ZMS: Zone Abstraction for Mobile Flash Storage

Joo-Young Hwang, Seokhwan Kim, Daejun Park, Yong-Gil Song, Junyoung Han, Seunghyun Choi, and Sangyeun Cho, *Samsung Electronics;*
Youjip Won, *Korea Advanced Institute of Science and Technology*

https://www.usenix.org/conference/atc24/presentation/hwang

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

ZMS: Zone Abstraction for Mobile Flash Storage

Joo-Young Hwang[1], Seokhwan Kim[1], Daejun Park[1], Yong-Gil Song[1], Junyoung Han[1], Seunghyun Choi[1], Sangyeun Cho[1], and Youjip Won[2]

[1]*Samsung Electronics*
[2]*Korea Advanced Institute of Science and Technology*

Abstract

We propose an I/O stack for ZNS based flash storage in mobile environment, *ZMS*. The zone interface is known to save the flash storage from two fundamental issues which modern flash storage suffers from: logical-to-physical mapping table size and garbage collection overhead. Through extensive study, we find that realizing the zone interface in mobile environment is more than a challenge due to the unique characteristics of mobile environment: the lack of on-device memory in mobile flash storage and the frequent `fsync()` calls in mobile applications. Aligned with this, we identify the root causes that need to be addressed in realizing the zone interface in mobile I/O stack: *write buffer thrashing* and *tiny synchronous file update*. We develop a filesystem, block I/O layer, and device firmware techniques to address the above mentioned two issues. The three key techniques in ZMS are (i) IOTailor, (ii) budget-based in-place update, and (iii) multi-granularity logical-to-physical mapping. Evaluation on a real production platform shows that ZMS improves write amplification by 2.9–6.4× and random write performance by 5.0–13.6×. With the three techniques, ZMS shows significant performance improvement in writing to the multiple zones concurrently, executing SQLite transactions, and launching the applications.

1 Introduction

Zoned namespace (ZNS) SSD [1–3] is proposed to address the two key technical issues which the modern flash storage faces; garbage collection overhead [4–22] and the memory overhead of logical to physical mapping table [23–32]. ZNS SSD saves the flash storage from the two issues. Due to its append-only write nature, ZNS SSD can use zone granularity mapping. Assuming that the zone size is 1 GiB, the size of the mapping table for zone granularity mapping is $1/10^6$ times smaller than that for the page mapping.

In this work, we propose a ZNS for mobile storage device. The potential benefits of enabling zone interface in mobile flash storage are clear. First, eliminating the garbage collection from the storage device, the write amplification will improve, thereby extending the lifespan of the mobile device. Second, the mobile device becomes better responsive to the user input [33, 34]. Third, the read performance improves since the mapping table of mobile flash storage can be fully loaded into on-device memory. Legacy mobile flash storage caches only a fraction of the mapping table since the mapping table is much larger than on-device memory. Despite all these potential benefits, there have been little efforts that explore the zone abstraction for mobile flash storage. JEDEC, which is a standardization body for memory and storage, recently started to discuss the zone interface for the mobile UFS (Universal Flash Storage) [35]. We carefully argue that the zone interface for mobile device deserves more attention from researchers as well as from practitioners than it does now.

ZNS has originally been proposed for the server SSD which has sufficient amount of DRAM and power loss protection (PLP) feature. Mobile storage does not have these luxuries. Mobile storage does not have DRAM and is loaded with small amount of SRAM, e.g. 2 MiB. Mobile storage does not have PLP feature either. This critical difference of hardware features between server SSD and mobile flash storage imposes unique challenges in using the zone interface with mobile flash storage. We identify two key technical challenges when designing zone abstraction for mobile flash storage: *write buffer thrashing* and *tiny synchronous file update*.

Write Buffer Thrashing. We find that the zone interface suffers from excessive write buffer flushes when it is used for the mobile flash storage. We call this phenomenon as *Write Buffer Thrashing*. Write buffer thrashing is caused by two reasons combined together: lack of write buffer and large programming unit of the underlying flash storage. A flash storage device has a write buffer where the user data is buffered until the total amount of the user data reaches the size of a programming unit. Due to the lack of on-device memory, mobile flash storage cannot allocate the separate write buffers for individual open zones. Instead, the open zones need to share the write buffers. Each time when the storage device switches the write buffer from one zone to another, it flushes the write buffer.

The lack of on-device memory causes the write buffer to be flushed not only frequently but also, more importantly, *prematurely*. Modern multi-level cell flash storage device uses large programming unit which is a multiple of a flash page. If the amount of the data blocks to be programmed is smaller than the programming unit, the device firmware flushes the write buffer's content to a single-level-cell (SLC) buffer, which are later migrated to the multi-level cell block. This unaligned buffer flush doubles the amount of writes in the flash storage device and significantly reduces the performance and lifespan of the storage device.

Tiny Synchronous File Update. Mobile I/O workload is known for its excessive `fsync()` calls [36]. All Android applications use SQLite to persistently manage data. SQLite persists every individual update to its database file and to its log file via `fsync()`. Subsequently, with few exceptions, the mobile applications frequently synchronize the small update to the storage device. To handle this tiny synchronous file update efficiently, F2FS uses in-place update policy instead of append-only logging that incurs writing of file metadata. This in-place update policy of F2FS directly contradicts with the append-only nature of the zone interface. If we are to synchronize the tiny file update to the zoned storage device, we do not have any choice but to do it inefficiently, i.e. in append-only manner.

We develop the **z**oned **m**obile I/O **s**tack (ZMS) that effectively addresses the two challenges: write buffer thrashing and tiny synchronous file update. ZMS spans all layers of the I/O stack. For storage device, we develop zoned UFS, from commodity UFS product, that implements a *budget-based in-place update* to address the performance issues associated with synchronizing the small file update. For filesystem, we modify the stock F2FS to implement the budget-based in-place update. For the host side block I/O layer, we develop *IOTailor* to address the write buffer thrashing. IOTailor reshapes the incoming workload from the file system and makes it aligned with a superpage size of the underlying storage device so that the FTL can fully utilize the device's internal parallelism while avoiding unaligned buffer flush.

To the best of our knowledge, this work is the first design and implementation of the zone interface on mobile flash storage. We use original F2FS running on a block interface device as the baseline for comparison. Compared with the baseline, ZMS shows the similar performance for basic I/O in clean condition. ZMS manifests itself in aged condition. In deeply aged condition, ZMS shows 5.0–13.6$\times$ better random write throughput than the baseline, and reduces the overall end-to-end write amplification by as much as 2.9–6.4$\times$ against the baseline. We show that IOTailor reduces the amount of SLC buffering and improves performance for concurrent writes to multiple open zones. ZMS shows 37–44% higher throughput in wide range random read workloads, which contributes to the 6–12% reduction of application launch times. In SQLite benchmark, ZMS shows 60–100% better throughput for SQLite write-ahead-log mode journal. Our contributions are three-fold:

- **Identification of the challenges.** We find that the zoned mobile device suffers from excessive write buffer flushing activity and performance degradation in serving synchronous file updates.
- **Development of ZMS.** We develop ZMS, a novel I/O stack for zoned mobile storage, consisting of zoned UFS and the host side techniques to address the two challenges.
- **Performance analysis.** With comprehensive performance evaluation and analysis, we show the benefits of the zone interface in mobile flash storage and the effectiveness of the proposed techniques.

The remainder of this paper is organized as follows. We describe background and motivations of this work (§2), and detail the two key challenges (§3). We overview ZMS (§4), then describe zoned UFS (§5) and the host side techniques (§6). We describe performance evaluation results (§7) and related works (§8), then conclude (§9).

2 Background and Motivations

2.1 Mobile Flash Storage Internals

A mobile flash storage device consists of a main controller, limited volatile memory and NAND memories (Figure 1). In this work, we use the mobile flash storage and UFS storage interchangeably since UFS is a *de facto* standard for modern mobile flash storage interface. An embedded software, called flash translation layer (FTL) runs on the controller, performing translation of logical block address (LBA) to physical page address (PPA). The volatile memory is used for keeping the code and the data for the FTL and the logical-to-physical (L2P) mapping table. The L2P mapping table is loaded to memory in an on-demand manner if the volatile memory is not large enough to load the entire L2P table.

There are mainly three operations in flash device; read, program (write) and erase. A flash block (or erase block) is an array of flash pages. Flash memory has erase-before-write idiosyncrasy: a flash block should be erased before being written. In modern flash memory, the size of a page is typically 16 KiB. For a single-level cell (SLC) flash block, SSD controller allows the page to be programmed in a sub-page unit, e.g. 4 KiB. In multi-level cell device, e.g. TLC or QLC, the programming unit is a multiple of a flash page. For example, in TLC flash with 16 KiB page size, SSD controller programs three flash pages at a time. A flash chip may also have multiple planes. SSD controller can program the multiple pages that are at the same offset from the beginning of each plane together. Combining these, the TLC programming unit for a chip with 16 KiB page with 2 planes is 96 KiB (= 16 KiB $\times$ 3 $\times$ 2 planes). A flash block can be programmed in different modes, e.g. SLC or TLC, which can be set when the flash

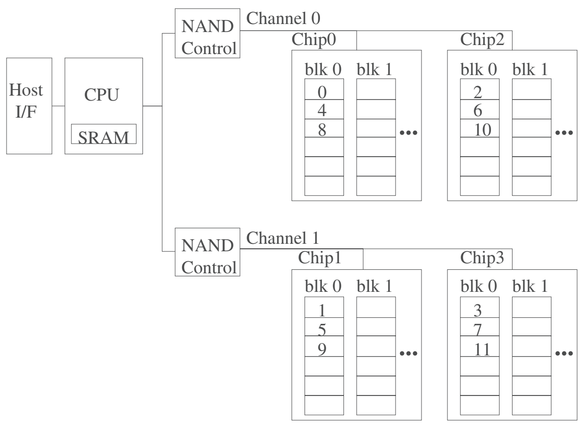


Figure 1. Internals of a UFS device having four TLC chips in two channels

block is erased.

