 KB random read on Optane SSD. (drawn to scale in time)

that occur synchronously to an actual device I/O operation. This synchronous design is common and intuitive and works well with slow storage devices because in such cases, the time spent on CPU is negligible compared to the total I/O latency. However, with ultra-low latency SSDs, the amount of CPU time spent on each operation becomes a significant portion of the total I/O latency.

Table 1 summarizes the ratio of each operation to the total kernel time. With small I/O sizes, the context switching is the most time-consuming operation. In this case, it is possible to reduce the overhead through the use of polling [5, 47]. The copy-to-user is another costly operation but is a necessary operation when file data is backed by the page cache. If we exclude these two operations, the remaining operations account for 45-50% of the kernel time.

Upon careful analysis of the code, we find that many of the remaining operations do not have to be performed before or after the device I/O time. In fact, such operations can be performed while the device I/O operation is happening since such operations are mostly independent of the device I/O operation. This motivates us to overlap such operations with the device I/O operation as sketched in Figure 4(b) (shaded in dark gray).

2.3 Write Path

2.3.1 Vanilla Write Path Behavior

Buffered write system calls (e.g., `write()`) usually buffer the modified data in the page cache in memory. When an application calls `fsync()`, the kernel actually performs write I/Os to synchronize the dirtied data with a storage device.

The buffered write path has no opportunity to overlap computation with I/O because it does not perform any I/O operation. On the other hand, `fsync` accompanies several I/O operations due to the writeback of dirtied data as well as a crash consistency mechanism in a file system (e.g., file system journaling). Since `fsync` most heavily affects the application

Layer	Action	Lines in Figure 2	% in kernel time
Page cache	Missing page lookup Page allocation	Line 18-24	9–10.8%
	Copy-to-user	Line 11-12	4.5–12%
File system	Page cache insertion LBA retrieval `bio` alloc/submit Make `request` from `bio`	Line 30-35	26–28.5%
Block	I/O scheduling (noop)	Line 36	10–11%
Driver	DMA mapping/unmapping NVMe command submit		
Scheduler	Context switch (2 times)	Line 10	25–41.5%

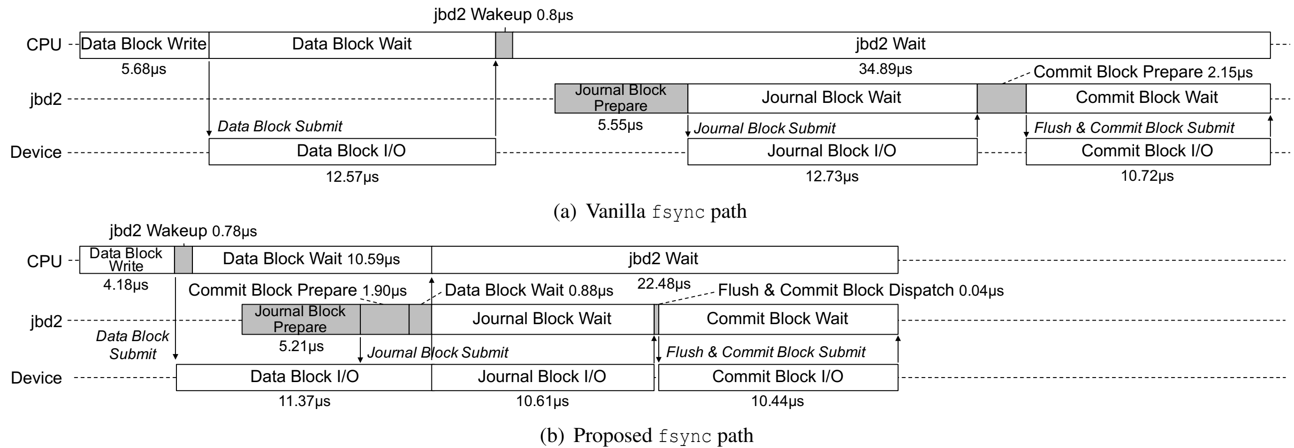
Table 1: Summary of operations and their fractions in kernel time in the read path. (4–16 KB FIO rndrd on Optane SSD)

performance in the write path, we examine it in more detail.

Figure 5(a) shows the operations and their execution times during an `fsync` call on Ext4 file system using ordered journaling. First, an application thread issues write I/Os for dirty file blocks and waits for them to complete. Then, the application thread wakes up a journaling thread (`jbd2` in Ext4) to commit the file system transaction. It first prepares the write of modified metadata (*journal block* in the figure) onto the journal area and issues the write I/O. Then, it waits for the completion of the write. Once completed, it prepares the write of a commit block and issues the write I/O. A flush command is enforced between the journal block write and the commit block write to enforce the ordering of writes [45]. Hence, total three device I/O operations occur for a single `fsync` call.

2.3.2 Motivation for Asynchronous Write Path

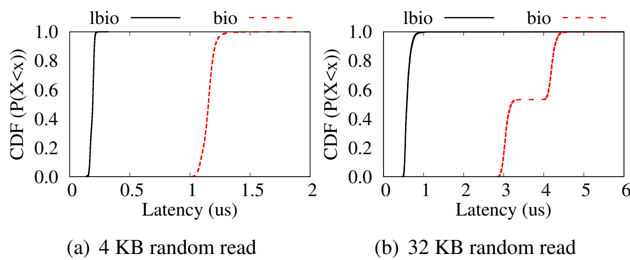
As in the case of the read path, there is also an opportunity to overlap the device I/O operations with the computation parts in the `fsync` path. As shown in Figure 5(a), the journaling thread performs I/O preparation and I/O waiting synchronously. Each I/O preparation includes assigning blocks in the journal area to write on, allocating buffer pages, allo-



(a) Vanilla `fsync` path

(b) Proposed `fsync` path

Figure 5: The operations and their execution times in the `fsync` paths during 4 KB random write with `fsync` on Optane SSD. (drawn to scale in time)



(a) 4 KB random read

(b) 32 KB random read

Figure 6: The CDF of the block I/O submission latency in the Linux block layer (*bio*) and the proposed lightweight block layer (*lbio*) on Optane SSD.

cating/submitting a `bio` object, assigning DMA address, and so forth. If these CPU operations are overlapped with the previous device I/O time, the total latency of the `fsync` system call can be greatly reduced as shown in Figure 5(b).

2.4 Motivation for Lightweight Block Layer

The Linux kernel uses the multi-queue block layer for NVMe SSDs by default to scale well with multiple command queues and multi-core CPUs [3]. This block layer provides functionality like block I/O submission/completion, request merging/reordering, I/O scheduling and I/O command tagging [3]. While these features are necessary for general block I/O management, they delay the submission time of an I/O command to a storage device.

