Towards Low-Latency I/O Services for Mixed Workloads Using Ultra-Low Latency SSDs

Mingzhe Liu, Haikun Liu, Chencheng Ye, Xiaofei Liao, Hai Jin, Yu Zhang, Ran Zheng, Liting Hu†
National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computing Science and Technology, Huazhong University of Science and Technology, Wuhan, China
†Department of Computer Science, Virginia Tech, USA
{mzliu, hkliu, yecc, xfliao, hjin, zhyu, zhraner}@hust.edu.cn, litinghu@cs.vt.edu

ABSTRACT

Low-latency I/O services are essential for latency-sensitive workloads when they co-run with throughput-oriented workloads in cloud data centers. Although advanced SSDs such as Intel Optane SSDs can offer ultra-low latency at the device layer, I/O interference among various workloads through the I/O stack can still significantly enlarge I/O latency. It is still an open problem to best utilize ultra-low latency SSDs in cloud computing environments.

In this paper, we analyze the entire I/O stack and reveal that I/O interference is mainly attributed to resource contention in the SSD device, transactions commit in the file system, and costly process scheduling. To address these problems, we propose FastResponse, a holistic approach to use ultra-low latency SSDs for latency-sensitive workloads. First, we propose a new I/O scheduler at the block layer to throttle I/O requests of throughput-oriented workloads, and thus reduce the resource contention in the SSD device. Second, we develop a fine-grained journaling scheme to reduce the latency of transaction at the file system layer. Third, we redesign *Completely Fair Scheduler* (CFS) to promote the priority of latency-sensitive processes. We implement FastResponse in Linux kernel and evaluate it with several mixed workloads. Compared with the vanilla Linux and the state-of-the-art SelectISR, FastResponse can reduce the average response time of latency-sensitive workloads by 18-70% and 10-67%, respectively, and reduce the 99.9th percentile response time by 58-80% and 52-78%, respectively. Meanwhile, the performance degradation for throughput-oriented workloads is less than 6%.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Software and its engineering** → **Operating systems**.

KEYWORDS

I/O Interference, I/O Scheduling, Ultra Low-Latency SSD, Storage System

Corresponding Author: Haikun Liu.

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06…$15.00
https://doi.org/10.1145/3524059.3532378

ACM Reference Format:
Mingzhe Liu, Haikun Liu, Chencheng Ye, Xiaofei Liao, Hai Jin, Yu Zhang, Ran Zheng, Liting Hu. 2022. Towards Low-Latency I/O Services for Mixed Workloads Using Ultra-Low Latency SSDs. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524059.3532378

1 INTRODUCTION

Datacenters have become the first option for *Internet Service Providers* (ISPs) to deploy their workloads such as social media, search engines, online shopping, and advertising [1, 2]. To improve resource utilization and energy efficiency, a popular trend is to co-locate different kinds of workloads together in datacenter servers [3, 4]. However, the co-location interference between throughput-oriented workloads (e.g., MapReduce workloads) and latency-sensitive workloads (e.g., database query services) usually causes significant performance degradation for latency-sensitive workloads [5–7].

To meet the *Service Level Agreement* (SLA), fast storage devices, such as *ultra-low latency* (ULL) SSDs [8–10], are used to provide low-latency cloud services. ULL SSDs can offer 10μs access latency which is 10 times lower than that of traditional SSDs. However, we observe that even using ULL SSDs, the response time of latency-sensitive workloads is still high in case of severe I/O contention. The tail latency of latency-sensitive workloads can significantly increase by dozens of times when they co-run with throughput-oriented workloads (Section 2.2). To alleviate the interference between throughput-oriented workloads and latency-sensitive workloads in the I/O stack, a lot of recent efforts have been made [11–16] to prioritize I/O requests of latency-sensitive workloads. The I/O priority is typically enforced at the block-level scheduler or other layers, such as page cache and the device driver.

However, these approaches have some limitations when applying them to ULL SSDs due to the following reasons. 1) *Most previous approaches do not consider the cross-layer interference in the entire I/O stack.* For example, many I/O schedulers such as K2 [16] and D2FQ [17] dispatch requests with high priority (weight) first, but only work at the block layer. I/O interference in other storage layers can still enlarge the latency of mission-critical workloads. 2) *Most previous approaches are designed for traditional flash SSDs, and fail to exploit the promising feature of ULL SSDs.* For example, Intel Optane SSD leverages 3D-XPoint technology. It can achieve very high bandwidth utilization even with a small queue depth while still offering low latency. Previous studies [11, 12, 15, 18] cannot fully utilize these features to provide low-latency I/O services. 3) *Previous approaches usually ignore the impact of process scheduling on the I/O*

latency [13–15]. We find that the cost of process scheduling can be as high as 22μs when latency-sensitive workloads and throughput-oriented workloads co-run together (Section 2.2). Since the access latency of ULL SSDs is as low as 10μs, the cost of process scheduling is too large relative to ULL SSDs.

In this paper, we first identify three root causes of I/O interference in the entire I/O stack. Specifically, in the file system layer, a compound transaction committed by JBD2 usually contains uncorrelated file updates, and thus enlarges the latency of synchronous writes. In the block device layer, burst I/O requests can significantly increase the queue depth of ULL SSDs, and thus enlarge the I/O latency. Moreover, the CFS scheduler usually takes a long time to reschedule the latency-sensitive process after it is awakened. To shorten the latency of mission-critical I/O requests, we propose FastResponse, a holistic and cross-stack approach for ULL SSDs. We offer user-level APIs to characterize the attribute of co-running I/O workloads, and coordinate different I/O layers together to mitigate the I/O interference. FastResponse can greatly reduce the response time of latency-sensitive workloads while having trivial impact on the performance of throughput-oriented workloads.

We make the following contributions in this paper:

- We identify the root causes of I/O interference on ULL SSDs for co-running workloads, including compound transaction commit in the file system layer, resource contention in the block device layer, and costly process scheduling (Section 2).
- In the block layer, we split large I/O requests into small ones to mitigate their impact on latency-sensitive I/Os. We also throttle I/O requests of throughput-oriented workloads moderately to mitigate the interference on latency-sensitive workloads (Section 3.3).
- In the file system layer, we develop a lightweight journaling scheme for latency-sensitive workloads particularly, and commit file-level transactions for latency-sensitive workloads individually to mitigate the I/O interference. Since transactions committed by latency-sensitive processes only contain file metadata, the latency of compound transactions can be significantly reduced (Section 3.4).
- We redesign *Complete Fair Scheduler* (CFS) to fully exploit the feature of ULL SSDs. The new process scheduler promotes the priority of latency-sensitive processes to minimize the waiting time after critical processes are woken up (Section 3.5).

We implement FastResponse based on Linux kernel 5.3.7, and the source code is available at Github [19]. We evaluate FastResponse with several I/O benchmarks. Compared with the vanilla Linux, FastResponse can significantly reduce the average and 99.9th response time of latency-sensitive workloads by 18-70% and 58-80%, respectively. Compared with the state-of-the-art SelectISR [11], FastResponse can reduce the average and 99.9th percentile response time by 10-67% and 52-78%, respectively, with less than 6% performance degradation for throughput-oriented workloads.

The remainder of this paper is organized as follows. Section 2 discusses the background and motivation. Section 3 describes the design and implementation of FastResponse. Sections 4 presents the experimental setup and performance evaluation. Section 5 discusses the related work. We conclude in Section 6.