Modern flash storage devices adopt multi-channel and multi-way architecture to achieve higher performance. The data is striped over multiple channels and multiple ways. In Figure 1, we show the internal structure of a UFS device having two channels and two chips per channel. Each chip has two planes. Data is striped over the four chips and the stripe unit is a pair of TLC pages that are on different planes in a chip. The group of programming units at the same offsets from each chip is called a *superpage*. The stripe units {0, 1, ..., 11} of Figure 1 forms a superpage and the size of a superpage is 384 KiB (= 16 KiB $\times$ 2 planes $\times$ 3 pages $\times$ 4 chips). The group of erase blocks at the same offsets from each chip is called a *flash block group* or a *superblock*. In Figure 1, the superblock 0 is the group of the first blocks (blk 0) of all the chips. A write buffer is set to be as large as a superpage to fully exploit the internal parallelism. When a device has multiple open superblocks, the total amount of write buffers that are needed corresponds to the size of a single write buffer multiplied by the number of open superblocks.

Table 1. Server SSD vs. mobile flash device (UFS)

	Server SSD	UFS
DRAM support	Yes	No
# Write buffers	16	2
L2P map caching	Fully loaded	Demand paging
PLP	Yes	No

Server SSD vs. Mobile Flash Storage. In comparison to server-grade SSDs, mobile flash devices have two limitations. First, due to cost and form factor constraints, it does not have DRAM and has only very limited SRAM (a few MiB) on its controller. The number of write buffers is much smaller than server SSDs (Table 1). In addition, L2P map is loaded on demand, so its random read performance depends on the locality of requests. Second, power loss protection (PLP) for buffered data is not provided. So, buffered data should be flushed immediately when host requests `fsync()`. Reducing the latency of `fsync()` is particularly important in mobile flash storage because synchronous write latency significantly affects the application responsiveness.

2.2 Block Abstraction for Flash Storage

Legacy block device abstraction views the storage device as an array of blocks. In block device abstraction, the host can read and write any block in the storage device via supplying the block address to the storage device. Unlike block device abstraction, the flash device cannot be overwritten. A flash block needs to be erased before it is reprogrammed. Flash storage adopts two schemes to hide the erase-before-write characteristics and make itself as a conventional block storage: L2P mapping and garbage collection. The two schemes leave flash storage under two fundamental technical challenges [37]: *write amplification* and *L2P mapping table overhead*.

Write Amplification. Write amplification in flash storage has been subject to intense research for decades [4–6, 8–12, 38]. The root cause of write amplification is garbage collection. There are two key ingredients to reduce the garbage collection overhead and subsequently to reduce the write amplification. First is to cluster the data blocks of similar lifespan together at the same erase block. A few works proposed a mechanism, where the host provides entropy of the data blocks [15, 20] or *stream* id of the data blocks [7, 13, 14, 18, 19, 38]. SSD controller can cluster the data blocks with the similar entropy or with the same stream id together. These pieces of information are provided by the host to the device as hints. The host does not have any control on determining the physical location of the data blocks. Second is to accurately determine whether a given flash page contains the valid content or not. There is a non-zero time interval between when the filesystem invalidates a data block, e.g. `unlink()` and when the filesystem informs the device about the invalidated block. The garbage collection module of the device may blindly migrate the flash page whose contents are no longer valid at the filesystem. Unfortunately, the host is very discreet in informing the storage device about the invalidated file blocks since it can negatively interfere with the other foreground filesystem activity [11].

L2P Mapping Table Size. Mapping table size becomes a more substantial issue as the capacity of the flash storage increases. Flash storage employs the page mapping [39–45] to reduce write amplification. In the page mapping scheme, each L2P mapping entry encodes the physical address of a logical page (typically 4 KiB). When using 4 bytes for a mapping entry, the mapping table corresponds to approximately 0.1% of the device capacity. For a 4 TiB SSD, there needs to be 4 GiB of DRAM to fully load the mapping table in memory. When a device cannot have the entire L2P mapping table in memory, a fraction of the L2P mapping table is loaded in an on-demand manner [39]. Random read performance can be degraded when the required mapping entries are not resident in memory.

2.3 Zone Abstraction for Flash Storage

Zone abstraction views the storage device as a set of zones, each of which is a fixed size array of blocks and each of which can be written only in an append-only manner. Zoned storage device has three essential operation; read, write (append) and zone-reset. A zoned device maintains a write pointer for each open zone, indicating the next location to write within a zone.

Zone abstraction addresses the aforementioned problems of block abstraction [1, 2, 46, 47]. In ZNS SSD, the host can control the physical data placement and therefore can potentially eliminate the device-level garbage collection. ZNS SSD can significantly reduce the mapping table size via zone-granularity L2P mapping.

Zone abstraction was proposed for server-grade flash storage where the flash storage has sufficient amount of DRAM and a large degree of parallelism with multi-channel/multi-way architecture [48–50]. Sustained throughput and the tail latency are important in server-grade storage. We carefully argue that zone abstraction can fit well for the mobile flash storage where the user perceived latency, i.e. *responsiveness*, matters the most. For responsiveness, random read performance and write amplification are of prime concerns. In particular, write amplification is of important concern in mobile storage because the read latency can get worse as the underlying flash storage ages [51–54]. The benefit of the zone abstraction becomes more substantial in aged condition because the devices suffer from the garbage collection and severe file system fragmentation [31, 55].

2.4 F2FS

F2FS is a log-structured file system built for flash memory based storage [20]. It is now being employed by Android platform as a default file system. Here, we describe F2FS design features that are relevant for the zone interface.

Section in F2FS vs. Zone in ZNS. F2FS has geometry structures that are friendly to zoned devices. F2FS defines two essential units; *segment* and *section*. Segment (2 MiB) represents a set of filesystem blocks that need to be clustered and written together. Section consists of a fixed number of segments and is a unit of filesystem level garbage collection. All blocks in a section are reclaimed together when F2FS performs garbage collection. By aligning the sections of the F2FS with the zones in the zoned storage device, we can make each F2FS section represent an individual zone.

Writing Modes. F2FS supports three writing modes: append logging, threaded logging and in-place update (Figure 2). The append logging writes the data blocks in an append-only manner. Threaded logging writes the data blocks at the invalid blocks of a segment [20]. Threaded logging cannot be used on zoned devices because it generates random writes. F2FS uses in-place update to quickly service `fsync()`. File metadata in F2FS, called node blocks, are not modified when the data block is updated in place. Hence the in-place update of F2FS

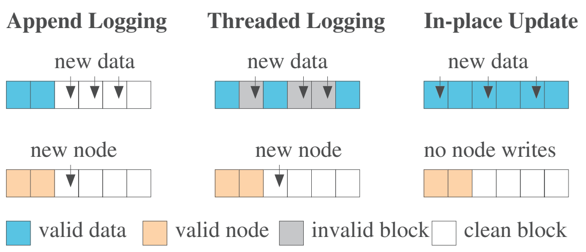


Figure 2. Three modes of writing in F2FS

does not write node blocks. This greatly improves the fsync performance. However, in-place update cannot be used in zoned devices because the in-place update may render random write at the storage device.

Temperature-aware Data Placement. F2FS employs temperature-aware data separation to reduce its garbage collection overhead. It maintains six active logs each of which is dedicated to accommodate the segments with the same temperature and the same block type. F2FS defines three levels of temperature (hot, warm and cold) and two block types (data and node). In devices with legacy block abstraction, it is possible that the data blocks in the different segments are collocated in the same erase block. On the same token, the data blocks at the same segment can be placed at the different erase blocks in the storage. By allocating a separate zone for each active log, F2FS can physically cluster the filesystem blocks of the same active log at the storage device.

3 Design Challenge

3.1 Unaligned Buffer Flush

The storage device temporarily stores the incoming data from the host to the write buffer until a sufficient amount of data, programmable to a flash block, is collected. When the storage controller adopts multi-channel/multi-way organization, the storage controller waits till the amount of data blocks in the write buffer reaches the superpage size to fully utilize parallelism. In some situation, FTL needs to flush the write buffer even though the amount of data in the write buffer falls short of the size of the programming unit. We call this phenomenon as *Unaligned Buffer Flush*. In mobile flash storage, there are two causes of unaligned buffer flush: `fsync()` request from the host or write buffer switching. In both cases, the storage controller should flush the write buffer even though the amount of data in the write buffer does not reach the programming unit size. We call this situation as write buffer is flushed *prematurely*. This situation is more likely to happen in mobile environment than in server environment. Unlike in ZNS SSDs for server environment, `fsync()` is called frequently due to the absence of PLP and the write buffer switch occurs due to insufficient on-device memory in mobile environment.