Figure 6 shows the block I/O submission latency, time from allocating a `bio` object to dispatching an I/O command to a device (denoted as *bio*); the figure also includes the same measurement using the proposed lightweight block layer in Section 3.1 (denoted as *lbio*). We use two I/O sizes, 4 KB and 32 KB, to minimize and maximize the number of dynamic memory allocations during I/O submissions, respectively. Considering that the device time for a 4 KB read is around 7.3 μs on ultra-low latency SSDs, the amount of time spent in the block layer is about 15% of the device time, which is a non-negligible portion.

While block I/O submission/completion and I/O command tagging are necessary features, request merging/reordering and I/O scheduling are not significant. The multi-queue block layer supports various I/O schedulers [8] but its default configuration is *noop* since many studies report that I/O scheduling is ineffective for reducing I/O latency for latency-critical applications on fast storage devices [34, 46, 51]. I/O scheduling can also be replaced by the device-side I/O scheduling capability [14]. The effectiveness of request merging/reordering is also questionable in ultra-low latency SSDs because of their high random access performance and the low probability to find adjacent or identical block requests.

Based on these intuitions, we propose to simplify the roles of the block layer and make it specialized for our asynchronous I/O stack to minimize its I/O submission delay.

3 Asynchronous I/O Stack

The proposed asynchronous I/O stack (AIOS) consists of two components: the lightweight block I/O layer (LBIO) and the modified I/O stack that overlaps I/O-related computations with the device I/O operations. This section first explains LBIO and then explains the modified read and write paths.

3.1 Lightweight Block I/O Layer

To minimize the software overheads in the block layer, we design a lightweight block I/O layer (or LBIO), a scalable, lightweight alternative to the existing multi-queue block layer. LBIO is designed for low-latency NVMe SSDs and supports only I/O submission/completion and I/O command tagging. Figure 7 shows the overview of our proposed LBIO.

Unlike the original multi-queue block layer, LBIO uses a single memory object, `lbio`, to represent a single block I/O request, thereby eliminating the time-consuming `bio`-to-`request` transformation in the original block layer. Each

`lbio` object contains LBA, I/O length, pages to copy-in and DMA addresses of the pages. Containing DMA addresses in `lbio` leverages the asynchronous DMA mapping feature explained in the following sections. An `lbio` only supports 4 KB-aligned DMA address with I/O length of multiple sectors to simplify the codes initializing and submitting block I/O requests. This approach is viable with the assumption of using the page cache layer. Similar to the original block layer, LBIO supports queue plugging to batch multiple block I/O requests issued by a single thread.

LBIO has a global `lbio` two-dimensional array whose row is dedicated to each core, and a group of rows is assigned to each NVMe queue pair as shown in Figure 7. For example, if a system has 8 cores and 4 NVMe queue pairs, each `lbio` array row is one-to-one mapped to each core and two consecutive rows are mapped to an NVMe queue pair. When the number of NVMe queue pairs is equal to the number of cores, lockless `lbio` object allocations and NVMe command submissions are possible, as in the existing multi-queue block layer. The index of an `lbio` object in the global array is used as a tag in an NVMe command. This eliminates the time-consuming tag allocation in the original block layer.

Once an `lbio` is submitted, the thread directly calls `nvme_queue_lbio()` to dispatch an NVMe I/O command. Note that LBIO does not perform I/O merging or I/O scheduling, and thus reduces I/O submission delay significantly. Without the I/O merging, it is possible for two or more `lbio`'s to access the same logical block. This potentially happens in the read path and is resolved by the page cache layer (see Section 3.2). However, this does not happen in the write path because the page cache synchronizes writeback of dirty pages.

Figure 6 shows the reduced I/O submission latency with LBIO. On average, a block I/O request in LBIO takes only 0.18–0.60 μs, which is 83.4%–84.4% shorter latency compared to that of the original block layer.

3.2 Read Path

In addition to the introduction of LBIO, our approach reduces the I/O latency of the read path by detaching synchronous operations from the critical path as sketched in Figure 4(b). The following subsections describe each relocated operation and additional work to support the relocation.

3.2.1 Preloading Extent Tree

For a read operation, retrieving LBAs corresponding to the missing file blocks is a necessary step to issue a block request and thus this operation should be in the critical path. Instead of taking this step off the critical path, our proposal focuses on reducing its latency itself. The implementation of the Linux Ext4 file system caches logical to physical file block mappings in memory, and this cache is called extent status tree [19]. When a mapping can be found in the cache, obtaining an LBA takes a relatively short time; however, when the mapping is

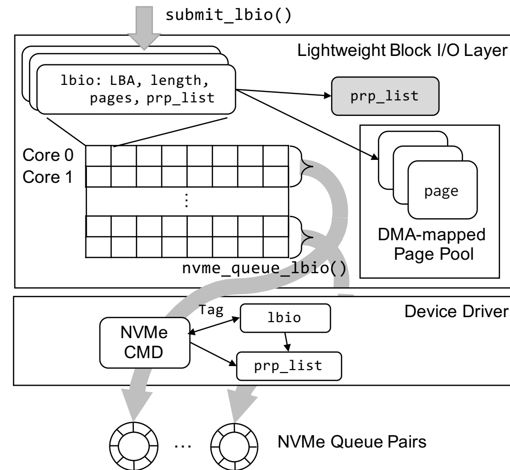


Figure 7: The proposed lightweight block I/O layer (LBIO). Shaded objects are dynamically allocated.

not found, the system has to issue an I/O request to read the missing mapping block and thus incurs much longer delay.

To avoid this unnecessary overhead, we adopt a plane separation approach [28]. In the control plane (e.g., file open), the entire mapping information is preloaded in memory. By doing so, the data plane (e.g., read and write) can avoid the latency delay caused by a mapping cache miss. The memory costs of caching an entire tree can be high; the worst case overhead is 0.03% of the file size in our evaluation. However, when there is little free memory, the extent cache evicts unlikely-used tree nodes to secure free memory [19]. To reduce the space overhead even further, this technique can be selectively applied to files requiring low-latency access.

3.2.2 Asynchronous Page Allocation/DMA Mapping

Preparation of free pages is another essential operation in the read path. In the original read path, a page allocator of the kernel performs this task, and it consumes many CPU cycles. For example, a single page allocation takes 0.19 μs on average in our system as shown in Figure 4(a). Similarly, assigning a DMA address to each page (DMA mapping) takes a large number of CPU cycles (754 cycles or 0.29 μs). Our approach is to take these operations off from the critical path and perform them while the device I/O operation is happening.