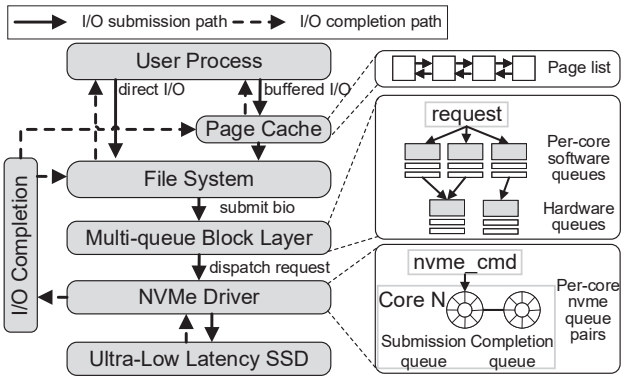


Figure 1: An overview of I/O stack

2 BACKGROUND AND MOTIVATION

2.1 Linux I/O Stack

Fig. 1 illustrates the I/O stack of a typical Linux system. It consists of multiple layers: page cache, file system, multi-queue block layer, and NVMe driver. In the following, we elaborate the read and write paths across the I/O stack.

Read Path. Fig. 2 illustrates the operations in the read path. The process enters the kernel space via a *read* system call. (1) The data is first retrieved from the page cache. Upon a cache miss, free pages are allocated to buffer I/O requests. (2) The file system inserts pages into the page cache, retrieves the *logical block address* (LBA), and finally sends I/O requests to the underlying layer through a `bio` structure. (3) The multi-queue block layer (blk-mq) performs I/O scheduling and dispatch. It converts the `bio` structure into a `request` structure and puts it into per-core software queues mapped to hardware queues. (4) The NVMe driver converts the `request` structure into a `nvme_command` structure and puts it into the *submission queue* (SQ). Then, the I/O process yields the CPU and waits for the I/O completion. When the I/O operation in the device is completed, the device controller writes an entry in the *completion queue* (CQ) and sends a *message signaled interrupt* (MSI) to a CPU core. (5) The interrupted core executes an *Interrupt Service Routine* (ISR) to handle the I/O completion. In this stage, the blocked I/O process is woken up. However, only when the process scheduler reschedules it, it can obtain CPU time to complete the following-up work such as copying pages to the user. Therefore, the process scheduling is on the critical path of an I/O operation. At last, the read system call returns.

Write Path. The *write* system call usually writes data from a buffer to the page cache and returns immediately. However, when applications perform synchronous writes (i.e., `write()`+ `fsync()`), the kernel should perform several I/O operations to write back dirty blocks and metadata to guarantee data consistency.

Fig. 3 shows operations of the `fsync` in the ext4 file system. First, the application thread writes dirty blocks of the target file ('D1') and waits for the I/O completion. Second, the kernel wakes up the *Journaling Block Device* (JBD2) thread in ext4 file system to commit a compound transaction which contains updates of multiple files. The

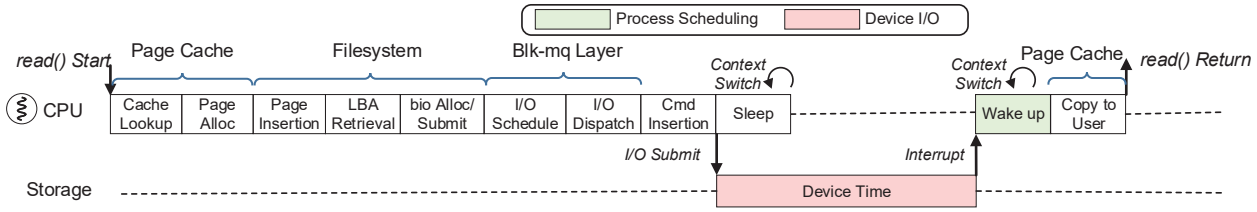


Figure 2: The read operation involves several layers and one process scheduling on the critical path.

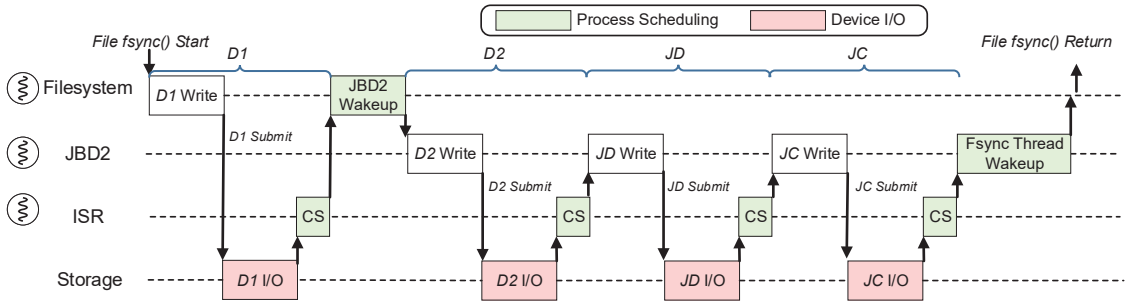
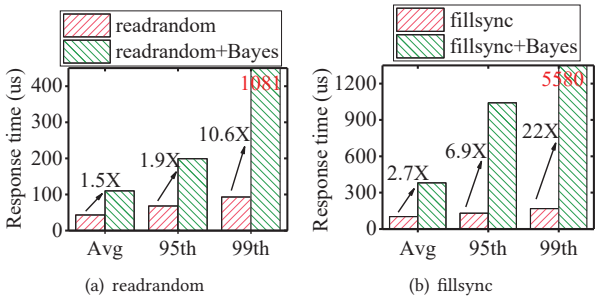
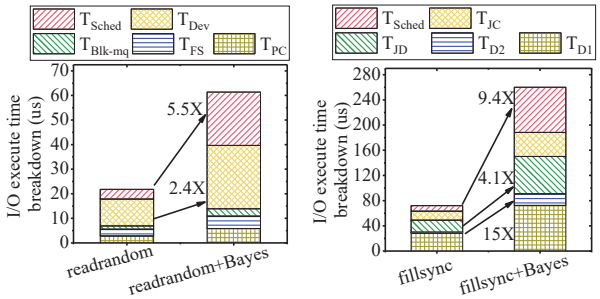


Figure 3: The operations in the `fsync` path, in which 'D1', 'D2', 'JD' and 'JC' represent dirty data blocks of the target file, dirty data blocks of other files involved in the compound transaction, journal blocks, and the journal commit block, respectively. 'CS' denotes context switch. One `fsync` leads to four device I/O operations and six context switches.



(a) readrandom (b) fillsync

Figure 4: I/O interference due to co-running latency-sensitive database workloads (RocksDB) and a throughput-oriented workload (Bayes) on ULL SSD.



(a) readrandom (b) fillsync

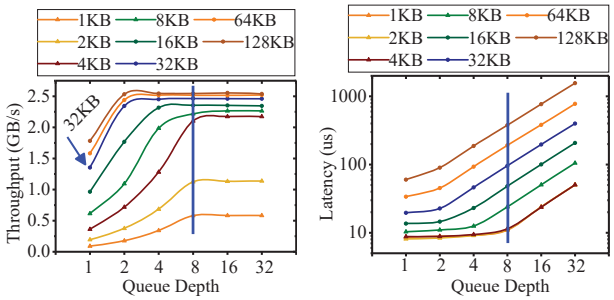
Figure 5: The breakdown of the average latency for *readrandom* and *fillsync*

JBD2 thread should write back all dirty blocks of other files involved ('D2') in the transaction. Third, after data blocks are written to the device, the JBD2 thread writes back metadata blocks ('JD'), such as the inode bitmap, external extent blocks. Finally, it writes a commit block ('JC') to guarantee the integrity of a transaction. Overall, one `fsync` contains four device I/O operations and six process scheduling in the entire I/O path.

2.2 Motivation

We co-run latency-sensitive workloads (*readrandom* or *fillsync* in RocksDB [20]) with *Bayes* [21] (a throughput-oriented workload) on a server equipped with the ULL SSD. The *fillsync* benchmark writes a set of values with random order using the `sync` mode. Bayes performs machine learning (20 GB dataset) on Hadoop framework. Detailed experimental setup can be found in Section 4. As shown in Figure 4, the average latencies of the *readrandom* and *fillsync* in the co-running model increase by 1.5× and 2.7× compared with the stand-alone execution model, respectively. The 99.9th percentile latency even increases by dozens of times. To explore the root causes, we profile the execute time spent in the I/O stack for the `read` and `fsync` paths, respectively.