Figure 3 illustrates how the data in a write buffer are written to the flash device. Each rectangle numbered from 0 to 9 denotes the stripe unit, a pair of flash pages (32 KiB). The

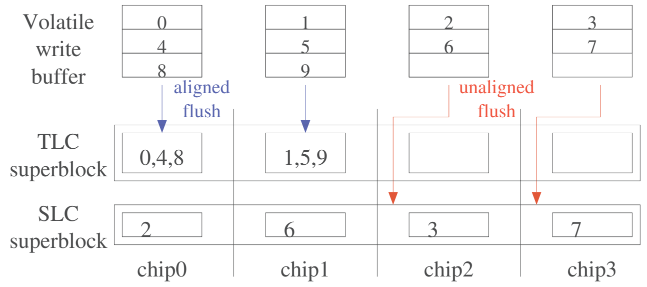


Figure 3. A scenario of flushing write buffer of an open zone.

numbers denote the striping order. We use the term *volatile buffer* to distinguish it from the SLC-based write buffer. The size of a volatile write buffer corresponds to a superpage size. The buffered data for chip 0 and 1 are aligned with the TLC programming unit (96 KiB), so they can be programmed to a TLC block. In contrast, the amount of data that need to be flushed to chip 2 and 3 are less than the programming unit. So, they are programmed to a SLC block.

It is a simple way of handling unaligned buffer flush to pad dummy data to the write buffer to form a TLC programming unit. However, it is not recommended since it wastes storage space and makes the write pointer management very cumbersome. When the flash controller pads the dummy data to the write buffer, there comes a gap between the amount of data which the host writes to the storage and the amount of data written at the flash memory. It requires a complicated protocol between the host and the device to make the write pointer match with the TLC block's write offset.

To handle the unaligned flush, we propose to use SLC blocks as a supplementary *non-volatile write buffer*. A SLC block can be programmed in a multiple of 4 KiB. When a device has to handle unaligned flush, the device logs the data into a SLC block. The data buffered in the SLC block are written to a TLC block as soon as the amount of data in the SLC block combined with the data in the volatile write buffer reaches the TLC programming unit size. For instance, when the host would write stripe 10 in Figure 3, the stripe units {2,6,10} would be flushed to the TLC block in the chip 2.

3.2 Write Buffer Thrashing

Running F2FS on the mobile zoned device is subject to a large number of unaligned flushes because the small number (two or less, typically) of write buffers are shared among more open zones (as many as six when running F2FS file system) than the number of write buffers. We call this phenomenon as *Write Buffer Thrashing*. When running F2FS on a zoned device, F2FS maintains up to six active logs each of which corresponds to a zone. In ZNS SSD (for server), each of the open zones can be allocated a dedicated write buffer as shown in Figure 4a. So, ZNS SSD is free from write buffer thrashing. In contrast, since there exist only two write buffers available in zoned UFS, the write buffers should be shared among the six

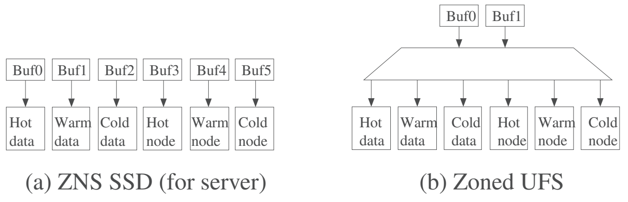


Figure 4. Write buffer allocation for open zones

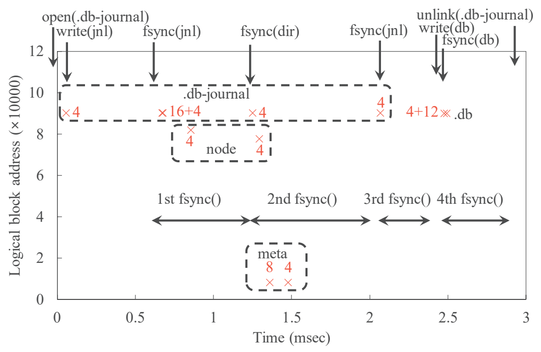
zones as shown in Figure 4b. When reclaiming a write buffer being used for a zone A for another zone B, the buffered data should be flushed to the zone A in order to accommodate data for the zone B. If the data in the write buffer is smaller than the TLC programming unit, it may render unaligned flush. In handling the unaligned flush, the buffered data are written to SLC buffer. SLC buffering incurs double writes (once written to a SLC block, then migrated to a TLC block). As a result, write buffer thrashing hampers the performance and the lifespan of the storage device. Therefore, it is important to avoid or minimize the write buffer thrashing in designing a mobile zoned device.

3.3 Tiny Synchronous File Update

A file can be updated in two ways; overwrite and append. In in-place update filesystem such as EXT4, overwriting an existing file does not require any updates in the file map. In contrast, out-of-place update file systems like F2FS modifies the file map no matter whether a file content is over-written or is appended at the end of the existing file. In F2FS, therefore, when the application synchronizes the filesystem state to the disk, it not only flushes the updated file block but also the updated node block (file map) to the storage. Subsequently, when the application calls an `fsync()`, F2FS may write twice as many blocks to the storage as EXT4 does.

In mobile environment, it is critical to handle the small size `fsync()` efficiently. SQLite, a widely used embedded database in Android, relies on its own journal file for data recovery. Let us briefly explain how the SQLite's `insert()` transaction flushes the filesystem blocks to the storage. Figure 5a illustrates the IO trace in inserting a 100 byte record to SQLite DBMS using rollback journaling. In a single `insert()` transaction, SQLite calls `fsync()` four times. They are for synchronizing the result of each of the following activities: creating the journal file, updating the associated directory entry, undo logging at the journal file, and updating a database file. When updating the database file, SQLite overwrites the updated data blocks to the database file. After the transaction completes, SQLite deletes, truncates or persists the journal [36].

To efficiently handle small `fsync()`, F2FS selectively applies in-place update (IPU) policy in serving `fsync()` for requests smaller than a configurable parameter, `min_fsync_blocks` (32 KiB by default). This optimization effort is successful in handling the `fsync()` efficiently. De-


(a) Rollback journal w/ legacy UFS

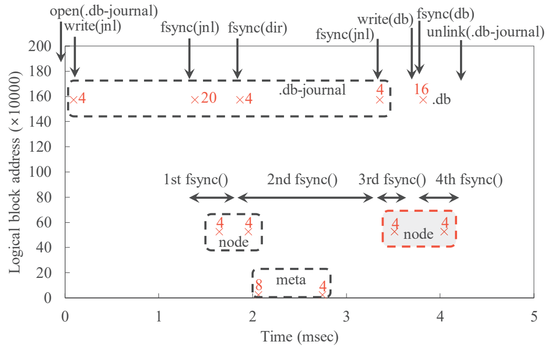

(b) Rollback journal w/ ZMS

Figure 5. IO access pattern of an SQLite, `insert()` transaction, record size = 100 byte, DELETE journal mode

spite four `fsync()` calls, the node blocks are written only twice: one for the journal file (1st `fsync()`) and the other for the directory (2nd `fsync()`). Please refer to the rectangle labeled as 'node'.

Unfortunately, however, this IPU policy cannot be used on zoned devices because zoned devices do not allow zones to be overwritten. We disable the IPU in F2FS and perform the same `insert()` transaction. Figure 5b illustrates the IO trace captured in executing the `insert()` transaction with the IPU policy turned off. Here, F2FS updates the database file and the journal file in an append-only manner. With the IPU policy disabled, F2FS writes additional two node blocks, which prolongs the `fsync()` latency. Please refer to the rectangle with grey background, labeled as 'node'. One node is for the journal file (3rd `fsync()`) and the other is for the database file (4th `fsync()`).

4 ZMS: an overview

In this work, we develop a zoned mobile I/O stack, *ZMS*. ZMS consists of the filesystem layer and the block device layer at the host, and the zoned UFS device. Before we delve into details, we first examine its mechanism through an example.

In Figure 6, we assume applications are writing warm and cold data concurrently. F2FS writes these data to warm data zone, denoted as Z_{wd}, and cold data zone, denoted as Z_{cd}, respectively. There are two write buffers in the storage device. They are labeled as Buf_0 and Buf_1, respectively. Without IOTailor, writes to the Z_{wd}, denoted as W_{wd}, and writes to the Z_{cd}, denoted as W_{cd}, are sent to device and buffered in Buf_0 and Buf_1 of the device, respectively. When applications write hot data, one write buffer needs to be reclaimed and switched to accommodate the newly incoming hot data. Let Z_{hd} represent the hot data zone. In this example, we assume that Buf_0 is selected for Z_{hd}. Before switching Buf_0 to Z_{hd}, the data in Buf_0 needs to be flushed. If the amount of data in Buf_0 is smaller than the size of TLC programming unit, it renders unaligned flush and the data is flushed to a SLC buffer block.