To this end, we maintain a small set of DMA-mapped free pages (a linked list of 4 KB DMA-mapped pages) for each core. With this structure, only a few memory instructions are necessary to get free pages from the pool (*Pagepool Alloc* in Figure 4(b)). The consumed pages are refilled by invoking page allocation and DMA mapping while the device I/O operation is occurring. This effectively hides the time for both page allocation and DMA mapping from the application-perceived I/O latency as shown in the figure. Note that, when the number of free pages in the pool is smaller than the read request size, page allocation and DMA mapping happens synchronously as in the vanilla kernel case.

3.2.3 Lazy Page Cache Indexing

Insertion of a page into a page cache index structure is another source of the large kernel I/O stack latency. Our approach is to overlap this operation with the device I/O operation while resolving the potentially duplicated I/O submissions.

In the vanilla kernel, the page cache works as a synchronization point that determines whether a block I/O request for a file can be issued or not. File blocks whose cache pages are successfully inserted into the page cache are allowed to make block requests (Line 31 in Figure 2), and a spinlock is used to protect the page cache from concurrent updates. Consequently, no duplicated I/O submission occurs for the same file block.

However, if we delay the page cache insertion operation to a point after submitting an I/O command to a device, it is possible for another thread to miss on the same file block and to issue a duplicate block request. To be exact, this happens if another thread accesses the page cache after the I/O request is submitted but before the page cache entry is updated.

Our solution is to allow duplicated block requests but resolve them at the request completion phase. Although there are multiple block requests associated with the same file block, only a single page is indexed in the page cache. Then, our scheme marks other pages as *abandoned*. The interrupt handler frees a page associated with the completed block request if it is marked *abandoned*.

3.2.4 Lazy DMA Unmapping

The last long-latency operation in the read path is DMA unmapping that occurs after the device I/O request is completed. The vanilla read path handles this in the interrupt handler, which is also in the critical path. Our scheme delays this operation to when a system is idle or waiting for another I/O request (*Lazy DMA unmap* in Figure 4(b)).

Note that this scheme prolongs the time window in which the DMA buffer is accessible by the storage device. This is essentially an extended version of the deferred protection scheme used in Linux by default [20]. Deferring the DMA unmapping (either in our scheme or in Linux) may potentially create a vulnerability window from a device-side DMA attack. However, with an assumption that the kernel and the device are neither malicious nor vulnerable, the deferred protection causes no problem [20]. If the assumption is not viable, users can disable the lazy DMA unmapping.

3.3 Write and fsync Path

As explained in Section 2.3.1, an `fsync` system call entails multiple I/O operations, and thus it is not possible to reuse the same scheme we proposed for the read path. For example, by the time `fsync` happens, pages are already allocated and indexed in the page cache. Instead of overlapping the I/O-related computation with individual device I/O operation, we focus on applying the overlapping idea to the entire file system journaling process.

Specifically, we overlap the computation parts in the journaling thread with the previous I/O operations in the same write path. As shown in Figure 5(a), there are two I/O preparation operations: journal block preparation and commit block preparation. Each preparation operation includes allocating buffer pages, allocating a block on the journal area, calculating the checksum and computation operations within the block and device driver layers. Since these operations only modify in-memory data structures, they have no dependency on the previous I/O operation in the same write path. Note that, at any given time, only a single file system transaction can be committed. While a transaction is in the middle of commit, no other file system changes can be entangled to the current transaction being committed. Hence, if the ordering constraint, which allows the write of a commit block only after the data blocks and journal blocks are made durable on the storage media, is guaranteed, our approach can provide the same crash consistency semantic provided by the vanilla write path.

To this end, we change the `fsync` path as shown in Figure 5(b). Upon an `fsync` system call, an application thread issues the writeback of dirty data pages first. Then, it wakes up the journaling thread in advance to overlap the data block I/Os with the computation parts in the journaling thread. The application thread finally waits for the completion of the writeback I/Os as well as the completion of the journal commit. While the data block I/O operations are happening, the journaling thread prepares the journal block writes and issues their write I/Os. Then, it prepares the commit block write and waits for the completion of all the previous I/O operations associated with the current transaction. Once completed, it sends a flush command to the storage device to make all the previous I/Os durable and finally issues a write I/O of the commit block using a write-through I/O command (e.g., FUA in SATA). After finishing the commit block write, the journaling thread finally wakes up the application thread.

When an `fsync` call does not entail a file system transaction, it is not possible to overlap computation with I/O operation. In this case, the use of LBIO reduces its I/O latency.

3.4 Implementation

The proposed scheme is implemented in the Linux kernel version 5.0.5. A new file open flag, `O_AIOS`, is introduced to use the proposed I/O stack selectively. The current implementation supports `read()`, `pread()`, `fsync()`, and `fdatasync()` system calls. For now, other asynchronous or direct I/O APIs are not supported.

The LBIO layer shares the NVMe queue pairs used in the original block layer. The spinlock of each queue pair provides mutual exclusion between LBIO and the original block layer. The most significant bit of a 16-bit tag is reserved to distin-

	Object	Linux block	LBIO
128 KB block request	(l)bio + (l)bio_vec	648 bytes	704 bytes
	request	384 bytes	
	iod	974 bytes	
	prp_list	4096 bytes	4096 bytes
	Total	6104 bytes	4800 bytes
Statically allocated (per-core)	request pool	412 KB	
	lbio array		192 KB
	free page pool		256 KB

Table 2: Memory cost comparison

guish the two block layers. Since the NVMe SSDs used for evaluation supports 512 or 1024 entries for each queue, the remaining 15 bits are sufficient for command tagging.

Table 2 summarizes the memory cost of our scheme and the original block layer. To support a single 128 KB block request, both layers use a comparable amount of memory for (l)bio, (l)bio_vec, and prp_list objects. However, the original block layer requires two additional memory objects: request and iod, and thus requires extra memory compared to LBIO.

As for the cost of statically allocated memory, the original block layer maintains a pool of request objects (1024 objects with 1024 I/O queue depth), which requires 412 KB memory per core. LBIO replaces the per-core request pool with the per-core lbio row of size 192 KB. Meanwhile, our scheme also maintains a pool of DMA-mapped free pages. We maintain 64 free pages for each free page pool, hence consuming additional 256 KB memory per core.

In order to implement the AIOS fsync path, the jbd2 wakeup routine in ext4_sync_file() is relocated to the position between data block write and data block wait. The function jbd2_journal_commit_transaction() is also modified to implement our scheme. The routine to prepare a commit block is moved to the position before the waiting routine for the journal block writes. The waiting routine for data block writes (using t_inode_list) is also relocated to the position before the waiting for journal block writes. The routine to issue commit block write I/O (i.e., submit_bh() in the vanilla path) is split into two routines: one for allocating an lbio and mapping DMA address, and the other for submitting an I/O command to a device (i.e., nvme_queue_lbio()). With this separation, AIOS can control the time to submit the commit block I/O so that it can satisfy the ordering constraints, while allowing the overlap of block request-related computations (e.g., DMA mapping) with the previous I/O operations.