Fig. 5(a) shows the time breakdown for the `read` path. T_{PC}, T_{FS}, T_{Blk}, and T_{Dev} represent the time spent in the page cache layer, the file system layer, the blk-mq layer, and the device layer, respectively. T_{Sched} represents the waiting time due to process scheduling. The execution time in each portion increases significantly. Particularly, the dominant portions T_{Dev} and T_{Sched} increase by 2.4× and 5.5×, respectively.



(a) Average request throughput (b) Average request latency

Figure 6: The performance characteristics of ULL SSD.

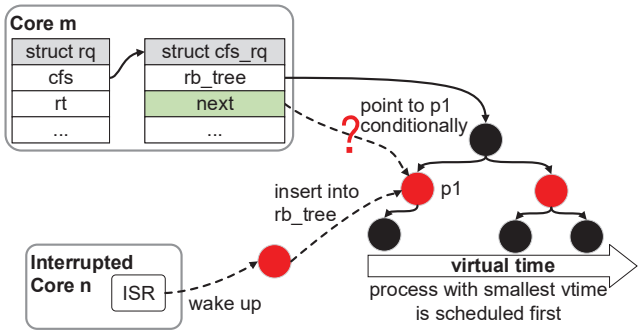


Figure 7: The scheduling of the awakened I/O process

Fig. 5(b) shows the time breakdown of the `fsync` path. T_{D1}, T_{D2}, T_{JD}, and T_{JC} represent the time spent in writing data blocks of the target file, data blocks of other uncorrelated files, journal blocks, and the commit block, respectively. T_{Sched} represents the waiting time for process scheduling after waking up JBD2 thread and `fsync` thread. T_{D2} and T_{Sched} are trivial when *fillsync* runs alone. However, when it co-runs with *Bayes*, T_{D2} and T_{Sched} increase by 15× and 9.4×, respectively. The latency of other portions also increases by 2-3×.

According to the above observations, we identify three root causes for the co-location interference as follows.

Root Cause 1: The block layer is unable to limit the arrival rate of burst I/O requests to ULL SSDs. At first, we explore the reason of significant latency increased in the device for the read path. We measure the average latency and throughput of Optane SSD using FIO benchmark [22] for the random read pattern. The I/O queue depth (the number of requests submitted but not yet completed) ranges from 1 to 32, and the I/O size ranges from 1 KB to 128 KB.

As shown in Fig.6, for all request sizes, once the queue depth exceeds a certain value (8), the throughput does not increase any more, but the latency increases significantly. Also, the Optane SSD approaches its maximum bandwidth when the request size increases to 32 KB. Moreover, the request latency increases rapidly with the growth of request size. We observe that the performance of the Optane SSD for the random write pattern is similar to the random read pattern. Therefore, *it is essential to use moderate-sized I/O requests (no more than 32KB) and maintain a relatively low queue depth (no more than 8) to reduce the request latency while still best utilizing the maximum bandwidth of ULL SSDs.*

The queue depth of ULL SSDs usually increases significantly because most throughput-oriented applications generate burst I/Os. In our experiment, the queue depth even exceeds 160, and thus significantly enlarges the request latency. Moreover, some filesystems such as HDFS usually use large blocks (e.g., 128 KB), which also exacerbates the I/O congestion in ULL SSDs. Unfortunately, the block layer is unable to limit the arrival rate of burst I/O requests to the device. The contention in the device often results in long latency of I/O requests.

Root Cause 2: A compound transaction committed by JBD2 usually involves uncorrelated file updates, and thus increases the latency of synchronous writes. File systems also have a significant impact on the latency of I/O requests. Here, we explore the reason for the increased latency in the `fsync` path. Ext4 file system guarantees data consistency by keeping all updates in a large compound transaction. Thus, the file system has to spend more time in waiting for the transaction committing due to the following two reasons: 1) The transaction involves dirty data blocks of other uncorrelated files, and thus introduces T_{D2}. 2) JBD2 has to submit more metadata blocks which also lead to a growth of T_{JD}. Particularly, the latency due to the journaling file system increases significantly when the block device is busy.

Root Cause 3: CFS is unable to schedule latency-sensitive processes immediately even if they are awakened. We now explore the reason about the increased latency in process scheduling. Fig. 7 illustrates the scheduling of I/O process. CFS schedules processes according to the virtual time, i.e. `vtime`. Generally, a process with a smaller `vtime` is scheduled first. When a suspended process (e.g., `p1`) is woken up, `p1` is inserted into the current run queue of a CPU core, but it still has to wait till it is rescheduled. To schedule `p1` immediately, CFS maintains a pointer `next` to point to a process that should be scheduled first in the next scheduling period.

The scheduling of `p1` may be postponed when `p1` co-runs with *Bayes* for two reasons. (1) The scheduler cannot schedule `p1` immediately because it may wait a long time to begin the next scheduling period. Without a preemption mechanism, *Bayes* may run for a long time, and the CPU has no opportunity to schedule `p1` until an interrupt is triggered. (2) The pointer `next` can point to `p1` only when the difference of `vtime` between the current running process and `p1` is larger than a given threshold (4 milliseconds on our server). We note that the `next` pointer still does not work for ULL SSDs even using a smaller threshold (e.g., 1 ms), since the threshold is still two orders of magnitude higher than the access latency of ULL SSDs. On the other hand, a shorter scheduling interval may cause frequent context switching and waste CPU time.

3 DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of FastResponse, including the workload characterization, the block layer enhancement, a file-level journaling scheme, and the extension of CFS process scheduler.

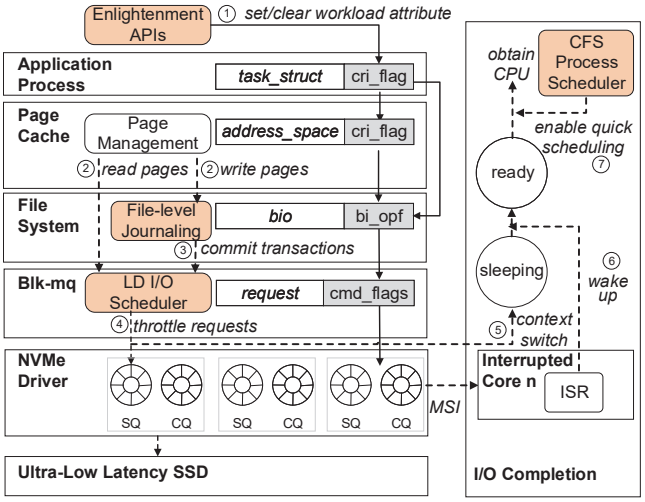


Figure 8: An overview of FastResponse, with four key modules colored

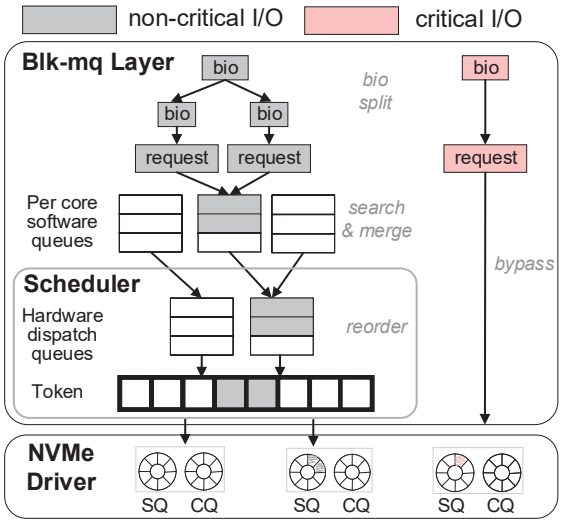


Figure 9: An overview of blk-mq enhancement, the optimizations include (1) splitting the large and non-critical I/Os, (2) bypassing I/O queues for critical requests, and (3) throttling non-critical I/O requests.