IOTailor transforms the write requests to a zone to superpage-aligned requests. IOTailor ensures that all buffered data is programmable to a superpage, which eliminates unaligned flush and utilizes full internal parallelism. With IOTailor, W_{wd} and W_{cd} are enqueued to the warm data queue and the cold data queue of IOTailor, respectively. IOTailor transforms W_{wd} and W_{cd} to W^t_{wd} and W^t_{cd}, respectively, that are aligned with the size of the superpage. Then, W^t_{wd} and W^t_{cd} are sent to the device in a round-robin fashion. Since the write requests to the device are ensured to be aligned with superpage, unaligned buffer flush does not occur.

When F2FS performs the garbage collection for a victim zone, Z_v, it copies the valid data blocks of Z_v to Z_{cd}. In consolidating the valid blocks in the victim zone, we propose to offload the copy operation to the storage device. This is to save the host's CPU from processing copy operations and the overhead of transferring data between the host and the device.

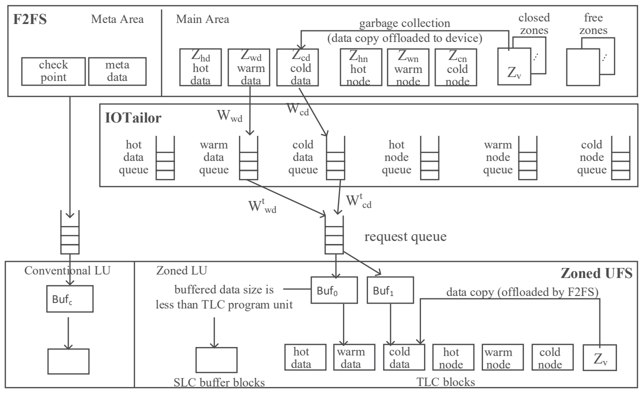


Figure 6. Overview of ZMS I/O stack

5 The Zoned UFS Device

5.1 Organization

We develop a zoned UFS complying with a zoned block command (ZBC) specification [56] by revising the firmware of a commercial legacy UFS product with block interface. The host system can get information of zone size, superpage size and the number of write buffers by reading vital product data (VPD) pages as defined in the UFS standard [35].

Logical Units. A logical unit (LU) refers to a contiguous logical address range of a device. The zoned UFS provides two types of LU: a conventional LU providing block interface, and a zoned LU providing zone interface. The conventional LU is for the F2FS metadata area that is to be updated randomly. The main area of F2FS is placed in the zoned LU.

Flash Blocks and Volatile Memory. The device has four 256 Gbit NAND chips connected to the storage controller via two channels (two chips per channel). There are 943 superblocks, 938 of which are allocated for zoned LU. The controller does not have DRAM, only SRAM memory. The L2P map cache size directly affects random read performance. The legacy device, used as the baseline of our evaluation, has 1 MiB for L2P map cache. For fair performance comparison, we use the same size of memory for L2P map cache in both the baseline and the zoned UFS. A write buffer for a zone is 384 KiB which corresponds to the size of a superpage. The device provides the memory for up to three write buffers. The size of the TLC programming unit is 96 KiB. In its original setting of the legacy device, the three buffers are used for the following three superblocks, respectively: one for writes less than 32 KiB, one for writes equal to or larger than 32 KiB, and another for garbage collection. For the zoned UFS, we allocate one write buffer for the conventional LU and two write buffers for the zoned LU.

Zone Size. Zone size is a critical design parameter that impacts per-zone performance and file system garbage collection. The per-zone performance depends on how the device stripes the zone data over the chips. There are two approaches of the zone data placement: full striping and partial striping. In the full striping approach, a zone's data is striped over all available chips and a zone corresponds to a superblock (§2.1). The partial striping approach is to reduce the stripe width. With the full striping approach, we can fully utilize internal parallelism. However, with more erase blocks in a flash block group, it takes longer for F2FS to perform garbage collection and subsequently the I/O requests need to be blocked for longer period of time. The partial striping approach can better separate the data of the different applications and reduce garbage collection latency whereas per-zone performance is reduced. In this work, we choose the full striping approach because responsiveness for foreground applications is of prime concern in a single user mobile environment [57] and providing higher performance improves responsiveness.

5.2 Multi-granularity L2P Mapping

Zoned UFS uses three different L2P mapping granularities: page (4 KiB), chunk (4 MiB) and zone (a superblock). We call our mapping as multi-granularity L2P mapping. Due to the lack of memory in the mobile flash device, the multi-granularity L2P mapping table is cached on demand.

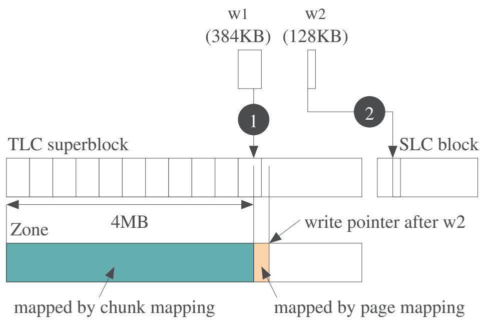


Figure 7. Multi-granularity L2P mapping

For zones that are fully written sequentially, we use zone-granularity mapping (or zone mapping for brevity). One advantage of zone abstraction is that we can use zone mapping that greatly reduces the mapping table size. Zone mapping requires that entire zone data is physically contiguous. For SLC blocks, zoned UFS employs page mapping. The data blocks in ZMS is not always contiguously written to a TLC block for two reasons (§3.1). First is when the device switches a write buffer of a zone for another zone. Second is when the host requests `fsync()`.

To further reduce the memory requirement for caching the page mapping table, we use chunk granularity mapping. In on-demand page mapping, a 4 KiB L2P map page contains the array of physical page numbers for 1024 contiguous logical pages, called chunk. If all of the physical page numbers in a map page are contiguous, a single 4 byte entry (chunk mapping entry) can represent 1024 contiguous pages. By doing so, we can reduce memory used for L2P map caching by 1024 times.

Figure 7 illustrates the operation of the multi-granularity mapping. We assume that ten superpages were written to a zone and the two write requests (w1 and w2) arrived. The w1 is written to the TLC block (❶) because it is superpage aligned. The w2 request (unaligned buffer flush) is written to a SLC block (❷). After serving w1, the 4 MiB region of the zone (the green box) becomes physically contiguous, so chunk mapping is used for the region. For the other pages of the zone, page mapping is used.

6 Filesystem and Block Layer in ZMS

6.1 IOTailor

To address the write buffer thrashing problem, we propose a novel I/O reshaping method, called *IOTailor*. IOTailor sits between the filesystem and the legacy block device layer. IOTailor ensures that I/O requests sent to the device are aligned with a superpage size. IOTailor achieves maximum device write performance and avoids unaligned flush when switching the write buffers. IOTailor maintains a number of queues each of which corresponds to a zone opened by the filesystem. We call it a *per-zone queue*. In current implementation, IOTailor

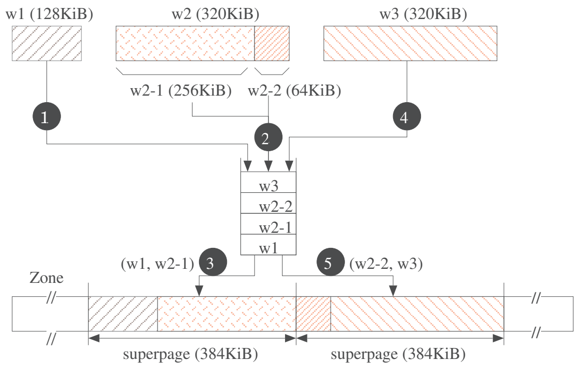


Figure 8. Request reshaping by IOTailor (request order: w1, w2, w3)

defines six queues since it is designed to work with F2FS which may open up to six zones.

IOTailor performs two tasks; *request split* and *request grouping*. First is request split. When a request from file system arrives, it examines whether the request crosses the superpage boundary. If it does, IOTailor splits the request on the superpage address into two requests: the one that ends at the preceding superpage and the other that starts at the following superpage. When IOTailor splits the request, it creates the child requests that are associated with the original request. Once all child requests are complete, the original request is marked as completed. IOTailor enqueues the split requests into the associated per-zone queue. Second is request grouping. When the set of requests in a per-zone queue forms a superpage, it is bundled into a single I/O group and is passed to the I/O request scheduler. The I/O scheduler coalesces them if possible. IOTailor processes the per-zone queues in a round-robin manner for fairness.

The operation of IOTailor is illustrated in Figure 8. The request w1 is enqueued to the queue for its zone (❶). IOTailor splits w2 into two requests (w2-1 and w2-2) because it crosses the superpage boundary and enqueues them (❷). Since w1 and w2-1 forms a superpage, they are sent to I/O scheduler. I/O scheduler merges the two into a single write command and dispatch the associated command to the storage device (❸). On enqueuing w3 (❹), since w2-2 and w3 form another I/O group, they are merged and sent to device (❺).

In a per-zone queue, IOTailor waits for up to five seconds to group the split requests with subsequent requests. Hence the split requests may wait in the queue for up to five seconds if there is no subsequent requests. However, this waiting time does not directly manifest as latency. IOTailor immediately issues the split requests without waiting for subsequent requests if they are latency-critical, e.g. issued by the flusher thread or written with direct I/O mode (written with `O_DIRECT` flag).