4 Evaluation

4.1 Methodology

We use Dell R730 Server machine with Intel Xeon E5-2640 CPU and 32 GB DDR4 memory for our experiments. For the ultra-low latency storage devices, we evaluate both Samsung Z-SSD and Intel Optane SSD; both integrate a non-volatile write cache, which ignores flush and FUA commands in the block layer. We implement our proposed scheme AIOS on Linux kernel 5.0.5, denoted as *AIOS*. The baseline is the vanilla Linux kernel 5.0.5 using the Ext4 file system, denoted as *vanilla*. Table 3 summarizes our experimental setup.

Server		Dell R730
OS		Ubuntu 16.04.4
Base kernel		Linux 5.0.5
CPU		Intel Xeon E5-2640v3 2.6 GHz 8 cores
Memory		DDR4 32 GB
Storage devices	*Z-SSD*	Samsung SZ985 800 GB
	Optane SSD	Intel Optane 905P 960 GB
	NVMe SSD	Samsung PM1725 1.6 TB
	SATA SSD	Samsung 860 Pro 512 GB

Table 3: Experimental configuration

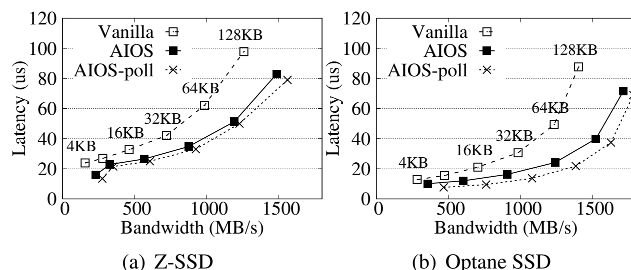


Figure 8: FIO single-thread random read latency and throughput with varying block sizes.

For evaluation, we utilize both the synthetic microbenchmark and real-world workloads. For the synthetic microbenchmark, we use FIO [2] using the *sync* engine with varying I/O request sizes, I/O types, the number of threads and so forth. For real-world workloads we utilize various applications such as key-value store (RocksDB [10]), file-system benchmark (Filebench-varmail [40]), and OLTP workload (Sysbench-OLTP-insert [18] on MySQL [22]). Specifically, we run *readrandom* and *fillsync* workloads of the DBbench [30] on RocksDB; each representing a read-intensive case and fdatasync-intensive case, respectively. Filebench-varmail is fsync-intensive, and Sysbench-OLTP-insert is fdatasync-intensive.

4.2 Microbenchmark

4.2.1 Read Performance

Random read latency. Figure 8 shows the effect of AIOS on FIO random read latency and throughput with varying block sizes. The figure shows that AIOS reduces random read latency by 15–33% when compared to the vanilla kernel on both Z-SSD and Optane SSD. In general, a larger block size results in greater latency reduction because a larger portion of the read-related kernel computation gets overlapped with the device I/O operation (see Figure 4). One important note is that AIOS achieves single-digit microseconds latency for a 4 KB random read on Optane SSD, which was previously not possible due to substantial read path overheads.

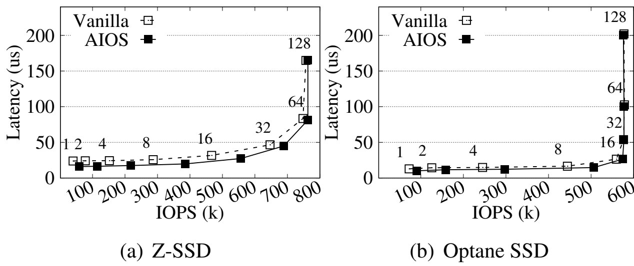


Figure 9: FIO 4 KB random read latency and throughput (in IOPS) with varying the number of threads.

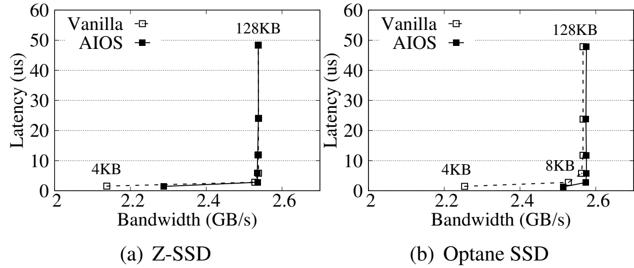


Figure 10: Single-thread FIO sequential read latency and bandwidth with varying block sizes.

Polling vs. interrupt. Figure 8 also shows the impact of I/O completion scheme to the read latency. In the figure, *AIOS-poll* denotes the AIOS scheme using not interrupt but polling as its I/O completion method. In general, polling is better than interrupt in terms of latency because it eliminates context switches [47]; Table 1 has shown that context switches account for the largest fraction in kernel time. With small I/O sizes, the latency reduction from polling is comparable to that from AIOS. However, with large I/O sizes, the latency reduction from AIOS is greater than that from polling because of I/O-computation overlap to a greater extent. Please note that the interrupt mode is used by default in the rest of this section.

Random read throughput. Figure 9 shows the FIO 4 KB random read latency and throughput in I/O operations per second (IOPS) with varying the number of threads. Here, each thread is set to issue a single I/O request at a time (i.e., queue depth is one). In this setting, a large number of threads means that a large number of I/O requests are outstanding, hence mimicking high queue-depth I/O. As shown in the figure, both Z-SSD and Optane SSD achieve notably higher IOPS (i.e., up to 47.1% on Z-SSD and 26.3% on Optane SSD) than the baseline when the number of threads is less than 64. From that point, the device bandwidth gets saturated, and thus AIOS does not result in additional performance improvement.

Sequential read bandwidth. Figure 10 shows the effect of AIOS on a single-thread sequential read workload with varying block sizes. Because the workload uses buffered reads, the readahead mechanism in Linux prefetches data blocks with large block size (128 KB) into the page cache. This results in high sustained bandwidth. In both Z-SSD and Optane SSD

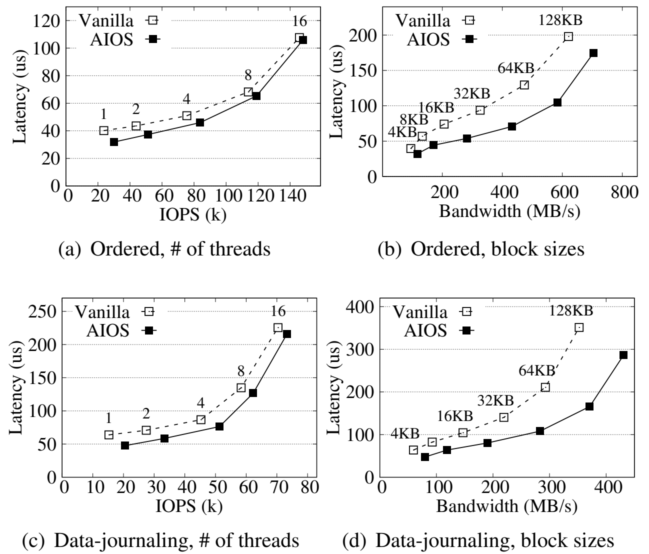


Figure 11: Fsync performance with different journaling modes, number of threads and block sizes on Optane SSD.

