D2FS: Device-Driven Filesystem Garbage Collection

Juwon Kim and Seungjae Lee, *Korea Advanced Institute of Science and Technology (KAIST); * Joontaek Oh, *University of Wisconsin–Madison;* Dongkun Shin, *Sungkyunkwan University;* Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*

https://www.usenix.org/conference/fast25/presentation/kim-juwon

This paper is included in the Proceedings of the 23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

D2FS: Device-Driven Filesystem Garbage Collection

Juwon Kim† Seungjae Lee† Joontaek Oh*‡ Dongkun Shin§ Youjip Won†
†*KAIST* ‡*University of Wisconsin–Madison* §*Sungkyunkwan University*

Abstract

In this work, we propose a mechanism to free the log-structured filesystem from running the garbage collection. We exploit the garbage collection functionality of the underlying flash storage to reclaim the invalid sections in the filesystem partition. We call it a *Log-structured Filesystem with Device-Driven Garbage Collection, D2FS*. D2FS consists of three key ingredients: Coupled Garbage Collection, Migration Upcall, and Virtual Overprovisioning. Coupled Garbage Collection consolidates the valid flash pages at the storage device and remaps the migrated flash pages to new filesystem locations so that the valid pages are clustered not only physically but also logically. Migration Upcall asynchronously notifies the host about the file mappings updated by the Coupled Garbage Collection, minimizing interference with the foreground filesystem operations. Virtual Overprovisioning separates the size of the filesystem partition from the physical capacity of the associated storage partition and sets the size of the filesystem partition larger than the physical storage partition. Virtual overprovisioning ensures that FTL runs the device-level garbage collection on time so that the filesystem partition never runs out of free sections. By integrating these techniques, we save the log-structured filesystem from the garbage collection overhead, a primary obstacle hindering its widespread adoption in production environments. D2FS outperforms F2FS by 3× (FIO), zoned F2FS by 1.7× (FIO), and IPLFS by 1.5× (MySQL YCSB-F).

1 Introduction and Summary

In many storage media, e.g. Flash, Shingled Magnetic Recording (SMR), and even DRAM, placing data consecutively is much faster than placing each of them apart. Aligned with this characteristic, numerous studies have proposed storing the data in a log-structured manner. They include log-structured filesystem [45,46,64,66] and log-structured merge-tree based key-value store [19,20,61,78,81].

Despite its promising characteristics, the log-structured filesystem suffers from critical drawback: the overhead of reclaiming the invalid filesystem blocks, the *garbage collection overhead*. The garbage collection involves suspending incoming write requests [5], and updating and checkpointing the filesystem state [3], rendering a significant performance drop and an excessive tail latency.

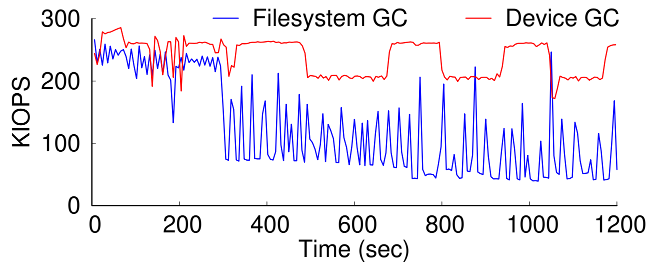


Figure 1: Performance Overhead of Garbage Collection; filesystem-level garbage collection vs. device-level garbage collection (Samsung 970 Pro [74])

Numerous works were proposed to mitigate the garbage collection overhead. They include performing the garbage collection in a pre-emptible manner [82] or during idle periods [18,54], clustering the filesystem blocks with similar lifespans in the same section[1] [46,56,75], offloading consolidation of valid filesystem blocks to the storage [28], or exporting an excessively large filesystem partition to avoid running out of free sections [40]. All of these works have pros and cons and leave substantial room for improvement. Fully pre-emptible garbage collection is practically impossible [42]. Predicting the filesystem idle period is error-prone [18,59]. Exporting a very large filesystem partition [40] requires the FTL to handle prohibitively large LBA space, leading to excessive memory pressure and complexity in the L2P mapping design.