We develop IOTailor as a separate layer between the filesystem and the block I/O layer instead of integrating its function with a specific I/O scheduler. This design choice allows IOTailor to be used by various legacy I/O schedulers available in the block device layer. An I/O scheduler may blindly merge the requests and may create the new request that is not aligned with the superpage boundary. To prohibit the I/O scheduler from creating the unaligned request, IOTailor places a `no_merge` flag at the first and the last requests of an I/O group.

6.2 Budget-based In-Place Update

To efficiently service tiny synchronous file updates, we develop a technique called *budget-based in-place update*. It allows the underlying zoned UFS to service the in-place file update of F2FS. To facilitate the in-place update, we streamline the application, file system, and the storage device. We define a new flag, `in-place`, for the file and a new tag, `overwrite`, for I/O request, respectively. The application is changed to set the `in-place` flag for a file through `fcntl()` to specify that the file needs be updated in-place. When the application calls `fsync()` on the file with the `in-place` flag, F2FS examines the amount of update to be flushed. If the amount of updated data blocks to be flushed is small, e.g. less than 32 KiB, F2FS places the `overwrite` tag to the I/O request.

Zoned UFS device allocates a SLC region consisting of SLC blocks to accommodate the incoming in-place update request (the small write with `overwrite` tag). Data blocks stored in this region is managed by page granularity mapping.

We introduce the notion of *IPU budget*, or budget for short. A budget is the maximum amount of valid data blocks in the SLC region. On receiving requests with the `overwrite` tag, zoned UFS first checks if it can accommodate the in-place update without exceeding the IPU budget. If it exceeds the budget, the requests cannot be serviced and the device returns an error. In case of the budget overflow error, F2FS retries with append logging mode. F2FS stops requesting IPU and falls back to the append logging mode if the total amount of valid IPU data, outstanding IPU requests and the current IPU request is greater than IPU budget. The device reports the valid IPU data size and IPU budget to the host through SMART [1] interface. We define a notion of *dirty zone*, one or more of whose blocks are written in IPU mode. F2FS keeps track of the list of dirty zones. If the valid IPU data size reaches the IPU budget, F2FS performs garbage collection in background for the dirty zones to decrease the valid IPU data size.

The device keeps track of the amount of valid IPU data by checking the location of the data to be updated in-place utilizing the L2P mapping. The data blocks of the `overwrite` requests can exist either in the SLC block or in the TLC block. If the blocks to be updated in place are present in a TLC block, the blocks are new IPU data (not updated in place after being written to its zone), so the device increments the valid IPU data size. If the blocks are present in a SLC block, the blocks are already accounted for the valid IPU data, so the device does not increment the valid IPU data size.

[1] Self-Monitoring Analysis and Reporting Technology

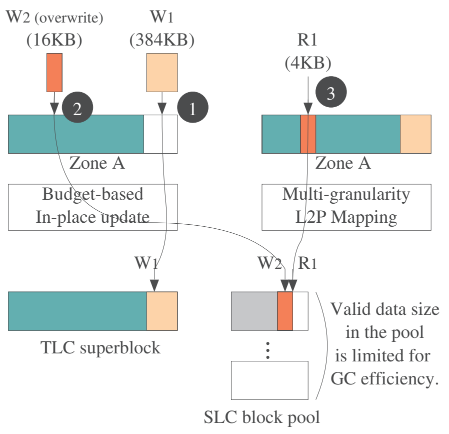


Figure 9. Budget-based in-place update support

When there is a lack of free blocks in the SLC region, the zoned UFS needs to perform garbage collection for the SLC region. Garbage collection is executed in the background to minimize the interference with the foreground I/O service. Mobile device is known to have substantial amount of idle time [58]. So, we carefully suspect that the garbage collection does not interfere with foreground I/O. In certain workloads that do not have enough idle time, garbage collection might interfere with the foreground I/O. The cost of garbage collection decreases as the proportion of valid data in erase blocks decreases [59, 60]. For the efficiency of garbage collection, we limit the amount of valid IPU data as *budget* and reserves sufficient amount of free SLC blocks.

Figure 9 illustrates the operation of in-place update. Assume that F2FS writes the superpage size data to the zone A (W1). W1 is written to the TLC superblock corresponding to the zone A (➊). Then, F2FS writes 16 KiB data with overwrite tag (W2). The device writes W2 to an SLC block (➋) if there exists IPU budget left. The associated L2P mapping is updated and the data is retrieved from the SLC block (➌).

6.3 Copy Offloading

ZMS employs copy offloading scheme to reduce F2FS garbage collection cost. F2FS garbage collection consists of five phases: (1) load metadata, (2) load inode blocks, (3) load node blocks, (4) load valid data pages of a victim segment, and (5) write the data to a target segment. After all the phases are done, F2FS updates the metadata and creates a new checkpoint. Among these phases, the phase 4 and 5 that copy the valid data are the major bottleneck points. By offloading the data copy to the storage device, we eliminate the host side I/O command processing and data transfer between the host and the device. We elaborate our implementation of internal zone compaction and modifications of F2FS to offload data copy in the following.

Zone Copy and Compaction Command. Similar to internal zone compaction (IZC) of ZNS+ [61], we implement zone copy and compaction (ZCC) command in our zoned UFS device. Whereas IZC issues a copy offload request per 2 MiB F2FS segment, ZCC issues a copy offload request per zone unit. This reduces the command transfer overhead. IZC is developed for the NVMe SSDs and ZCC is developed for the UFS (SCSI). Since they are designed for the different platforms, server vs. mobile device, it is difficult to make direct apple-to-apple comparison between their command transfer times. As an indirect comparison, we compare the number of copy offload commands required to process a zone. The zone size of the device used in our work is 138 MiB. For the zone size, ZCC requires only one command to request copy offloading, whereas IZC needs to send up to 69 commands.

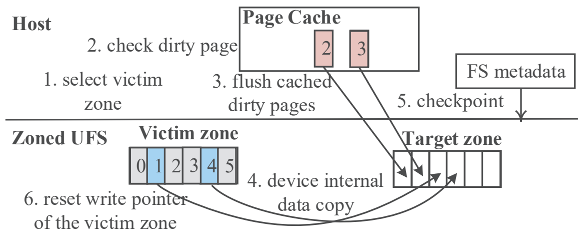


Figure 10. F2FS garbage collection with copy offloading. The boxes with color codes denote pages of the victim zone (grey: invalid, red: dirty cached, blue: clean and not cached).

F2FS Changes for Copy Offloading. We modify F2FS's foreground garbage collection to offload zone compaction. F2FS foreground garbage collection first selects a victim section, identifies and adds its valid blocks to a zone copy (ZC) list. Including dirty pages of the victim section to the ZC list will incur copy of blocks that will soon be invalidated when the dirty pages are written back. So, we add only clean pages to the ZC list. After the ZC list is prepared, F2FS writes back the dirty pages, sends a ZCC command, and waits for the command to be complete. The operation of the modified F2FS garbage collection is illustrated in Figure 10.

6.4 Crash Recovery

Crash recovery of ZMS consists of three stages: device level recovery, file system integrity checking, and file system recovery. On a system crash, the device identifies which zones were open at the time of the crash. Then it scans the superblocks of the open zones sequentially from their first page until it encounters clean (not programmed) page and recovers the write pointers. After the device level recovery, the write pointers are recorded in the zone descriptors. Then, `fsck`, a file system integrity checker, reads the zone descriptors from the device to check if zone write pointers maintained by the file system match those reported by the device, then marks a flag if file system recovery is required at mount time. At mount time, F2FS recovers the write pointers of open zones, scans each active log, performs roll-forward recovery if necessary, then creates a checkpoint. After the recovery, F2FS continues logging beyond the recovered write pointer. The overall recovery procedure of ZMS takes slightly longer than conventional I/O stack because zoned UFS has to scan more superblocks and F2FS has to recover the write pointers. However, sudden power failure happens rarely in modern mobile phones because battery is non-removable.

7 Performance Evaluation

7.1 Experiment Setup

We implement ZMS on Qualcomm SM8350 board, running Android 11 with Linux 5.4 kernel. We implement the zone interface on a commercial TLC-based 128 GiB UFS product by revising the original firmware of the device. The baseline system runs stock F2FS on the UFS device with legacy block interface.

Benchmarks. Table 2 summarizes the workload that are used in our study. As microbenchmarks, we use flexible I/O tester (FIO) [62]. We use 512 KiB and 4 KiB I/O requests for sequential and random tests, respectively. To measure SQLite performance, we use mobibench [63, 64]. We test SQLite transaction processing throughput for both rollback journal and write-ahead log mode. We select 37 applications according to the categories shown in Table 2 and launch them in sequence. We repeat the launch sequence 50 times to get average launch time. The application launching test is done for both clean and aged condition to see the performance impact of file system fragmentation.