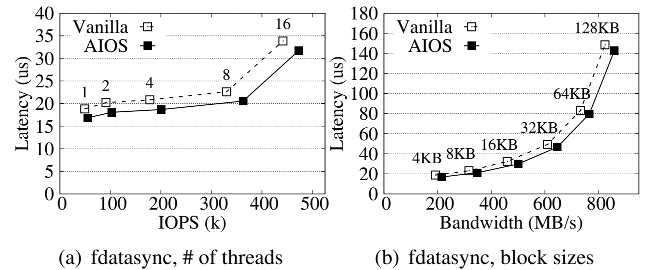


Figure 12: Fdatasync performance with ordered mode with varying the number of threads and block sizes on Optane SSD.

cases, AIOS enables higher sustained bandwidth usage only for 4 KB block reads. For 16–128 KB blocks, AIOS does not result in any bandwidth improvement because the baseline scheme already reaches peak bandwidth.

4.2.2 Write Performance

Fsync performance. Figure 11 shows the performance impact of AIOS on FIO 4 KB random write followed by fsync workload. Here, we evaluate two journaling modes: ordered mode and data-journaling mode. With a single thread, our scheme shows IOPS improvement by up to 27% and 34% and latency reduction by up to 21% and 26% in the ordered mode and data journaling mode, respectively. With increasing the number of threads, natural overlap between computation and I/O occurs, thereby diminishing the performance benefit of our scheme. With large block sizes, the long I/O-computation overlapping happens, thereby widening the absolute performance gap between our scheme and the baseline. In the data-journaling mode, the length of the overlapped portion is longer than that of the ordered mode, and thus its latency advantage

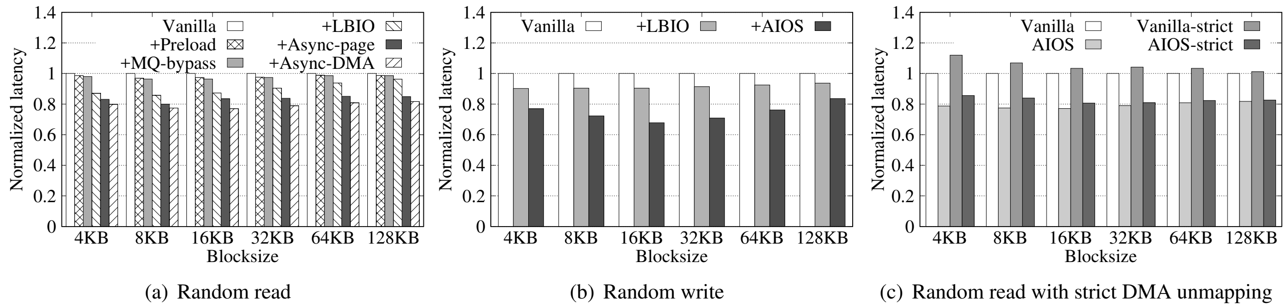


Figure 13: Normalized Latency of FIO 4 KB random read or write workloads with varying optimization levels on Optane SSD.

is slightly larger than that of the ordered mode.

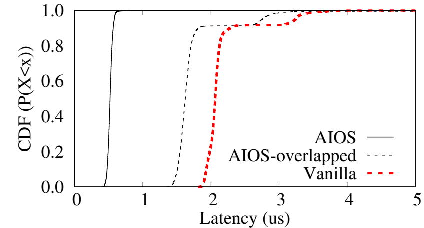
Fdatasync performance. Figure 12 shows the performance impact of AIOS on FIO 4 KB random write followed by fdatasync under the ordered mode. Unlike fsync, fdatasync does not journal metadata if the metadata has not changed; hence showing fewer performance gains than the fsync cases. Our AIOS presents up to 12% IOPS increase with a single thread. AIOS shows up to 10.5% latency decrease on the 4 KB random write workload using a single thread.

4.2.3 Performance Analysis

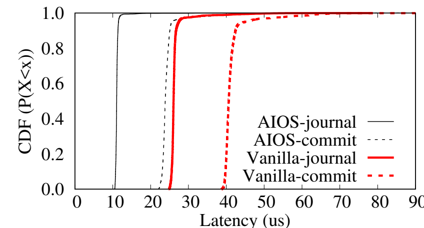
Read latency. Figure 13(a) compares the 4 KB random read latency on Optane SSD with varying optimization levels. The leftmost bar represents the baseline, and the next five bars represent the latency with cumulatively applying the optimizations presented throughout the paper. First, the preloading of extent tree (*+Preload*, Section 3.2.1) shows up to 3.1% latency reduction. Second, bypassing the block multi-queue scheduling (*+MQ-bypass*) provides 1.1% of additional latency reduction. The complete use of LBIO (*+LBIO*, Section 3.1) reduces I/O latency additionally by up to 12.5% because of its low overhead. By overlapping only cache indexing (Section 3.2.3) and page allocation (Section 3.2.2) with device I/O operation, the latency is reduced by up to 11.9% (*+Async-page*). Finally, the asynchronous DMA mapping/unmapping (*+Async-DMA*, Section 3.2.4) further reduces the I/O latency by up to 7.8%. The latency benefit of LBIO is similar across different block sizes. The benefit of the asynchronous operations, however, increases as the block size increases.

Write latency. Figure 13(b) compares the 4 KB random write+fsync latency on Optane SSD with varying optimization levels. For the write operation, the use of LBIO (*+LBIO*) shows up to 9.8% of latency reduction. Our asynchronous write path (*+AIOS*, Section 3.3) achieves additional latency reduction by up to 24.4%.

Cost of safety. Figure 13(c) shows the latency penalty of disabling the deferred DMA unmapping (i.e., unmapping DMA addresses immediately after I/O completion). *AIOS-strict* denotes our scheme without the lazy DMA unmapping (Section 3.2.4). For a fair comparison, we also measured the performance of the baseline without the deferred DMA unmapping [20] (denoted as *Vanilla-strict*). As shown in the figure, the use of strict DMA unmapping incurs a slight increase in I/O latency for both schemes by 1.2–11.9%.