Modern flash storage performs device-level garbage collection to reclaim invalid flash pages. When a log-structured filesystem runs on top of flash storage, both the filesystem and the underlying storage perform their own garbage collection routines in an uncoordinated manner, increasing write amplification and degrading the system performance [80]. To mitigate this redundant garbage collection overhead, some works involve performing garbage collection solely on the host [16, 17, 49, 58]. In this approach, the host is fully responsible for reclaiming invalid flash blocks, so device-level garbage collection is eliminated. However, this approach is not free from costly filesystem garbage collection overhead,

*This work was done while the author was a graduate student at KAIST.

[1] In this paper, we refer to the garbage collection unit of a log-structured filesystem as a section.

such as checkpoint overhead, and user request suspension.

We examine the overhead of filesystem-level and device-level garbage collections (Fig. 1). We create 256 GByte storage partition on 512 GByte SSD. For filesystem-level garbage collection test, we format the storage partition with stock F2FS and fill 70% of the partition with a single file. Then, we perform random write [13] on the file. We leave non-partitioned storage space empty. The filesystem runs garbage collection when the storage partition is full. Since the storage device still has free space left, it does not run garbage collection. For device-level garbage collection test, we fill 70% of the 256 GByte storage partition with data. We also fill the non-partitioned storage space with dummy data. We mount the storage partition as a raw device and perform random write on the data in the storage partition. When the device runs out of free flash pages, it triggers garbage collection. We observe that both filesystem-level and device-level garbage collections degrade the application performance (Fig. 1). However, filesystem-level garbage collection has a more drastic impact, lowering benchmark throughput to $\frac{1}{5}$. Device-level garbage collection reduces the performance by up to 20%.

In this work, we revisit the issue of addressing the garbage collection overhead in the log-structured filesystem. The essential idea is simple; *the storage device reclaims the free sections at the filesystem on behalf of the filesystem with minimal interference to the foreground filesystem activity*. We exploit the garbage collection mechanism of the underlying storage in reclaiming the free sections at the log-structured filesystem. Our work distinguishes itself from the earlier works in that we delegate the garbage collection from the filesystem to the storage device, not vice versa.

We need to address three technical challenges in realizing device-driven filesystem-level garbage collection. First is filemap update. The storage device should be able to update the logical block address (LBA) of the consolidated flash page and seamlessly synchronize it with the host. The second is the lightweight device-host interface. The storage device should be able to inform the host about the updated LBA without significant IO communication overhead. Third is the timely activation of the device-level garbage collection. Before the filesystem runs out of the free sections, the device-level garbage collection should timely reclaim the free sections on filesystem. We address these challenges through the following key technical ingredients; Coupled Garbage Collection, Migration Upcall, and Virtual Overprovisioning.

In *Coupled Garbage Collection*, the device-level garbage collection not only consolidates the valid flash pages but also remaps them to the new filesystem location. Coupled Garbage Collection consolidates the valid filesystem blocks without physically copying them to the new locations.

With *Migration Upcall*, the storage device piggybacks the event notification about updated LBAs onto the other IO completion signal sent to the host, thereby avoiding the need for dedicated polling or interrupt to notify LBA updates. Migration Upcall neither consumes host CPU cycles nor requires defining a new interrupt.

In *Virtual Overprovisioning*, we separate the filesystem partition size from the underlying storage device capacity. By virtually expanding the filesystem partition, we ensure that the storage device runs garbage collection and reclaims free sections before the filesystem runs out of free sections.

Combining these, we present D2FS, a log-structured filesystem with device-driven filesystem garbage collection. It is a variant of F2FS. The contributions of our work are as follows.

- We free the host CPU from the overhead of running garbage collection. We achieve this by orchestrating the existing IO stack features, e.g. device garbage collection, filesystem overprovisioning, and the NVMe interface, without introducing significant changes to the IO stack design.
- We discover that by employing a moderate degree of virtual overprovisioning, such as 2.4, the device-level garbage collection can seamlessly supply free sections to the filesystem. This finding highlights that we can eliminate filesystem-level garbage collection without requiring an excessively overprovisioned filesystem partition by leveraging device-driven filesystem garbage collection.
- We show that by piggybacking the notification signal to the other IO completion signal from the device to the host, the storage device can deliver the updated file mapping information to the host without relying on host-centric event detection mechanisms such as interrupt and polling.

Eliminating the filesystem-level garbage collection, D2FS outperforms F2FS by 3×, IPLFS [40] by 1.5× (YCSB-F), and F2FS ported on ZNS SSD [4] by 1.7× (FIO). D2FS reduces device memory pressure by $\frac{1}{14}$ compared to IPLFS.