3.1 Overview

In order to solve the above problems, our goal is to mitigate I/O interference in the file system layer, the block layer, and process scheduling. There remain three key technical challenges: (1) how to deliver the attribute of latency-sensitive I/Os through the I/O stack, (2) how to mitigate interference in different layers of the entire I/O path, and (3) how to mitigate the negative impact on throughput-oriented workloads.

With these challenges in mind, we propose FastResponse, a holistic approach for improving the performance of latency-sensitive workloads using ULL SSDs. FastResponse is composed of four key modules, as shown in Fig. 8. The attribute of I/Os is defined by user APIs, and the OS kernel passes the tagged attribute through the entire I/O stack (①). When a `read` system call misses in the page cache, the file system sends the `bio` structure to underlying layers after pages are allocated(②). When an `fsync` system call is invoked, the file system commits file-level transactions for latency-sensitive processes to guarantee data consistency (③). When the blk-mq receives the upper-layer `bio`, our I/O scheduler throttles the requests of throughput-oriented workloads (④). Then, the current I/O process yields CPU to another process(⑤). Once the I/O request is completed, ISR wakes up the previous process (⑥). CFS gives a high priority to the critical I/O process so that it can be scheduled immediately (⑦).

3.2 Workload Characteristic Awareness

At first, we should identify whether a I/O request is latency-sensitive (also called critical) or not in the storage stack. It is difficult for the kernel to predict the criticality of I/Os by their patterns or history behaviors [13]. Instead, we provide APIs for users to characterize the attribute of latency-sensitive I/Os. The attribute is then delivered through the entire I/O stack. As shown in Fig. 8, we extend the data structures in different I/O layers to pass the I/O attribute. A new entry is added in `task_struct` and `address_space` to identify the I/O attribute. We also extend the `bi_opf` (in `bio`) and `cmd_flags` (in `request`) to store the attribute. Note that the attribute should be passed across the entire I/O stack because other I/O layers may fetch invalid `task_struct` reference [11].

The API can be used flexibly to set the attribute of I/O requests. If programmers are aware of the characteristics of applications, they can modify the applications to define the attribute of latency-sensitive tasks of a code segment or the entire lifetime of a thread. On the other hand, since client users may selfishly configure their applications with the high priority, the API can be only provided for system administrators to set the attribute of I/O requests at the server side.

3.3 Blk-mq Enhancement

Main idea. In the block layer, our goal is to control the rate of dispatching large I/O requests to the underlying device layer, and thus mitigate the impact of I/O contention on the latency of critical requests. More specifically, we design a lightweight I/O scheduler to split large non-critical I/O requests and throttle them effectively.

I/O scheduler. To limit the arrival rate of large I/O requests to ULL SSDs while still fully utilizing their high bandwidth, we design a new I/O scheduler called *Limited Depth* (LD), as shown in Fig. 9. We use tokens to throttle the number of non-critical requests to ULL SSDs moderately. Each non-critical request is dispatched with a token which is released during the I/O completion. Thus, non-critical requests are blocked if there is no available token. In this way, the impact of I/O interference on latency-sensitive requests in the device can be mitigated. Moreover, unlike BFQ [23] and MQ-DD [24] that use a global queue to dispatch all requests, LD attaches a dispatch queue to each hardware queue, and thus can fully utilize the high bandwidth of ULL SSDs through parallel I/O dispatching. On the other hand, we bypass these I/O queues for latency-sensitive

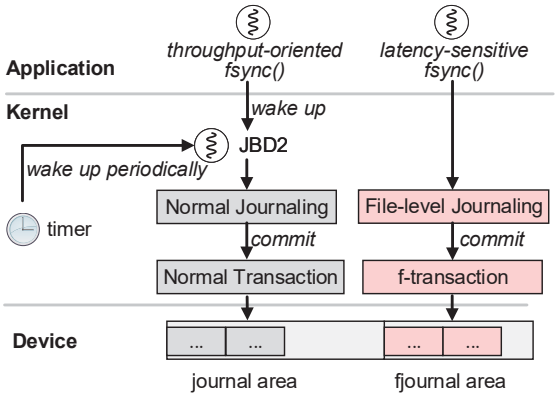


Figure 10: Transactions commit in the file system, we provide file-level journaling for latency-sensitive processes.

I/Os because I/O scheduling often has a negative impact on the latency [25, 26] and these I/Os should not be throttled.

Splitting large non-critical requests. Because large non-critical requests also lead to a high degree of resource contention on ULL SSDs, we split a large non-critical `bio` into multiple small `bio`, and thus offer more opportunities to serve critical requests. We find that the optimal block size is 32 KB because this configuration can achieve almost the maximum bandwidth of ULL SSDs. However, if the split sub-requests are served out of order, the parent request has to wait for the last sub-request to complete, and the latency of the parent large request would be very high. Also, the throughput of non-critical applications declines. To this end, our I/O scheduler inserts the split sub-requests into the queue in order according to their LBAs so that they can be dispatched consecutively.

3.4 File-level Journaling

Main idea. Fig. 10 shows different transactions commit in the file system. To shorten the latency of critical I/O processes, we design a file-level journaling called `fjournaling` particularly. Since `fjournaling` commits file-level transactions (f-transactions) that only include minimum metadata of the fsynced file, the latency of f-transaction can be minimized. To simplify the implementation of `fjournaling` while still guaranteeing crash consistency, `fjournaling` is implemented with no dependency on the normal journaling (i.e., JBD2). Thus, the metadata blocks committed by `fjournaling` would be committed again by JBD2 later.

`fjournaling` reduces the latency of `fsync()` in three ways. (1) It does not have to wait for writing back data blocks of uncorrelated files involved in the compound transaction, and thus T_{D2} can be completely eliminated. (2) It reduces the amount of metadata blocks submitted, and thus T_{JD} is reduced. (3) The `fsync` process itself is responsible for committing transactions, and thus avoids the cost of process scheduling for JBD2 thread and `fsync` thread wake-up, i.e., T_{Sched} can be completely eliminated.

Fjournaling. `fjournaling` only commits minimum metadata including the inode entry and external extent blocks that are necessary for crash recovery. Other metadata blocks shared between

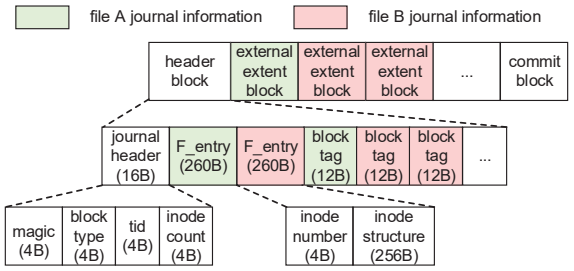


Figure 11: The data structure of the f-transaction, file A and B are updated by latency-sensitive processes.

multiple files, such as block bitmap, inode table, and *global descriptor table* (GDT) are not recorded because they can be recovered based on f-transactions and the latest normal transaction committed by JBD2. As shown in Fig. 10, we commit f-transactions in a new area, which is separated from the normal journal area. Since blocks of a transaction in the disk must be continuous, we separate the journal area of the `fjournaling` and the normal journaling so that normal transactions and f-transactions can be committed in parallel.

File-level transaction. Fig. 11 shows the data structure of an f-transaction. Like the normal transaction, an f-transaction is composed of a header block, several metadata blocks (i.e., external extent blocks), and a commit block. The header block is composed of a journal header, several F_entry, and several block tags which store the mapping between each metadata block and its block number in the file system. Note that only modified external extent blocks are recorded for each uncommitted file. The commit block is used to validate the completion of an f-transaction.