(a) Latency from `read` entry to I/O cmd. submit

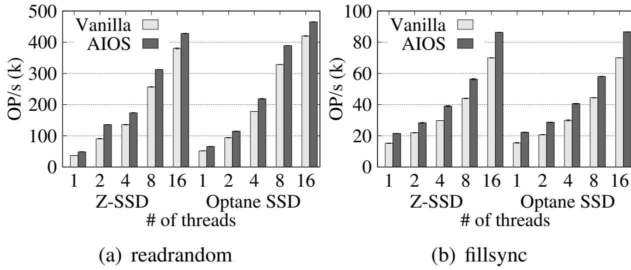


(b) Latency from `fsync` entry to each I/O cmd. submit (journal block and commit block)

Figure 14: CDF of I/O command submission latencies in 4 KB random read and 4 KB random write+fsync workloads on Optane SSD.

Overlapping analysis. One of our key ideas is to overlap computations with I/O so that the I/O command can be submitted as early as possible. To clarify this behavior, we measured the latency to submit I/O command(s) to a storage device and present the cumulative distribution function (CDF) of latencies. Figure 14(a) shows the time between the `read` system call entry and the I/O command submission. We also show the time to complete the overlapped operations in our scheme (denoted as *AIOS-overlapped*). As shown in the figure, the I/O submission latency is greatly reduced to 1.63 μs on average (75% reduction compared to the baseline). Also, note that the time to complete the overlapped operations is earlier than the I/O submission latency in the baseline. This is because of the additional latency reduction achieved by LBIO.

Similar measurement is also made on the write paths. Figure 14(b) shows the time between the `fsync` system call entry and the I/O command submission for journal block(s) and commit block (denoted as *-journal* and *-commit*, respectively).



(a) readrandom (b) fillsync

Figure 15: DBbench on RocksDB Performance (IOPS) with varying the number of threads.

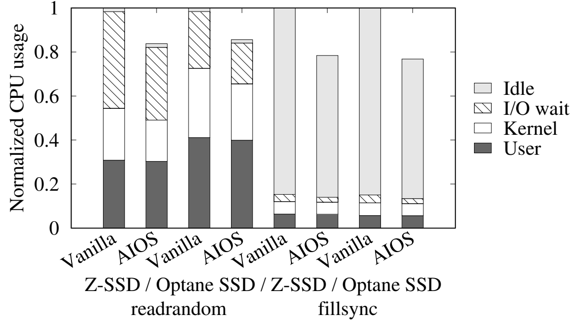


Figure 16: Normalized CPU usage breakdown for DBbench (*readrandom* and *fillsync*) on RocksDB using eight threads.

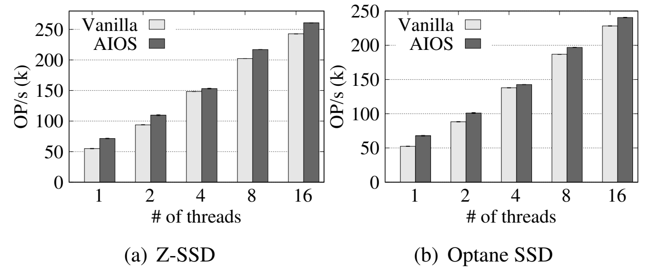
As shown in the figure, our scheme brings the I/O command submission times forward. Interestingly, the time to submit commit block I/O in AIOS is even earlier than the time to submit journal block I/O in the baseline.

4.3 Real-world Applications

4.3.1 Key-value Store

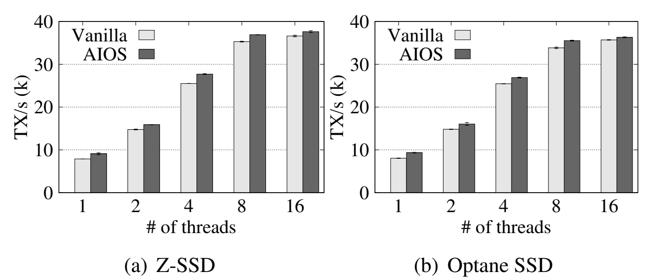
Performance. Figure 15 demonstrates the performance result of DBbench on RocksDB. Specifically, we evaluate the *readrandom* (64 GB dataset, 16-byte key and 1000-byte value) and *fillsync* (16 GB dataset) workloads in DBbench, each representing the random read-dominant case and the random write (and `fdatasync`)-dominant case. Overall, AIOS demonstrates notable speedups on the readrandom workload by 11–32% and the fillsync workload by 22–44%. Recall that AIOS allows duplicated I/Os because of the lazy page cache indexing feature (Section 3.2.3). Duplicated I/Os happen in this experiment. However, the frequency of such events is extremely low (e.g., less than once in 10 million I/Os on the readrandom workload).

CPU usage breakdown. Figure 16 shows the normalized CPU usage breakdown analysis of each workload using eight threads. Overall, AIOS reduces the CPU time spent on I/O wait and kernel. This indicates that AIOS effectively overlaps I/O operation with kernel operations without incurring extra overhead and LBIO effectively reduces the block I/O management overhead. Furthermore, by providing a low random read and write latency, AIOS reduces the overall runtime as well. The trend is similar in both Z-SSD and Optane SSD.



(a) Z-SSD (b) Optane SSD

Figure 17: Filebench-varmail performance with varying the number of threads.



(a) Z-SSD (b) Optane SSD

Figure 18: Sysbench-OLTP-insert performance with varying the number of threads.

4.3.2 Storage Benchmark and OLTP Workload

Figure 17 and Figure 18 show the Filebench-varmail (default configuration) and Sysbench-OLTP-insert (10 GB DB table size) performance with varying the number of threads. On the single thread cases of Filebench-varmail, utilizing AIOS results in 29.9% and 29.4% of improved throughput on Z-SSD and Optane SSD, respectively. Similarly, on the single thread cases of Sysbench-OLTP-insert, AIOS achieves 15.4% and 16.2% performance increase with Z-SSD and Optane SSD, respectively. In general, the use of multi-threading diminishes the benefits of our scheme because of the natural overlap of computation with I/O happens.