2 Background

2.1 Flash Storage

Flash storage has three operations; read, write (program) and erase, and it defines three corresponding units; page, block, and superblock. A flash page is the write unit, typically ranging from 4 to 64 KByte. A flash block is the erase unit, consisting of 512 to 1024 pages depending on the SSD design. Modern SSD controllers employ multiple levels of parallelism, including at the channel, way, and chip levels. A superpage and superblock are a group of flash pages and a group of flash blocks that are at the same offset within the chips across different ways and channels [27, 55, 72, 77], respectively. SSD controller uses a superpage and a superblock as a unit of write (program) and a unit of garbage collection to exploit the internal parallelism, respectively.

Flash Translation Layer (FTL) is a software layer within SSD. FTL is responsible for logical-to-physical address (L2P) mapping, garbage collection, and wear-leveling. When there runs out of free flash blocks, FTL performs *device-level* *garbage collection*; it selects a victim block based upon its victim selection policy [36, 52, 70], copies valid pages in the victim block to the destination free block, and then erases the victim block to reclaim it. Often, recent SSD performs device-level garbage collection at superblock granularity [27, 77]. Flash storage defines Discard command (or TRIM command) [69]. Discard command informs the storage about the filesystem blocks that are no longer in use. Discard command prevents FTL from blindly migrating invalid file blocks during device-level garbage collection [8, 16, 40, 63].

2.2 Log-structured Filesystem

The log-structured filesystem is an append-only filesystem initially developed to mitigate the seek overhead in accessing data and associated metadata in HDDs [64, 66]. It partitions the entire filesystem into a sequence of fixed-size units, called *segments*. A segment is a set of filesystem blocks that are allocated and written together to the storage device [46, 64, 66]. When the log-structured filesystem runs out of free segments, it consolidates the valid file blocks to the new segments, and reclaims the old segments. We call this activity *garbage collection*. Early log-structured filesystems [64] use the segment as a unit for both flush and garbage collection. A recent log-structured filesystem for flash storage, F2FS [46], separates the garbage collection unit from the allocation unit (or unit of flush, equivalently). F2FS defines the garbage collection unit as a *section*. A section is a set of one or more segments.

The separation of sections and segments in F2FS is an important and wise design decision. To minimize write amplification in reclaiming the free filesystem space, the filesystem needs to align its garbage collection unit size with the device-level garbage collection unit size. The segment size of F2FS, 2 MByte, is deeply embedded in its design. Changing the 2 MByte segment size in F2FS is practically infeasible since one needs to overhaul the entire filesystem design. The size of the device-level garbage collection unit varies across devices. Instead of blindly using 2 MByte segment as the garbage collection unit, F2FS introduces a new notion, a section. The section size can be configured to match the garbage collection unit size of the underlying device.

The filesystem-level garbage collection is expensive. During the garbage collection, the filesystem acquires an exclusive lock [5], suspends all operations that update the filesystem state, e.g. `write()`, `unlink`, `create()`, etc. Before and after the garbage collection, the filesystem checkpoints its states to the storage. This is to ensure the crash consistency of the garbage collection.

2.3 Addressing Garbage Collection Overhead

We categorize the existing efforts for addressing the garbage collection overhead into two: algorithmic improvement and framework redesign.

Algoritmic Improvement. The most common approach is to cluster the file blocks with similar lifespans at the same section [56]. The lifespan can be estimated by using the file extension [46], by using the hint from user [43, 62], or by using the update frequency [75]. Some works proposed to perform the garbage collection during the idle period [18, 54, 59]. Numerous works proposed to alleviate the device-level garbage collection overhead. They include exploiting the SSD's internal parallelism [68], coordinating a flash array to hide the garbage collection overhead [38, 50], and making the garbage collection pre-emptible [42, 48].