Crash recovery. Because dirty data blocks are written to the device before a transaction is committed, we only need to recover the committed transactions upon a system crash. Here, we only describe the recovery of f-transactions. First, the crash recovery module scans the normal journal area, redoes the valid normal transactions that have been committed in the log but not yet checkpointed, and finds the valid transaction with the largest ID (`Max_ID`). Then, the `fjournal` area is scanned. If the ID of an f-transaction is greater than the `Max_ID`, this f-transaction is valid. We continue to find out all valid f-transactions. Finally, the crash recovery module redoes all valid f-transactions in order.

For each valid f-transaction, the crash recovery module can recover the file blocks being written and the block bitmap using the inode structure, the external extent blocks, and the block tags. The inode number and the inode structure are used to recover file system level metadata, i.e. inode table, inode bitmap, and GDT. After all valid f-transactions are redone, the normal journal area and the `fjournal` area are reclaimed, and the file system can perform other operations.

3.5 Latency Sensitivity Aware Process Scheduling

Main idea. Our goal is to schedule critical I/O processes immediately after they are awakened. Since the process pointed by the

Algorithm 1 Wake up the critical I/O process

```
1: if next is NULL or points to a non-critical process then
2:   if the awakened_process is critical then
3:     next of CFS_RQ points to the awakened_process;
4:     mark CFS_RQ as need_rescheduled;
5:   end if
6: else
7:   enter normal wake-up flow;
8: end if
```

pointer `next` or with minimum `vtime` is scheduled first, this process can be scheduled earlier in the next scheduling period. To schedule the suspended critical I/O process as soon as possible, we should (1) *point* `next` *to the suspended I/O process*, and (2) *begin the next scheduling period as soon as possible*. We modify CFS based on these two principles.

Process scheduling policy. In the traditional CFS algorithm, the `next` points to an awakened process (`p1`) only when the difference of `vtime` between the current running process and `p1` is larger than a given threshold (4 ms on our server). Although a larger threshold can avoid frequent context switching, it impedes the adoption of ULL SSDs because the access latency of these devices can be as low as tens of microseconds. To address this problem, we redesign CFS to adapt to ULL SSDs. In our process scheduler, kernel first wakes up the I/O process that is suspended on a certain CPU, and then checks whether it can preempt the current running process on that CPU. As shown in Algorithm 1, if the *awakened_process* is critical, (1) the `next` of the CFS's run-queue (CFS_RQ) points to it immediately, oblivious to the time threshold for setting the `next`; (2) and marks the CFS_RQ as *need_rescheduled* so that the CPU can quickly begin the next scheduling period. In this way, the process pointed by `next` can be scheduled as quickly as possible.

Before the process pointed by `next` is scheduled, it may still be replaced by other processes. This would postpone the scheduling of the awakened critical process. Therefore, once `next` points to a critical process, we do not change the `next` until the awakened critical process is rescheduled.

Our process scheduling scheme has very little negative impact on non-critical processes. It is not possible to starve non-critical processes due to the following reasons. First, the process pointed by `next` is scheduled only if the difference between its `vtime` and the minimum `vtime` is less than a given threshold. Second, the critical process is rescheduled according to our policies only when it is woken up. These two constraints still offer massive opportunities to schedule non-critical processes.

4 EVALUATION

4.1 Methodology

System configurations. We evaluate the proposed techniques on an Inspur server. It is equipped with two octa-core 2.40 GHz Intel Xeon CPU E5-2630 v3 processor, 32GB 2666 MHz DDR4 memory, and a 480GB Intel Optane 900P SSD. The proposed techniques are integrated into a vanilla Linux kernel 5.3.7. By default, the operating system adopts ext4 file system with data journaling mode set as ordered.

Table 1: Workloads characteristics

Workload	Workload Type	R:W Ratio	Avg Arrival Interval (ms)	
			Reads	*Writes*
Throughput-oriented Workloads				
Terasort	Offline analytics	30:70	0.67	0.29
Bayes	Machine learning	57:43	1.11	1.47
Kmeans	Machine learning	33:67	1.99	0.97
PageRank	Websearch	70:30	1.31	3.14
Index	Websearch	20:80	0.78	0.19
Latency-sensitive Workloads				
RocksDB	readrandom	99:1	0.4	-
	fillsync	1:99	-	0.5

Compared systems. To elaborate our designs in different I/O layers, we evaluate the following schemes:

1. **Vanilla:** A vanilla Linux kernel with default configurations is used as a baseline;
2. **SelectISR:** FLASHSHARE [11] is so far the most relevant work to FastResponse. FLASHSHARE proposes cross-stack optimizations to reduce I/O interference through the software layer to the device layer. Because our work focuses on software approaches for public available ULL SSD devices, we only implement the software solution–SelectISR proposed in FLASHSHARE for a fair comparison. The major techniques include bypassing the block layer and polling I/O completions for critical I/Os.
3. **B-Opt:** Compared with `Vanilla`, we optimize the block layer and propose LD I/O scheduler to split large non-critical requests and to throttle non-critical requests;
4. **BS-Opt:** Compared with `B-Opt`, we further optimize CFS to shorten the waiting time for scheduling latency-sensitive processes;
5. **BSJ-Opt:** Based on `BS-Opt`, we propose `fjournaling` to optimize synchronous writes for latency-sensitive applications. This is the full design of FastResponse.

Applications and workloads. We evaluate the proposed techniques by co-running a latency-sensitive application and a throughput-oriented application.

For latency-sensitive application, we use RocksDB, an embedded, persistent key-value store underpinning a lot of low-latency services [27, 28]. It stores 40 GB dataset with regular-sized 4-byte keys and 4KB values. We use a widely-used workload generator–DBbench [29] to generate two workloads with diverse I/O characteristics, *readrandom* and *fillsync*. *Readrandom* is read-dominant. It reads random key-value pairs in the key-value store. Occasionally, it updates random key-value pairs using the `async` mode. *Fillsync* is write-dominant. It frequently updates random key-value pairs using the `sync` mode. To emulate a real-world scenario as suggested by a previous work [30], the workloads send two thousands requests to RocksDB per second.

For throughput-oriented applications, we adopt several applications selected from HiBench [21] and BigDatabench [31]. They exhibit various read-write ratios and request arrival intervals. We deploy MapReduce and HDFS on a single server in pseudo-distributed mode. The datasets of the applications all exceed 10 GB. The size of the read and write requests are all around 128 KB. Table 1 summarizes the characteristics of these workloads.

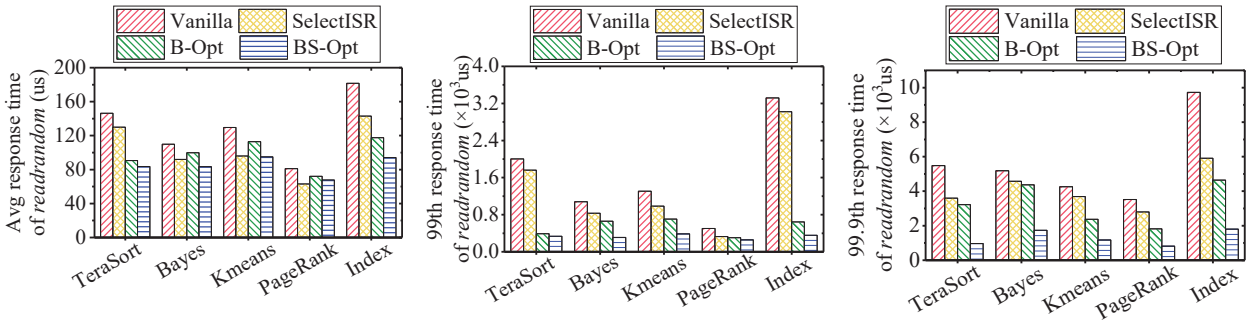


Figure 12: The response time of *readrandom* in RocksDB when it co-runs with various throughput-oriented workloads