5 Related Work

Many prior works aim to alleviate the overheads of the kernel I/O stack, some of which are deployed in commodity OS kernels (e.g., Linux). Multi-context I/O paths can increase the I/O latency due to the overhead of context switching [5, 35]. Today's I/O path design for NVMe SSDs reduces this overhead by eliminating the bottom half of interrupt handling [24]. Using polling instead of interrupts is another solution for removing context switching from the I/O path [5, 47]. Hybrid polling is also proposed to reduce high CPU overheads [9, 21]. Simplified scheduling (e.g., Noop) is effective for reducing I/O latency in flash-based SSDs due to its high-performance random access [46, 51]. Instead of providing I/O scheduling in software, the NVMe protocol supports I/O scheduling on the device side in hardware [14, 24]. Support for differentiated I/O path was introduced to minimize the overhead of I/O path

for high priority tasks [5, 17, 48, 51], which is similar to our LBIO. However, to the best of our knowledge, there is no work applying the asynchronous I/O concept to the storage I/O stack itself.

There are proposals to change the storage interface for I/O latency reduction. Scatter/scatter I/O coalesces multiple I/O requests into a single command, thereby reducing the number of round trips in storage access [37, 42, 50]. DC-express attempts to minimize protocol-level latency overheads by removing doorbells and completion signals [44].

Improving the performance of `fsync` operation is important in many applications as it provides data durability. Nightingale et al. propose to extend the time to preserve data durability from the return of an `fsync` call to the time when the response is sent back to the requester (e.g., remote node) [23]. Using a checksum in the journal commit record can be effective in overlapping journal writes and data block writes [29], albeit checksum collision can become problematic in production systems [41]. OptFS [7] and BFS [45] propose write order-preserving system calls (`osync` and `fbarrier`). With the order-preserving system calls, the overlapping effect in the `fsync` path will be identical. However, when applications need the `fsync` semantic, operations occur synchronously with regard to I/Os.

User-level direct access can eliminate the kernel I/O stack overhead in storage access [6, 16, 28, 49]. The lack of a file system can be augmented by many different approaches from a simple mapping layer [28, 49] to user-level file systems [15, 43, 49]. However, enforcing isolation or protection between multiple users or processes should be carefully addressed [6], and hardware-level support is highly valuable [24]. However, this is not available at the moment.

6 Discussion and Future Work

I/O scheduling. I/O scheduling is necessary for certain computing domains (e.g., cloud computing) [1]. Early versions of the block multi-queue layer provided no I/O scheduling capability [3], but recently, several I/O schedulers have been integrated [8]. Our LBIO eliminates software-level request queues, and thus the current implementation is not compatible with software-level block I/O schedulers. However, the NVMe protocol can support device-side I/O scheduling (e.g., weighted round robin with urgent priority feature [14, 24]), which can augment LBIO. Furthermore, we believe that LBIO can support proper process/thread/cgroup-level I/O scheduling if we relax the static mapping between cores and NVMe queues. We leave this as future work.

File system coverage. Our current implementation is based on the Linux kernel with the Ext4 file system. However, we believe that other journaling file systems (e.g., XFS [39]) or copy-on-write file systems (e.g., Btrfs [31]) may provide similar opportunities for overlapping computation in the I/O path with device access, considering out-of-place updates employed by these file systems.

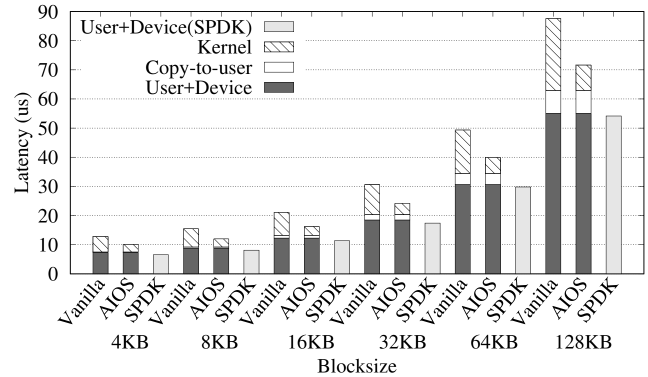


Figure 19: FIO random read latency breakdown in comparison with Intel SPDK on Optane SSD.

Copy-to-user cost. AIOS greatly reduces the I/O stack overhead of the vanilla Linux kernel as shown in Figure 19. However, our proposal does not optimize copy-to-user operations, which remain as a non-negligible source of the overhead, especially when the requested block size is large. Although the in-memory copy is inevitable for buffered reads, we are seeking solutions to take off the memory copy from the critical path so that our proposal can compete with the user-level direct access approach.

7 Conclusion

We propose AIOS, an asynchronous kernel I/O stack customized for ultra-low latency SSDs. Unlike the traditional block layer, the lightweight block I/O (LBIO) layer of AIOS eliminates unnecessary components to minimize the delay in submitting I/O requests. AIOS also replaces synchronous operations in the I/O path with asynchronous ones to overlap computation associated with `read` and `fsync` with device I/O access. As a result, AIOS achieves single-digit microseconds I/O latency on Optane SSD, which was not possible due to high I/O stack overhead. Furthermore, AIOS demonstrates significant latency reduction and performance improvement with both synthetic and real-world workloads on Z-SSD and Optane SSD[2].

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Youjip Won, for their valuable comments. We also thank Prof. Jin-Soo Kim for his devotion of time at LAX and valuable technical feedback. This work was supported partly by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2017R1C1B2007273, NRF-2016M3C4A7952587) and by Samsung Electronics.

[2]The source code is available at https://github.com/skkucsl/aios.

References

[1] AHN, S., LA, K., AND KIM, J. Improving I/O resource sharing of linux cgroup for NVMe SSDs on multi-core systems. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)* (Denver, CO, USA, 2016).

[2] AXBOE, J. FIO: Flexible I/O tester. https://github.com/axboe/fio.

[3] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *International Systems and Storage Conference (SYSTOR '13)* (New York, NY, USA, 2013), pp. 22:1–22:10.

[4] BROWN, N. A block layer introduction part 1: the bio layer, 2017. https://lwn.net/Articles/736534/.

[5] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)* (Atlanta, GA, USA, 2010), pp. 385–395.

[6] CAULFIELD, A. M., MOLLOW, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)* (New York, NY, USA, 2012), pp. 387–400.

[7] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, PA, USA, 2013), pp. 228–243.

[8] CORBET, J. Two new block I/O schedulers for 4.12, 2017. https://lwn.net/Articles/720675/.

[9] EISENMAN, A., GARDNER, D., ABDELRAHMAN, I., AXBOE, J., DONG, S., HAZELWOOD, K., PETERSEN, C., CIDON, A., AND KATTI, S. Reducing DRAM footprint with NVM in facebook. In *European Conference on Computer Systems (EuroSys '18)* (New York, NY, USA, 2018), pp. 42:1–42:13.

[10] FACEBOOK. Rocksdb. https://github.com/facebook/rocksdb/.

[11] HUFFMAN, A. Delivering the full potential of PCIe storage. In *IEEE Hot Chips Symposium* (2013), pp. 1–24.