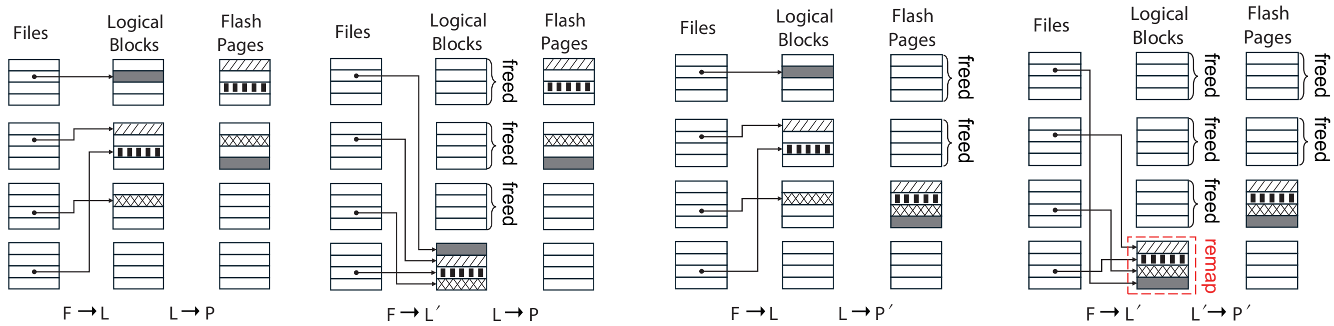
Framework Design. A few studies proposed that the host filesystem directly consolidates the valid flash pages at the storage device, eliminating the device-level garbage collection overhead [17,49,58,82,83]. This requires the storage device to expose its internal geometry to the host, which most vendors are reluctant to do. Another approach is to use append-only block devices like *zoned namespace* (ZNS) SSDs [16, 28, 57]. ZNS SSD organizes its partition as a set of *zone*s. A zone is an array of consecutive flash pages, which can be written only in append-only manner. A zone is invalidated and reclaimed as a single unit. ZNS SSD is free from device-level garbage collection, but it passes the garbage collection to the application, complicating the application design. IPLFS [40] takes the opposite approach; it removes the garbage collection from the filesystem and offloads it to the storage device. IPLFS defines an extremely large, e.g. ZByte, filesystem partition so that the filesystem never runs out of free sections during the SSD lifespan. At the exchange, the underlying flash storage needs to handle prohibitively large LBA space in its L2P mapping.

2.4 Device-Managed Filesystem Partition

In modern Operating System design, the host filesystem is solely responsible for managing the the filesystem partition. Recently, a few works proposed to endow storage device with a limited capability to manage the filesystem partition. In Range Write [11], the filesystem sends a set of LBAs along with the write command to the disk. The disk selects one of the offered LBAs and returns its choice to the host after servicing the write command. In Nameless Write [84], a host dispatches the write command to the storage without specifying a destination LBA. The device selects the location to place the data block, services the write request, and returns the LBA of the selected location to the host.

An NVMe specification standardizes the concept of device-managed filesystem partition [32]. In ZNS interface [16], `zone_append` command [15] specifies only zone id when it writes the data block to the storage. Then, the storage device determines the location of the data block within the designated zone, and returns the offset in a zone to the host. Several works exploit the `zone_append` command to optimize the key-value store [47], RAID [44], and swap subsystem [14].

Our work shares the same philosophy of allowing the de-



(a) Original layout (b) Filesystem-level GC (c) Device-level GC (d) Coupled GC

Figure 2: Filesystem-level garbage collection, Device-level garbage collection and Coupled Garbage Collection

vice to manage the filesystem partition with other works. We allow the storage device to allocate the free blocks at the filesystem partition and to update the filemap.

3 Design Principle

We exploit the garbage collection mechanism of the flash storage to reclaim the free sections at the filesystem partition. Our approach consists of three key technical ingredients: Device-Driven filemap management, Device-Host interface, and Timely garbage collection.

- **Device-Driven Filemap Management.** For the storage device to consolidate the valid filesystem blocks, the device firmware needs to modify the LBA of the physical block and update the associated filemap at the host. The storage device needs to seamlessly update the filemap of the filesystem without interfering with ongoing filesystem activities.
- **Device-Centric IO Interface.** The device needs to inform the host about the updated filesystem state with minimum interference. Legacy IO interface is host-centric. Both interrupt and polling are designed to capture the event predefined by the host. We carefully argue that they do not well suit for informing the host about the device-initiated activity such as device-level garbage collection.
- **Timely Activation of the Garbage Collection.** The storage device needs to run garbage collection on time so that it can provide the free sections to the filesystem without any interruption. We need to align the timing to run the device-level garbage collection with the availability of the free sections in the filesystem partition.

For device-driven filemap management, we develop *Coupled Garbage Collection*. For device-driven IO interface, we develop *Migration Upcall*. For timely synchronization of the garbage collection, we propose *Virtual Overprovisioning*. We will explain these three techniques in the following sections.