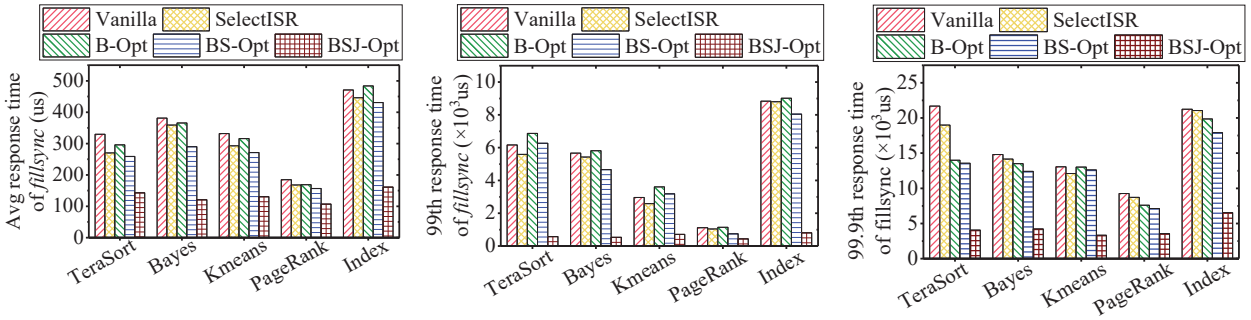


Figure 13: The response time of *fillsync* in RocksDB when it co-runs with various throughput-oriented workloads

4.2 Latency

In the following, we analyze the performance of read and write paths using workloads *readrandom* and *fillsync*, respectively.

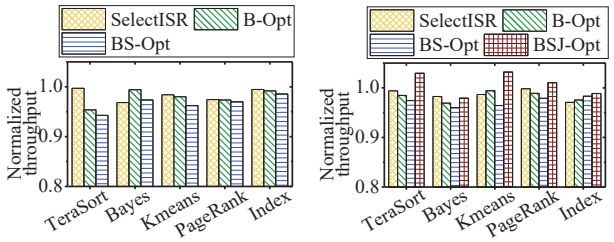
Read path. Fig. 12 shows the average, 99th, and 99.9th percentile response time (latency) of serving *readrandom* requests in RocksDB when it co-runs with different throughput-oriented workloads. Because the read path does not involve the optimization on journaling, we do not evaluate the `BSJ-Opt` in this case. Compared to Vanilla, `BS-Opt` substantially reduces the average response time by 36%, and the 99.9th percentile response time by 77% on average. In contrast, `SelectISR` can only reduce the average, the 99th, and the 99.9th percentile response times by 21%, 19%, and 28% on average, respectively. Even the partial solution `B-Opt` shows much lower latency compared with `SelectISR`.

`Vanilla` suffers from severe I/O interference for co-running applications, and thus represents the longest response time. `SelectISR` polls critical I/O requests to avoid the cost of context-switching introduced by interrupts (about 2μs). More importantly, a critical I/O process does not yield CPU time to other processes, and thus does not suffer from long waiting time for the process scheduling. However, since `SelectISR` is unable to reduce the latency in the device layer, it still achieves less performance optimization relative to `B-Opt` and `BS-Opt`, especially for the tail latency. Since `B-Opt` splits large non-critical requests and throttles non-critical requests to avoid resource contention in the device, it greatly reduces the 99th response time by 49% compared with `SelectISR`. The benefit is more significant when the latency-sensitive workload *readrandom* co-runs with throughput-oriented workloads, such as *Terasort* and *Index*.

Compared with `B-Opt`, `BS-Opt` further reduces the average, 99th, and 99.9th response time by 16%, 40%, and 59%, respectively. When *readrandom* co-runs with CPU-intensive workloads, such as *Bayes* and *Kmeans*, `BS-Opt` can even reduce the 99th latency by 56% by using the preemptive scheduling technique. These experimental results demonstrate the effectiveness of our optimizations on mitigating the co-location interference for latency-sensitive workloads.

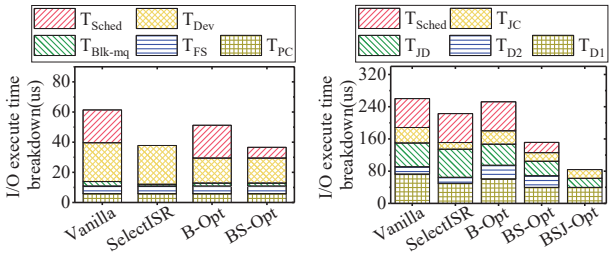
Synchronous write path. Fig. 13 shows the average, 99th, and 99.9th percentile response time (latency) of the *fillsync* workload when it co-runs with different workloads. Compared with the vanilla Linux, `SelectISR`, `B-Opt`, and `BS-Opt` only reduce the average response time by 10%, 6%, and 18%, respectively. However, `BSJ-Opt` can significantly reduce the average response time by 58%, the 99th response time by 83%, and the 99.9th response time by 74% on average. This implies that our journaling optimization is dominant for reducing the latency of synchronous writes.

Compared with *readrandom*, `B-Opt` yields less response time reduction for *fillsync*. The reason is that *fillsync* involves four I/O operations while *readrandom* only incurs one I/O operation. `B-Opt` can reduce the latency of critical requests in the device layer (i.e., T_{D1}, T_{JD}, and T_{JC} are reduced). However, `B-Opt` may even have a negative impact on reducing T_{D2}. Since dirty blocks of uncorrelated files involved in the compound transaction may be written back before `fsync()`, JBD2 has to wait for the completion of this operation. However, as these requests are not considered as critical, the blk-mq layer may throttle these I/O requests, and thus even increases T_{D2}, offsetting the benefits of `B-Opt`. This negative effect is particularly obvious for I/O intensive workloads, such as *Index*. This effect also results in even higher 99th response time.



(a) Co-running with *readrandom* (b) Co-running with *fillsync*

Figure 14: Normalized performance of throughput-oriented workloads when they co-run with *readrandom* or *fillsync*



(a) *readrandom* (b) *fillsync*

Figure 15: The latency breakdown of *readrandom* and *fillsync* in RocksDB when they co-run with *Bayes* (the definition of each portion of the time is described in Section 2.1)

Although the goal of `BS-Opt` is not to accelerate the write-back operations for other uncorrelated files, we still notice a moderate reduction (11%) of the average response time compared with `Vanilla`. `Fsync` introduces four I/O operations that all suffer from costly process scheduling, and there are also two times of process scheduling after waking up the JBD2 thread and the `fsync` thread. Since our process scheduler can promote the priority of critical I/O processes, `BS-Opt` can further reduce the 99th percentile latency compared with `B-Opt`. Similarly, `SelectISR` can avoid a long waiting time caused by process scheduling, and thus can reduce the average response time by up to 16% compared with `Vanilla`.

`BSJ-Opt` shows a significant reduction of response time for latency-sensitive workloads, as shown in Fig. 13. The reason is that `BSJ-Opt` allows the `fsync` thread itself to commit file-level transactions, and thus T_{Sched} is removed. The number of journal blocks is also reduced because only metadata blocks are written to ULL SSDs. More importantly, the `fsync` thread no longer needs to wait for the completion of uncorrelated data blocks, i.e., T_{D2} is removed. Overall, compared with `BS-Opt`, `BSJ-Opt` reduces the average, 99th, and 99.9th response time by up to 48%, 78%, and 64%, respectively. This demonstrates that the tail latency of `fsync` operations is mainly attributed to the cost of writing uncorrelated data blocks to the device, and our journaling scheme is effective for reducing the tail latency.

4.3 Throughput

Although we aim to reduce the response time of latency-sensitive workloads, the performance loss of throughput-oriented workloads is reasonable. Fig. 14 shows the throughput of throughput-oriented workloads when they co-run with *readrandom* and *fillsync*, respectively, all normalized to `Vanilla`. Overall, the performance degradation is marginal (no more than 6% for all applications). The reason is that throughput-oriented workloads are oblivious to the latency of each individual request, but are sensitive to the bandwidth of storage device. Our proposals can still maintain high utilization of the bandwidth of ULL SSD while reducing the latency of latency-sensitive workloads.