[12] INTEL. Breakthrough performance for demanding storage workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[13] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *USENIX Annual Technical Conference (USENIX ATC '13)* (San Jose, CA, USA, 2013), pp. 309–320.

[14] JOSHI, K., YADAV, K., AND CHOUDHARY, P. Enabling NVMe WRR support in Linux block layer. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)* (Santa Clara, CA, USA, 2017).

[15] KIM, H.-J., AND KIM, J.-S. A user-space storage I/O framework for NVMe SSDs in mobile smart devices. *IEEE Transactions on Consumer Electronics 63*, 1 (2017), 28–35.

[16] KIM, H.-J., LEE, Y.-S., AND KIM, J.-S. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)* (Denver, CO, USA, 2016).

[17] KIM, S., KIM, H., LEE, J., AND JEONG, J. Enlightening the I/O path: A holistic approach for application performance. In *USENIX Conference on File and Storage Technologies (FAST '17)* (Santa Clara, CA, USA, 2017), pp. 345–358.

[18] KOPYTOV, A. Sysbench: Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[19] KÁRA, J. Ext4 filesystem scaling. https://events.static.linuxfound.org/sites/events/files/slides/ext4-scaling.pdf.

[20] MARKUZE, A., MORRISON, A., AND TSAFRIR, D. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)* (New York, NY, USA, 2016), pp. 249–262.

[21] MOAL, D. L. I/O latency optimization with polling, 2017.

[22] MYSQL AB. MySQL. https://www.mysql.com.

[23] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Symposium on Operating Systems Design and Implementation (OSDI '06)* (Berkeley, CA, USA, 2006), pp. 1–14.

[24] NVM EXPRESS. NVM express base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3c-2018.05.24-Ratified.pdf.

[25] OHSHIMA, S. Scaling flash technology to meet application demands, 2018. Keynote 3 at Flash Memory Summit 2018.

[26] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[27] PETER, S., LI, J., ZHANG, I., PORTS, D. R., ANDERSON, T., KRISHNAMURTHY, A., ZBIKOWSKI, M., AND WOOS, D. Towards high-performance application-level storage management. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)* (Philadelphia, PA, USA, 2014).

[28] PETER, S., LI, J., ZHANG, I., PORTS, D. R., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, CO, USA, 2014), pp. 1–16.

[29] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. In *ACM Symposium on Operating Systems Principles (SOSP '05)* (New York, NY, USA, 2005), pp. 206–220.

[30] REECE, A. DBbench. https://github.com/memsql/dbbench.

[31] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS '13) 9*, 3 (2013), 9:1–9:32.

[32] SAMSUNG. Ultra-low latency with Samsung Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[33] SANDEEN, E. Enterprise filesystems, 2017. http://people.redhat.com/mskinner/rhug/q4.2017/Sandeen_Talk_2017.pdf.

[34] SAXENA, M., AND SWIFT, M. M. FlashVM: Virtual memory management on flash. In *USENIX Annual Technical Conference (USENIX ATC '10)* (Berkeley, CA, USA, 2010), pp. 14–14.

[35] SHIN, W., CHEN, Q., OH, M., EOM, H., AND YEOM, H. Y. OS I/O path optimizations for flash solid-state drives. In *USENIX Annual Technical Conference (USENIX ATC '14)* (Philadelphia, PA, USA, 2014), pp. 483–488.

[36] SILVERS, C. UBC: An efficient unified I/O and memory caching subsystem for NetBSD. In *USENIX Annual Technical Conference (USENIX ATC '00)* (San Diego, CA, USA, 2000), pp. 285–290.

[37] SON, Y., HAN, H., AND YEOM, H. Y. Optimizing file systems for fast storage devices. In *ACM International Systems and Storage Conference (SYSTOR '15)* (New York, NY, USA, 2015), pp. 8:1–8:6.

[38] SWANSON, S., AND CAULFIELD, A. M. Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage. *Computer 46*, 8 (August 2013), 52–59.

[39] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In *USENIX Annual Technical Conference (USENIX ATC '96)* (San Diego, CA, USA, 1996), vol. 15.

[40] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine 41*, 1 (2016), 6–12.

[41] TS'O, T. What to do when the journal checksum is incorrect, 2008. https://lwn.net/Articles/284038/.

[42] VASUDEVAN, V., KAMINSKY, M., AND ANDERSEN, D. G. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In *ACM Symposium on Cloud Computing (SoCC '12)* (New York, NY, USA, 2012), pp. 8:1–8:13.

[43] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *European Conference on Computer Systems (EuroSys '14)* (New York, NY, USA, 2014), pp. 14:1–14:14.

[44] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., MOAL, D. L., BUNKER, T., XU, J., SWANSON, S., AND BANDIĆ, Z. DC express: Shortest latency protocol for reading phase change memory over PCI express. In *USENIX Conference on File and Storage Technologies (FAST '14)* (Santa Clara, CA, USA, 2014), pp. 309–315.

[45] WON, Y., JUNG, J., CHOI, G., OH, J., SON, S., HWANG, J., AND CHO, S. Barrier-enabled IO stack for flash storage. In *USENIX Conference on File and Storage Technologies (FAST '18)* (Oakland, CA, USA, 2018), pp. 211–226.

[46] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance analysis of NVMe SSDs and their implication on real world databases. In *ACM International Systems and Storage Conference (SYSTOR '15)* (New York, NY, USA, 2015), pp. 6:1–6:11.

[47] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. In *USENIX conference on File and Storage Technologies (FAST '12)* (San Jose, CA, USA, 2012), pp. 3–3.

[48] YANG, T., LIU, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Redline: First class support for interactivity in commodity operating systems. In *USENIX Conference on Operating Systems Design and Implementation (OSDI '08)* (Berkeley, CA, USA, 2008), pp. 73–86.

[49] YANG, Z., HARRIS, J. R., WALKER, B., VERKAMP, D., LIU, C., CHANG, C., CAO, G., STERN, J., VERMA, V., AND PAUL, L. E. SPDK: A development kit to build high performance storage applications. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom '17)* (Hong Kong, 2017), pp. 154–161.

[50] YU, Y. J., SHIN, D. I., SHIN, W., YOUNG SONG, N., CHOI, J. W., KIM, H. S., EOM, H., AND EOM, H. Y. Optimizing the block I/O subsystem for fast storage devices. *ACM Transactions on Compueter Systems (TOCS '14) 32* (2014).

[51] ZHANG, J., KWON, M., GOUK, D., KOH, S., LEE, C., ALIAN, M., CHUN, M., KANDEMIR, M. T., KIM, N. S., KIM, J., AND JUNG, M. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *Symposium on Operating Systems Design and Implementation (OSDI '18)* (Carlsbad, CA, USA, 2018), pp. 477–492.