4 Coupled Garbage Collection

4.1 A Concept

Coupled Garbage Collection (CGC) is device-level garbage collection that reclaims not only the free blocks at the flash storage but also the free sections at the filesystem. Determining the physical location of a given filesystem block consists of two mapping layers; (i) from a file block ([fd, offset]) to an LBA and (ii) from an LBA to a physical page address in the flash storage (PPA). We call the former and the latter as F2L mapping and L2P mapping, respectively. For F2L mapping, the filesystem navigates through the one or more indirect blocks (or index blocks) to find the LBA of a given file block. The data structure associated with identifying the LBA of a given file block ([fd, offset]) is called *filemap*.

$$[\text{fd,offset}] \xrightarrow{F2L} \text{LBA} \xrightarrow{L2P} \text{PPA} \tag{1}$$

In legacy IO stack, F2L mapping and L2P mapping are exclusively managed by the host filesystem and the device FTL, respectively. Fig. 2 illustrates the concept of filesystem-level garbage collection and device-level garbage collection. In the original layout (Fig. 2a), there are four sections in the filesystem partition and four flash blocks at the storage partition. Fig. 2b illustrates the filesystem state after filesystem-level garbage collection runs. The three sections are reclaimed at the filesystem partition. The victim blocks are relocated to the new filesystem location. Filesystem-level garbage collection updates F2L mapping. The L2P mapping remains unchanged.

Fig. 2c illustrates the filesystem state and storage system state after the device-level garbage collection runs. Device-level garbage collection places the valid flash pages at the new flash block. It updates the L2P mapping. Two flash blocks are reclaimed as a result of running device-level garbage collection. Here, the F2L mapping remains unchanged.

Coupled Garbage Collection updates both F2L mapping and L2P mapping. After CGC migrates the victim flash pages to the new flash block, CGC updates not only the associated mapping table entry at the storage device but also the LBA of the victim flash page. We call the CGC's activity of updating the logical block address of the victim flash page as *remap*. Fig. 2d illustrates the filesystem state and storage state after CGC runs. With CGC, FTL frees two flash blocks and three filesystem sections. CGC migrates the victim flash pages and updates their LBAs. CGC ensures that the victim flash pages are clustered not only physically, i.e. at the same flash block, but also logically, i.e. at the same section at the filesystem partition. It reclaims the free filesystem sections without any data traffic between the host and the storage device.

Coupled Garbage Collection consists of three key ingredients: Block Associative Mapping, Consolidation of Valid blocks and F2L mapping Synchronization.

4.2 Block Associative Mapping

We develop *Block Associative Mapping*. For L2P mapping, we use block mapping [39]. Unlike legacy block mapping, our FTL allows the valid flash pages in the victim flash block to be placed anywhere in the target flash block. The block offset can change when a flash page is migrated to the target flash block. This mechanism is called *Block Associative Migration*. We call this L2P mapping scheme as Block Mapping with Block Associative Migration, Block Associative Mapping, for short. The key idea of Block Associative Mapping has been around for several decades. It has been widely used in managing the CPU cache, e.g. set-associative cache [9, 67], or the main memory, e.g. virtual memory [10, 37].

To preserve the invariant of the Block Mapping, we develop a mechanism, *remap*. In remap, if a flash page is migrated to the new flash block, the FTL updates the LBA of the victim flash page to a new one. With remap, the original content that was placed at LBA L and associated physical address P is migrated to the new LBA L' and new physical location P'. Let us provide an example. Assume there are two filesystem sections, starting at LBA 100 and LBA 200, respectively. For convenience, we refer to the flash block mapped to the former section as block α and the latter as block β. A flash page in block α with LBA 108 is migrated to block β, changing its block offset from 8 to 1. Then, the LBA of the migrated flash page is updated from 108 to 201.

A fair number of works have been proposed to have the filesystem to manage device's L2P mapping [17,49,58]. Block Associative Mapping proposes the opposite way; the storage device manages the host's F2L mapping.

4.3 Consolidating Valid Flash Pages

Consolidating the valid flash pages consists of three phases: (i) victim block selection, (ii) valid page copy and (iii) remap. The first phase, victim block selection, uses the same mechanism as the existing device-level garbage collection in selecting the victim blocks, e.g. greedy method [36], age policy [70], cost-benefit policy [64], and time window [25]. CGC currently uses the greedy method for victim selection.