Overall, `SelectISR` reduces the total throughput of mixed workloads by 1.6% due to the polling scheme. Since `SelectISR` consumes a large amount of CPU resource for polling the I/O completion of critical requests, less CPU time is assigned to throughput-oriented applications. `B-Opt` reduces the application throughput by about 1.9% because we throttle non-critical requests and split large non-critical requests. `BS-Opt` further reduces the throughput by 1.2% because we postpone the scheduling of throughput-oriented workloads moderately, and thus reduce the CPU time for throughput-oriented workloads to some extend.

`BSJ-Opt` even improves the application throughput by 4% on average compared with `BS-Opt`, as shown in Fig. 14(b). The reason is that `f-transactions` are committed by critical processes while JBD2 does not have to commit normal transactions frequently for `fsync()`. As a result, less CPU time is spent in handling lock contention by JBD2, and more CPU time can be spent by throughput-oriented workloads.

4.4 Analysis of Latency in the I/O Stack

We further analyze the root cause of latency reduction in the I/O stack. Fig. 15 shows the breakdown of I/O latencies when *readrandom* or *fillsync* co-runs with *Bayes*.

In the read path, as shown in Fig. 15(a), there is no T_{Sched} for `SelectISR` because its polling scheme completely eliminates the cost of process scheduling. `B-Opt` reduces the latency in the device layer from 26μs to 17μs, because it can alleviate I/O contention in the SSD device by throttling non-critical burst I/O requests and splitting large requests. Throughput-oriented applications often generate burst I/Os which result in extremely large queue depth (even over 160). This is the root cause of high tail latency. Fig. 16 illustrates the high correlation between the spike latency in the device layer and the queue depth. Since `B-Opt` can sustain a relatively low queue depth (about 10), and thus significantly lowers the tail latency in the device layer compared with `Vanilla`. `BS-Opt` reduces the cost of process scheduling from 22μs to 7μs because our scheduler can identify awakened critical I/O processes and schedule them first in the next scheduling period. Similar to `BS-Opt`, `SelectISR` achieves almost equivalent improvement for CPU-intensive workloads such as *Bayes*. However, for some I/O intensive workloads such as *TeraSort*, `SelectISR` is no longer effective because the latency in the device layer (i.e., T_{Dev}) becomes dominant, as shown in Fig. 17. Moreover, `SelectISR` leads to heavy CPU load due to the polling mechanism, as shown in Fig. 18. When the request issue rate of *readrandom* and *fillsync* is not limited (about 50K requests/sec), `BSJ-Opt` only consumes 73.47% and 41.23% CPU resource, while `SelectISR` incurs almost 100% CPU utilization.

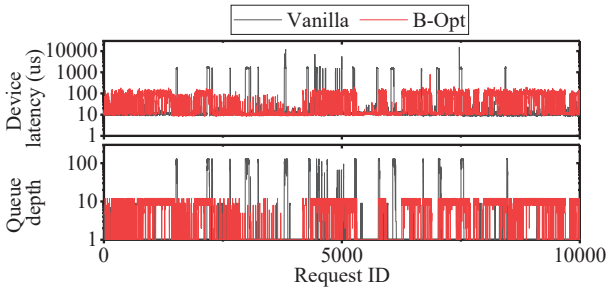


Figure 16: The device latency and the corresponding queue depth of SSD when *readrandom* co-runs with *Terasort*

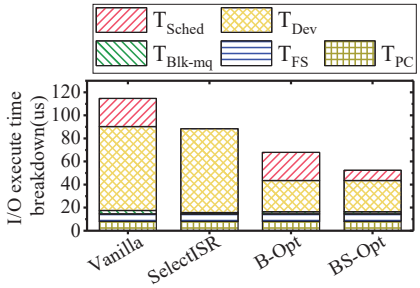


Figure 17: The latency breakdown of *readrandom* when it co-runs with *TeraSort*

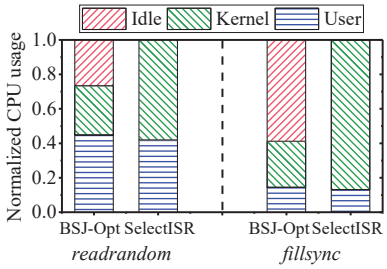
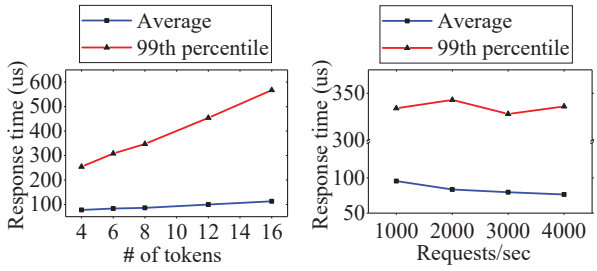


Figure 18: The breakdown of CPU usage for *readrandom* and *fillsync*, without limiting the request issue rate

In the synchronous write path as shown in Fig. 15(b), we configure JBD2 as a critical process in `B-Opt`, `BS-Opt`, and `SelectISR` so that the journaling operation can be scheduled as soon as possible. `SelectISR` can reduce T_{D1}, T_{D2}, and T_{JC} moderately because it polls the I/O completions for those I/Os. However, `SelectISR` has no impact on T_{Sched} since the wake-up of the JBD2 and `fsync` threads does not involve I/O operation. `B-Opt` can also reduce T_{D1}, T_{JD}, and T_{JC}. The I/O requests issued by these three operations are dispatched directly to the device. However, T_{D2} increases from 18μs to 34 μs because some dirty blocks of uncorrelated files involved in the transaction are written back before `fsync()`. However, since writing these dirty blocks are not considered as critical I/Os, the blk-mq layer throttles these requests and increases their latency. `BS-Opt` further reduces T_{D1}, T_{D2}, T_{JD}, and T_{JC} relative to `B-Opt` because these four I/O operations all involve process scheduling. Moreover, T_{Sched} is significantly reduced due to the same reason.



(a) Resp. time vs. tokens (b) Resp. time vs. request issue rate

Figure 19: The response time of *readrandom* when it co-runs with *TeraSort*, varying with the number of tokens and the request issue rate of *readrandom*

Since `BSJ-Opt` allows the `fsync` thread to commit file-level transactions by itself, there is no need to wake up JBD2 and `fsync` threads. Thus, T_{Sched} can be completely removed. As `BSJ-Opt` only records metadata blocks of the file to be written, the amount of metadata blocks is also reduced, i.e., T_{JD} is reduced. Moreover, because the `fsync` thread does not have to wait for writing back dirty blocks of uncorrelated files, T_{D2} can also be eliminated.

4.5 Sensitivity Studies

In this section, we explore how the performance of FastResponse is sensitive to some important parameters, such as the number of tokens that are used to throttle non-critical requests, and the request issue rate of critical I/Os.

The number of tokens. Fig. 19(a) shows the impact of tokens on the response time of *readrandom* in FastResponse when it co-runs with *TeraSort*. The tokens are used to limit the maximum number of non-critical requests that are received by the ULL SSD device in a period of time. We find that the average and 99th percentile response times of *readrandom* increase significantly when the number of tokens increases. Since a larger number of tokens implies that more non-critical requests can enter the device layer of the ULL SSD in a given time interval, the critical requests often suffer severe I/O interference due to resource contention, resulting in higher average and tail latencies.