Table 2. Benchmarks, R: read, W: write, buf.: buffered, sync.: synchronous, seq: sequential, rand: random

workload	configurations
seq R/W	Fio, 512 KiB IO size
buf rand R/W	Fio, 4 KiB IO over 1 GiB file
sync rand W	Fio, 4 KiB write followed by `fsync()`
range rand R	Fio, 4 KiB read over 8 GiB file
IOTailor test	three concurrent Fio writing jobs, each writing to its own files
mobibench	DELETE mode and WAL mode, 1M `insert()`, 3.9 MiB WAL file, 385 MiB database file
appl'n launch	category (number of apps): basic (8), image (3), video (5), education (4), game (17)

7.2 Performance in Clean Condition

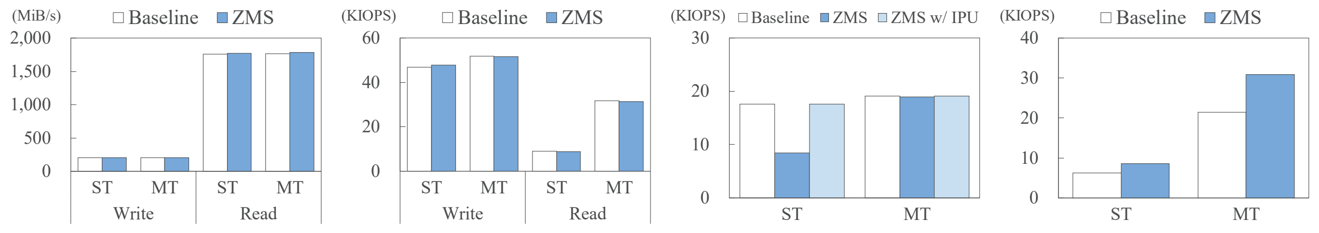
In clean condition, legacy UFS and zoned UFS show similar performance for the basic operations (sequential read/write and 1 GiB range random read/write). Figure 11 illustrates the results. In clean condition, garbage collection (GC) does not run in the filesystem nor in the device. For the 1 GiB range random read, the baseline and zoned UFS show similar performance. This is because entire L2P mappings for the file can be kept in on-chip memory. As the range of random read is extended to 8 GiB, L2P mappings for the file does not fit into the device memory in the baseline, and the fraction of the mapping table is loaded on demand. In the experiment, the baseline experiences L2P map miss for 27.1% of requests whereas there is no map miss in zoned UFS. As a result, ZMS shows 37-44% better random read performance than the baseline.

In the random write with `fsync()` test, each 4 KiB random write is followed by `fsync()`. ZMS shows 50% less throughput in the single thread test compared to the baseline. The reason is that the baseline uses in-place update (IPU) mode whereas zoned UFS does not allow it. Subsequently, F2FS should write the updated node blocks to record new data locations. In this experiment, the node block update accounts for approximately half of the blocks written at the storage device in zoned UFS. On the other hand, in the baseline, node blocks are not updated and therefore are not written to the storage device. It should be noted that the IPU mode does not guarantee data recovery from crash. We observe that budget-based in-place update successfully eliminate the overhead of node block updates. When using the budget-based in-place update (ZMS w/ IPU), zoned UFS renders the same performance as the baseline.

7.3 Performance in Aged Condition

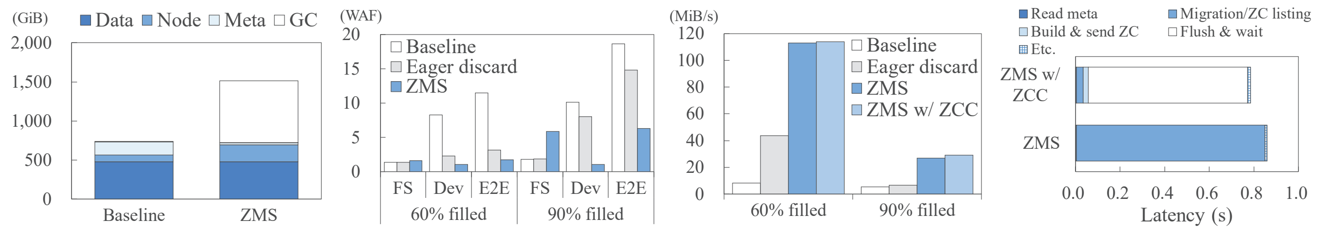
To investigate the performance in aged condition when device and/or file system perform garbage collection, we create a large file and repeatedly overwrite 4 KiB blocks randomly over the file. The file size is 64 GiB for 60% volume utilization and 100 GiB for 90% volume utilization, respectively. Figure 12a shows the amount of block I/O performed at 90% volume utilization. ZMS writes 2× more blocks to the device. The size of total data written is 1500 GiB for ZMS whereas it is 750 GiB for the baseline. This is because F2FS in the baseline performs threaded logging whereas F2FS in ZMS performs append logging. The proportion of F2FS garbage collection in ZMS is about half of the total I/O whereas it is negligible (0.04%) in the baseline. F2FS in ZMS writes checkpoints and more node blocks due to garbage collection, which accounts for the more node writes than the baseline.

F2FS in the baseline writes more metadata than F2FS in ZMS. This is because it writes more segment summary area (SSA) blocks. SSA is stored in the F2FS metadata area and records the owner inode of each data block of a segment. When a data block is written, corresponding SSA record is also updated accordingly. In serving the same amount of user data writes, threaded logging writes to more segments than append logging because threaded logging writes to dirty segments that have less number of free blocks than clean segments. So, the baseline writes more SSA blocks than ZMS. This accounts for the increased metadata writes in the baseline.



(a) Sequential I/O (b) Random I/O (c) Random write w/ `fsync()` (d) Wide range random read

Figure 11. Microbenchmark performance in clean condition (ST: single thread, MT: multi-thread, 4 threads in MT)



(a) I/O volume (b) WAF vs. space utilization (c) Throughput (d) GC latency breakdown

Figure 12. Performance in aged condition

Write amplification of a layer is the amount of data written to its underlying layer divided by the amount of data written to the layer. Figure 12b shows write amplification factor (WAF) of each layer and end-to-end (E2E) WAF. At 60% volume utilization, the baseline shows file system WAF of 1.39, device WAF of 8.27, and E2E WAF of 11.5. In ZMS, file system WAF is 18% higher than that of the baseline while device WAF is close to 1.0 because device garbage collection is eliminated. Despite the increased file system WAF, E2E WAF of ZMS is 1.72 which is 6.7× lower than that of the baseline. At 90% utilization, the file system WAF of ZMS is increased to 5.89 due to more garbage collection. However, ZMS still shows 2.96× lower E2E WAF than the baseline. Owing to the reduced E2E WAF, ZMS shows 13.6× and 5.1× higher throughput than the baseline for 60% and 90% utilization, respectively (Figure 12c).

The Effect of Eager Discard. The gap of information about block validity contributes to device level write amplification (§2.2). F2FS shuns discard commands while I/O is busy. To reduce copying invalid data at device level, we modify F2FS to send discard commands when writing a new checkpoint. With the eager discard mechanism, the device WAF of the baseline is reduced by 72% and the performance is improved by 5× at 60% utilization. However, ZMS still performs 2.5× better than the baseline with the eager discard mechanism.

Table 3. Tail latency (μsec) of random write in aged condition

percentile	baseline	ZMS w/o ZCC	ZMS w/ ZCC
average	684	160	133
99th	15,795	34	16
99.9th	20,055	103	5,407
99.99th	31,851	9,765	10,421

The Effect of Copy Offloading. When using copy offloading, the performance is enhanced further by 8% for 90% utilization, which was due to faster F2FS garbage collection completion by offloading data copy to device. Since writing dirty pages are blocked until foreground garbage collection is complete, reducing the garbage collection latency improves write throughput. F2FS garbage collection latency is reduced by 8.7% by using ZCC as shown in Figure 12d. ZCC improves throughput by 8% as shown in Figure 12c. In addition, tail latencies of random writes are reduced by 12–25% for 90% utilization by using ZCC as shown in Table 3.

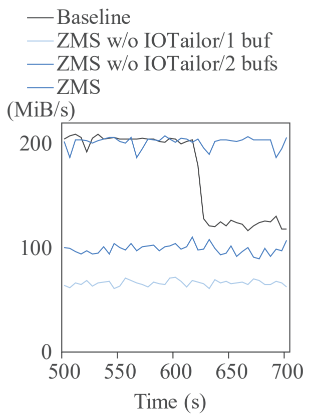
7.4 Writing to Multiple Open Zones

When the number of open zones being written concurrently exceeds the number of write buffers of the zoned UFS device, unaligned buffer flush takes place. The unaligned buffer flush has negative impact on performance and lifetime of the device. To examine the performance impact of the unaligned buffer flush in mobile usage scenario and the benefit of IOTailor, we devise a synthetic benchmark that simulates a practical Android storage access pattern [65–67]. We run hot, warm, and cold data streams, configured as follows.

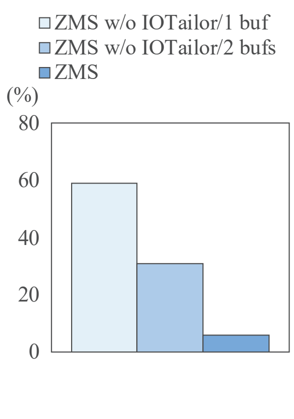
- hot: creates 2000 6 MiB files at initialization, then overwrites randomly selected files.
- warm: creates 2000 12 MiB files at initialization, then overwrites randomly selected files.
- cold: at each iteration, creates 4500 4 MiB files and deletes 900 randomly selected files.

The benchmark grows the amount of cold files at each iteration, and stops if no more space is available. We vary the number of write buffers (one or two) available in the device.