Request issue rate. Fig. 19(b) shows how the request issue rate affects the average and 99th percentile response times of *readrandom* in FastResponse when it co-runs with *TeraSort*. When the request issue rate increases, the average response time of *readrandom* decreases while the tail response time keeps stable roughly. Since *TeraSort* leads to moderate bandwidth consumption on the ULL SSD and only generates burst I/Os sometimes, it has a little impact on most critical I/Os. When the request issue rate of *readrandom* increases, we find that the average response time even shows a small decline. This trend may be somewhat confusing but explainable. The reason is that the proportion of critical I/O requests that may be interfered by the burst I/Os of *TeraSort* is even reduced when the request issue rate of *readrandom* becomes larger. As the number of burst I/Os is not changed in a given time interval and most critical I/O requests still can be served with low latency, the response time on average is reduced. However, if a throughput-oriented application incurs continuous high pressure

Table 2: A comparison on cross-layer I/O optimizations

Approaches	Latency reduction on I/O paths			Apply to commercial SSD
	Filesystem	Block Layer	Process scheduling	
Split [15]	✔	✔	✗	✔
RCP [13]	✗	✔	✗	✔
FastTrack [14]	✗	✔	✗	✗
Flashshare [11]	✗	✔	✗	✗
FastResponse	✔	✔	✔	✔

on the ULL SSD and maintains a large queue depth, the response time of critical I/O requests would undoubtedly increase with the growth of request issue rate.

5 RELATED WORK

There have been many studies on the performance isolation of co-running workloads in a single I/O layer. At the file system layer, CCFS [32] and ijournaling [33] enable fine-grained transaction commit to reduce the latency of compound transactions. At the block layer, many I/O schedulers [16, 23, 34] are proposed to dispatch I/O requests according to their priority (weights). WRR [35] provides differentiated I/O services by optimizing the NVMe driver. For the I/O completion, a number of proposals exploit polling based strategies [11, 36–40] to reduce context switches. Although these proposals provide sufficient performance isolation for a certain layer in the I/O stack, the latency of I/O requests to ULL SSDs can be substantially affected by the interference in multiple layers, as aforementioned in Section 2.2.

Existing solutions resolving cross-layer I/O interference either need customized hardware or still incur relatively high latency on ULL SSD since they do not reduce the latency incurred in at least one layer of the I/O stack, as shown in Table 2. Split-level I/O scheduling [15] provides a set of I/O handlers for an I/O scheduler at three I/O layers: system call, page cache, and block layer. However, hooking system calls usually extends the I/O path, and thus results in higher latency. RCP [13] solves the problem of I/O-priority inversion between foreground tasks and background tasks based on a priority inheritance protocol. It is not designed for reducing the latency of multiple co-running foreground programs. For example, it does not provide efficient compound transaction execution, which incurs high latency [32, 33]. FastTrack [14] solves the same problem in the page cache and device layers by preempting the background task. However, these proposals are designed for traditional SSDs, and are not applicable for ULL SSDs to meet low latency requirements.

Flashshare [11] is the most relevant work to this paper. It punches through the I/O stack to pass attributes of applications from the OS kernel to the SSD firmware. Since Flashshare does not consider the impact of process scheduling and compound transactions on latency-sensitive I/O requests, it cannot achieve complete performance improvement for mixed workloads. More importantly, Flashshare is based on a special SSD and have to modify the SSD firmware, impeding its adoption. In contrast, FastResponse is a software approach, and thus is applicable to most commercial ULL SSDs. To the best of our knowledge, FastResponse is the first cross-layer solution for commercial ULL SSDs.

6 CONCLUSION

In this paper, we propose FastResponse, a holistic approach to reduce the response time of latency-sensitive workloads when they co-run with throughput-oriented workloads. We propose several schemes to mitigate I/O interference. First, we propose a new journaling scheme to reduce the long latency of committing compound transactions. Second, we optimize the block layer to throttle large requests of throughput-oriented workloads, and thus guarantee the QoS of latency-sensitive workloads. Finally, we modify Linux kernel's CFS process scheduler to assign high priority to latency-sensitive workloads, and thus shorten the waiting time of process scheduling. Experimental results show that FastResponse can reduce the average and the 99.9th response time of latency-sensitive workloads by 18-70% and 58-80%, respectively, while still maintaining similar throughput for co-located throughput-oriented workloads.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their constructive comments. This work is supported jointly by National Natural Science Foundation of China (NSFC) under grants No.62072198, 61732010, 61825202, 61929103, and US National Science Foundation (NSF-CAREER-1943071, NSF-SPX-1919126). Haikun Liu is the corresponding author of this paper.

REFERENCES

[1] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Jack Zhang, and Junhua Wang. Perfiso: Performance Isolation for Commercial Latency-Sensitive Services. In *Proceedings of 2018 USENIX Annual Technical Conference (ATC '18)*, pages 519–532, 2018.

[2] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[3] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, pages 361–378, 2019.

[4] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan. Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 1241–1255, 2020.

[5] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 173–190, 2020.

[6] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 281–297, 2020.

[7] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Christina Delimitrou, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, pages 19–33, 2019.

[8] Intel Optane SSD 900P Series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/optane-ssd-900p-brief.html, 2017.

[9] Samsung Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf, 2017.

[10] XL-Flash 3D NAND. https://www.anandtech.com/show/13183/toshiba-announces-xlflash-lowlatency-3d-nand, 2018.

[11] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. FLASHSHARE: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and*

Implementation (OSDI '18), pages 477–492, 2018.

[12] Jiahao Liu, Fang Wang, and Dan Feng. CostPI: Cost-Effective Performance Isolation for Shared NVMe SSDs. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP '19)*, pages 1–10, 2019.

[13] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 345–358, 2017.

[14] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones. In *Proceedings of 2018 USENIX Annual Technical Conference (ATC '18)*, pages 15–28, 2018.

[15] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, pages 474–489, 2015.

[16] Till Miemietz, Hannes Weisbach, Michael Roitzsch, and Hermann Härtig. K2: Work-Constraining Scheduling of NVMe-Attached Storage. In *Proceedings of 2019 IEEE Real-Time Systems Symposium (RTSS '19)*, pages 56–68, 2019.

[17] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. D2FQ: Device-Direct Fair Queueing for NVMe SSDs. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST '21)*, pages 403–415, 2021.

[18] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*, pages 397–410, 2018.

[19] FastResponse. https://github.com/CGCL-codes/FastResponse, 2022.

[20] RocksDB. https://github.com/facebook/rocksdb, 2017.

[21] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *Proceedings of 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW '10)*, pages 41–51, 2010.

[22] FIO: Flexible I/O Tester. https://github.com/axboe/fio, 2017.

[23] Paolo Valente and Arianna Avanzini. Evolution of the BFQ Storage-I/O Scheduler. In *Proceedings of 2015 Mobile Systems Technologies Workshop (MST '15)*, pages 15–20, 2015.

[24] Block layer introduction. https://lwn.net/Articles/738449/, 2017.

[25] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-Latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC '19)*, pages 603–616, 2019.

[26] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories. In *Proceedings of 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*, pages 385–395, 2010.

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

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

[29] Dbbench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2017.

[30] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *Proceedings of 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14)*, pages 301–312, 2014.

[31] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: A Big Data Benchmark Suite from Internet Services. In *Proceedings of 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*, pages 488–499, 2014.

[32] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. *ACM Transactions on Storage*, 13(3):1 – 29, 2017.

[33] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *Proceedings of 2017 USENIX Annual Technical Conference (ATC '17)*, pages 787–798, 2017.

[34] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-Queue Fair Queuing. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC '19)*, pages 301–314, 2019.

[35] Kanchan Joshi, Kaushal Yadav, and Praval Choudhary. Enabling NVMe WRR Support in Linux Block Layer. In *Proceedings of 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*, 2017.

[36] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll is Better than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies, (FAST '12)*, pages 25–31, 2012.

[37] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *Proceedings of 2013 USENIX Annual Technical Conference (ATC '13)*, pages 309–320, 2013.

[38] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring System Challenges of Ultra-Low Latency Solid State Drives. In *Proceedings of 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '18)*, 2018.

[39] I/O Latency Optimization with Polling. https://events.static.linuxfound.org/sites/events/files/slides/lemoal-nvme-polling-vault-2017-final_0.pdf, 2017.

[40] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*, pages 1–13, 2018.