The test results are shown in Figure 13. In the baseline,



(a) Bandwidth



(b) SLC buffering ratio

Figure 13. Performance benefit of IOTailor

the device renders maximum write performance during the earlier phase of the experiment. However, its performance drops sharply when device garbage collection kicks in at around 620 seconds. Figure 13b indicates the ratio of data written to SLC blocks, due to the unaligned buffer flush, to the total data written by the host. The portion of data written to SLC blocks is 60% for the single buffer configuration (1 buf) and 30% for the two buffer configuration (2 bufs), respectively. Consequentially, ZMS without IOTailor shows worse performance than the baseline as shown in Figure 13a. With IOTailor, the SLC buffering ratio is reduced significantly to 5% and the performance is consistently better than the other configurations. ZMS with IOTailor still has the small SLC buffering ratio because the amount of node blocks written by F2FS is smaller than the TLC programming unit in most cases, so the node blocks are written to SLC blocks.

Table 4. Tail latency (μsec) of random writes (queue depth: 1)

percentile	w/o IOTailor	w/ IOTailor
average	62	62
99.99th	293	281
99.999th	5,014	4,948
99.9999th	7,635	8,586

Computational Overhead of IOTailor. To examine the latency increase due to computational overhead of IOTailor, we issue 4 KiB random writes with direct I/O mode to a 10 GiB file. In the direct I/O mode, only a single request is outstanding. As shown in Table 4, IOTailor does not show noticeable latency increase except for the 6-nine latency. The two-nine and the three-nine latencies are not shown here because those latencies with IOTailor are the same as those without IOTailor. Given the significant throughput improvement, this level of latency increase is acceptable.

7.5 Application Level Performance

7.5.1 SQLite Database Throughput

SQLite heavily issues small `fsync()` calls. So, the latency of `fsync()` seriously affects SQLite processing performance. Since zone abstraction does not allow in-place update, it incurs more node writes due to using append logging, which hampers the SQLite performance. As shown in Figure 14a, ZMS shows 35% less SQLite transaction throughput when using rollback journal with delete mode. The performance gap between ZMS and the baseline becomes larger in truncate mode and persist mode. The proposed budget-based in-place update eliminates the node writes by allowing the in-place update "conditionally". In-place update is allowed only if the amount of total in-place update data is within the IPU budget. This is to restrict the cost of garbage collection in the SLC region that is used for writing the in-place updated data. We observe that ZMS is able to achieve equal performance to the baseline by using the budget-based in-place update for the rollback journal mode.

In write-ahead logging (WAL) mode, since node modifications of multiple transactions are merged to a large sequential I/O, writing nodes for the database file is more efficient than in rollback journal mode. Therefore, despite increased node writes, ZMS shows 60–100% better performance in WAL mode because ZMS performs large sequential writes for data and nodes whereas the baseline issues random writes in updating the database file.

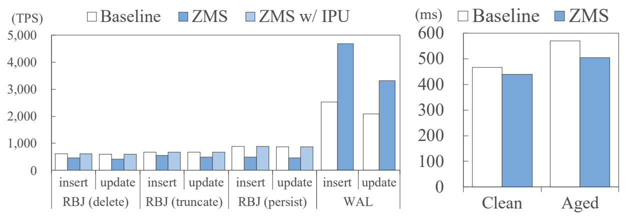
7.5.2 Application Launch Time

From 50 rounds of launching the 37 applications in sequence, we calculate the average launch time. The test is done in both clean and aged conditions. As shown in Figure 14b, ZMS and the baseline show the average launch times of 440ms and 467ms in clean condition, respectively. The I/O pattern consists of 67.8% random reads over 8 GiB range, and 61.1% of total I/Os is smaller than 32 KiB. Thanks to the multi-granularity L2P mapping, zoned UFS efficiently serves random reads, which in turn enhance application launch time.

A larger performance gap is observed in aged condition, when application files are fragmented over a 90 GiB address range. To simulate file fragmentation, we install applications while running a background FIO process requesting synchronous random writes on a 75 GiB file. The baseline suffers from wide range random reads due to more L2P map loadings, which degrades random read throughput.

8 Related Works

Zoned Namespace Storage Device. There have been growing interests towards ZNS [1–3, 46, 47, 61], but most of these works were targeted for server SSDs. Shin et al. developed



(a) SQLite `insert()`, mobibench (b) Application launch time

Figure 14. Application level performance

an analysis tool to analyze the performance, parallelism, isolation, and predictability of ZNS SSD [68]. Zonefs is a file system that exposes each zone of a zoned block device as a file to user space [69]. Btrfs and F2FS have native support for zoned block devices [70, 71]. Choi et al. proposed an LSM based garbage collection technique to mitigate the long latency that occurs when performing garbage collection for large zones [72]. Chung outlined the expected benefits of using ZNS in datacenter [73]. In [3], Matias et al. showed that RocksDB can get performance benefit from ZNS devices by aligning zones with SSTable files, thereby eliminating SSD level garbage collection. ZNS+ [61] proposed an interface to allow F2FS threaded logging for a zone. ZNS+ also proposed internal zone compaction (IZC). Our zone copy and compaction (ZCC) requires less number of commands than IZC. When performing garbage collection for a zone, IZC requires the host to issue a separate command per segment (2 MiB). In ZMS, a single ZCC command is necessary to offload copy for the entire data of a zone.

Alternative Device Abstraction. Like the zone interface, to reduce write amplification and ensure predictability, open-channel interface [16] moves many of FTL responsibilities to the host, including explicit data placement. Picoli et al. reported that the open-channel approach is not for all situations [74] because getting everything right (i.e., dealing with NAND geometries, fine-tuning OS and applications) is quite complex to do only at the host side. Compared with the open-channel interface, the zone interface pursues a better role partition between the host and the device, relieving the host of managing flash memory errors and wear leveling. To reduce write amplification, flexible data placement has been proposed [75]. Write amplification varies depending on the use cases of the host systems, and calls for more research to adapt the I/O subsystems and file systems to each workload.

L2P Mapping. L2P mapping schemes can be classified into three groups subject to mapping granularity: block mapping, page mapping and hybrid mapping. Block mapping [76] reduces L2P mapping size significantly by coarse-grained mapping. However, it suffers from high garbage collection overhead when modifying data of a block. Page mapping [39–45] has high efficiency of garbage collection, but its L2P mapping size is too large to fit into the internal memory of mobile storage device. In hybrid mapping [77–81], incoming writes are first placed at the log blocks then later are merged with the data blocks. Utilizing the log-based write operation, it shows better garbage collection efficiency than block mapping. Hybrid mapping reduces the L2P mapping size utilizing both page mapping and block mapping; log blocks are managed by page mapping and data blocks are managed by block mapping. ZMS differs from the conventional hybrid mapping techniques in that it uses chunk mapping (4 MiB granularity) for regions where pages are contiguously written and does not require log block merge operation. Jeong et al. proposed host performance booster (HPB) that uses part of the host memory as a cache of L2P mapping table to improve the random read performance of mobile storage [23]. Kim et al. proposed to manage HPB memory considering application status (foreground or background) and to resize the HPB memory under memory pressure of the smartphone to improve responsiveness [24].

Data Separation. Previous studies [82, 83] analyzed Android file access patterns and revealed that there is a stark difference in file lifetimes depending on file types. For example, [83] reported that 47% of files are deleted within 30 days after being created, files generated by applications are more likely to be deleted than user-generated media files, and video files have longer lifetimes than images. Attempts to enhance file lifetime prediction [4, 8, 12] are complementary to our work and can be employed on ZMS to better separate data to different zones.

9 Conclusions

In designing a new I/O stack based on zone abstraction for mobile flash storage, called ZMS, we identify two challenges: write buffer thrashing and tiny synchronous file update. By implementing IOTailor and the budget-based in-place update mechanism to address the challenges, we show that ZMS improves performance and write amplification compared to conventional block-based I/O stack. As further work, we plan to explore utilizing heterogeneous zones with different cell types to place the data blocks according to the performance and lifespan requirements. We are also likely to investigate the feasibility of providing more temperature levels in F2FS to further mitigate its garbage collection overhead.

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions. We express sincere gratitude to Yong Ho Song and Hwaseok Oh for their unwavering support throughout this research endeavor. We would like to extend our appreciation to Kate Mun, Jinhwan Park, and other members of the research team who contributed to this work. The work of Youjip Won is in part funded by Samsung Electronics.

References

[1] Matias Bjørling. From open-channel ssds to zoned namespaces. In *Linux Storage and Filesystems Conference (Vault 19)*, page 1, 2019.

[2] J. Gonzalez. Zoned namespaces: Standardization and linux ecosystem. In *SDC EMEA*, 2020.

[3] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. {ZNS}: Avoiding the block interface tax for flash-based {SSDs}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.

[4] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in {Log-Structured} storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, 2022.

[5] Yao-Jen Hsu, Chin-Hsien Wu, Yu-Chieh Tsai, and Chia-Cheng Liu. A granularity-based clustering method for reducing write amplification in solid-state drives. *ACM Transactions on Embedded Computing Systems*, 2023.

[6] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, 2013.

[7] Janki Bhimani, Jingpei Yang, Zhengyu Yang, Ningfang Mi, NHV Krishna Giri, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. Enhancing ssds with multi-stream: What? why? how? In *Proceedings of 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–2. IEEE, 2017.

[8] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, 2021.

[9] Trong-Dat Nguyen and Sang-Won Lee. I/o characteristics of mongodb and trim-based optimization in flash ssds. In *Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory*, pages 139–144, 2016.

[10] You Zhou, Ke Wang, Fei Wu, Changsheng Xie, and Hao Lv. Seer-ssd: Bridging semantic gap between log-structured file systems and ssds to reduce ssd write amplification. In *Proceedings of 2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 49–56. IEEE, 2021.

[11] Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H Noh. To trim or not to trim: Judicious trimming for solid state drives. In *Poster presentation in the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[12] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. Write amplification reduction in {Flash-Based}{SSDs} through {Extent-Based} temperature identification. In *Proceedings of 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[13] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for {Multi-Streamed}{SSDs} using program contexts. In *Proceedings of 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 295–308, 2019.

[14] Taejin Kim, Sangwook Shane Hahn, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. {PCStream}: Automatic stream allocation using program contexts. In *Proceedings of 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[15] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: random write considered harmful in solid state drives. In *Proceedings of USENIX Conferece on File and Storage Technologies (FAST'12)*, volume 12, pages 1–16, 2012.

[16] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. {LightNVM}: The linux {Open-Channel}{SSD} subsystem. In *Proceedings of 15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, 2017.

[17] Ohhoon Kwon, Kern Koh, Jaewoo Lee, and Hyokyung Bahn. Fegc: An efficient garbage collection scheme for flash memory based storage systems. *Journal of Systems and Software*, 84(9):1507–1523, 2011.

[18] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. Autostream: automatic stream management for multi-streamed ssds. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–11, 2017.

[19] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Fstream: managing

flash streams in the file system. In *Proceedings of 16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 257–264, 2018.

[20] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[21] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *Proceedings of 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.

[22] Kisung Lee and Youjip Won. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 23–32, 2012.

[23] Wookhan Jeong, Hyunsoo Cho, Yongmyung Lee, Jaegyu Lee, Songho Yoon, Jooyoung Hwang, and Donggi Lee. Improving flash storage performance by caching address mapping table in host memory. In *Proceedings of 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[24] Yoona Kim, Inhyuk Choi, Juhyung Park, Jaeheon Lee, Sungjin Lee, and Jihong Kim. Integrated {Host-SSD} mapping table management for improving user experience of smartphones. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 441–456, 2023.

[25] Kyusik Kim, Eunji Lee, and Taeseok Kim. Hmb-ssd: Framework for efficient exploiting of the host memory buffer in the nvme ssd. *IEEE Access*, 7:150403–150411, 2019.

[26] Kyusik Kim and Taeseok Kim. Hmb in dram-less nvme ssds: Their usage and effects on performance. *PloS one*, 15(3):e0229645, 2020.

[27] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, page 1, 2012.

[28] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaftl: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 442–456, 2023.

[29] Shengzhe Wang, Zihang Lin, Suzhen Wu, Hong Jiang, Jie Zhang, and Bo Mao. Learnedftl: A learning-based page-level ftl for reducing double reads in flash-based ssds. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 616–629. IEEE, 2024.

[30] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. File fragmentation in mobile devices: Measurement, evaluation, and treatment. *IEEE Transactions on Mobile Computing*, 18(9):2062–2076, 2019.

[31] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, 2017.

[32] Jonggyu Park and Young Ik Eom. Filesystem fragmentation on modern storage systems. *ACM Transactions on Computer Systems*, 41(1-4):1–27, 2023.

[33] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4):1–25, 2012.

[34] Cheng Ji, Li-Pin Chang, Riwei Pan, Chao Wu, Congming Gao, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Pattern-Guided file compression with User-Experience enhancement for Log-Structured file system on mobile devices. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 127–140. USENIX Association, February 2021.

[35] JEDEC Standard. Zoned storage for universal flash storage (ufs). *JESD220-5*, 2022.

[36] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/o stack optimization for smartphones. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC 13)*, pages 309–320, 2013.

[37] Theano Stavrinos, Daniel S Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don't be a blockhead: zoned namespaces make work on conventional ssds obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 144–151, 2021.

[38] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *Proceedings of 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[39] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. *Acm Sigplan Notices*, 44(3):229–240, 2009.

[40] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. *ACM SIGOPS Operating Systems Review*, 28(5):86–97, 1994.

[41] Han-Joon Kim and Sang-goo Lee. A new flash memory management for flash storage system. In *Proceedings. Twenty-Third Annual International Computer Software and Applications Conference (Cat. No. 99CB37032)*, pages 284–289. IEEE, 1999.

[42] M-L Chiang and R-C Chang. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Software*, 48(3):213–231, 1999.

[43] Fan Ni, Chunyi Liu, Yang Wang, Chengzhong Xu, Xiao Zhang, and Song Jiang. A hash-based space-efficient page-level ftl for large-capacity ssds. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6. IEEE, 2017.

[44] Youngjae Kim, Aayush Gupta, and Bhuvan Urgaonkar. A temporal locality-aware page-mapped flash translation layer. *Journal of Computer Science and Technology*, 28(6):1025–1044, 2013.

[45] Zhiguang Chen, Nong Xiao, Fang Liu, and Yimo Du. Hot data-aware ftl based on page-level address mapping. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 713–718. IEEE, 2010.

[46] David L. Black. The nvme standard: The next five years. *NVMe Developer Days*, 2018.

[47] Matias Bjørling. Zoned namespaces in practice. In *Flash memory summit*, 2019.

[48] Xiangqun Zhang, Shuyi Pei, Jongmoo Choi, and Bryan S Kim. Excessive ssd-internal parallelism considered harmful. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, pages 65–72, 2023.

[49] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, 2012.

[50] Jeong-Uk Kang, Jin-Soo Kim, Chanik Park, Hyoungjun Park, and Joonwon Lee. A multi-channel architecture for high-performance nand flash-based storage system. *Journal of systems Architecture*, 53(9):644–658, 2007.

[51] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *Proceedings of 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 521–526, 2012.

[52] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–33, 2009.

[53] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[54] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery. In *Proceedings of 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 438–449, 2015.

[55] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An empirical study of file-system fragmentation in mobile storage systems. In *Proceedings of 8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[56] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). https://www.t10.org/, 2014.

[57] Jinglei Ren, Chieh-Jan Mike Liang, Yongwei Wu, and Thomas Moscibroda. Memory-Centric data storage for mobile systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 599–611, Santa Clara, CA, July 2015. USENIX Association.

[58] Daniel Hintze, Philipp Hintze, Rainhard D Findling, and René Mayrhofer. A large-scale, long-term analysis of mobile device usage characteristics. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(2):1–21, 2017.

[59] Rajiv Agarwal and Marcus Marrow. A closed-form expression for write amplification in nand flash. In *2010 IEEE Globecom Workshops*, pages 1846–1850, 2010.

[60] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009*, pages 1–9, 2009.

[61] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *Proceedings of 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 147–162, 2021.

[62] fio - Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[63] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. Androstep: Android storage performance analysis tool. *Software Engineering 2013-Workshopband*, 2013.

[64] mobibench. https://github.com/ESOS-Lab/Mobibench.

[65] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment*, 8(12):1454–1465, 2015.

[66] Ashish Bijlani, Umakishore Ramachandran, and Roy Campbell. Where did my 256 gb go? a measurement analysis of storage consumption on smart mobile devices. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(2), jun 2021.

[67] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Chun Jason Xue. Inspection and characterization of app file usage in mobile devices. *ACM Trans. Storage*, 16(4), sep 2020.

[68] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. Exploring performance characteristics of zns ssds: Observation and implication. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–5. IEEE, 2020.

[69] Damien Le Moal and Ting Yao. zonefs: Mapping posix file system interface to raw zoned block device accesses. 2020.

[70] Naohiro Aota. btrfs: zoned block device support. https://lore.kernel.org/linux-btrfs/cover.1612433345.git.naohiro.aota@wdc.com/.

[71] Damien Le Moal. f2fs: Zoned block device support. https://lore.kernel.org/linux-f2fs-devel/1477644307-30115-1-git-send-email-damien.lemoal@wdc.com/.

[72] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. A new lsm-style garbage collection scheme for zns ssds. In *Proceedings of 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[73] W Chung. Benefits of zns in datacenter storage systems. In *Flash memory summit*, 2019.

[74] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-channel ssd (what is it good for). In *CIDR*, 2020.

[75] NVM Express Workgroup. TP4146 Flexible Data Placement 2022.11.30 Ratified. https://nvmexpress.org/specifications/, 2022.

[76] Amir Ban. Flash file system. *United States Patent, no. 5,404,485*, 1995.

[77] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, 2006.

[78] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18–es, 2007.

[79] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. Last: locality-aware sector translation for nand flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, 2008.

[80] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):1–23, 2008.

[81] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[82] Yunji Kang and Dongkun Shin. mstream: stream management for mobile file system using android file contexts. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1203–1208, 2021.

[83] Roy Friedman and David Sainz. File system usage in android mobile phones. In *Proceedings of the 9th ACM International Systems and Storage Conference*, pages 1–11, 2016.