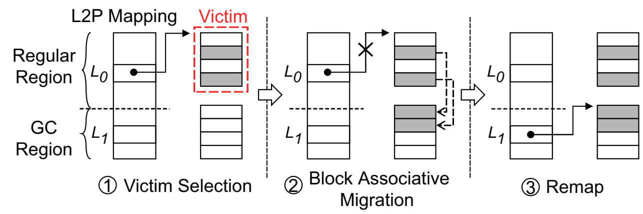


Figure 3: Coupled Garbage Collection. (L_0: victim section, L_1: target section)

The second phase is Block Associative Migration. The FTL allocates a free flash block as a destination and *sequentially* places the valid flash pages in the victim block to the destination. The offsets of the valid flash pages can change after they are migrated to the destination.

The third phase is Remap. CGC allocates a free entry in the mapping table and maps the destination flash block to the newly found mapping table entry. In allocating the new mapping table entry, the storage device may conflict with the host's filesystem. To prohibit the CGC from interfering with the host, we reserve a small region in the filesystem partition exclusively for CGC's use. We call this region as *garbage collection* region. CGC is allowed to allocate the mapping table entries associated with only the garbage collection region. Fig. 3 illustrates an example. CGC migrates the valid flash pages at the victim flash block, L_0, to the destination flash block. The block offset of the flash page changes when it is migrated to the destination flash block. When the valid pages are migrated to the new location, the destination block is assigned a mapping table entry, L_1, from the garbage collection region. The victim block L_0 is discarded.

The key of CGC is *remap*. Through remap, CGC integrates the filesystem-level garbage collection with the device-level garbage collection; it consolidates the valid filesystem blocks via migrating the valid flash pages to the new locations and remapping them to the new LBAs.

4.4 Synchronizing Updated LBAs to the Host

When the storage device updates the LBA of the migrated flash page, the storage device creates the record, <old LBA, new LBA>. It denotes the old LBA and the new LBA of the remapped page, respectively. We call it a migration record. The storage device sends all or a subset of the outstanding migration records to the host in a batched manner. When the host receives the migration records from the device, the host filesystem updates its F2L mapping accordingly. A migration record at the storage device can be in one of the three states: buffered, in-flight, and synced. The migration record is in buffered state from its creation until it is sent to the host. The migration record is in in-flight state from when it is sent to the host until the storage device is informed by the host that the filesystem state has been properly updated. The migration record enters the synced state when the storage device is notified that the host has updated the filesystem state properly. The host updates the filemap based upon the migration record which it has received from the storage device. When the host finishes updating the filesystem state with respect to the migration record, the host notifies it to the storage device.

There is a non-zero time interval between when the storage device remaps the flash pages and when the host filesystem updates the filemap based upon the migration records sent by the storage device. During this time interval, the filesystem state at the host and the filesystem state at the device are inconsistent. Read (or discard) command from the host may access the obsolete LBA whose page has been migrated to the new LBA by the storage device. To avoid the filesystem from reading or discarding an obsolete LBA, we develop *read-redirect* and *discard-redirect*. When the read (or discard) finds that the mapping table entry of the data block it needs to access is NULL, the FTL looks up the outstanding migration records. If it finds that the data block is migrated to the new location, it redirects the read (or discard) to the new location.

The FTL recycles the mapping table entry only after it ensures that the F2L mapping at the host has been updated with respect to the migration record. It is guaranteed that if an incoming read request finds that the mapping table entry is NON-NULL, the mapping is consistent with the host filesystem. For example, assume that the host issues a read command at LBA `0x000101`. The storage device has migrated the flash page and updated its LBA from `0x000101` to `0x800101`. The FTL finds that the mapping table entry for the LBA `0x000101` is NULL and that there is a migration record <`0x000101`, `0x800101`>. Referring to the migration record, the FTL redirects read command to `0x800101`.

5 Migration Upcall

5.1 A Concept

We develop an upcall [12] like mechanism to deliver the migration records from the storage device to the host. We call it a *Migration Upcall*. The existing IO interface mechanisms, interrupt and polling, are host-centric. They leave substantial room for improvement in informing the host about the device-initiated activity such as Coupled Garbage Collection. Using polling can waste host CPU cycles and can affect the foreground application performance. Using the interrupt mechanism for notifying the host about the migration activity is also non-trivial. Adding a new interrupt for Migration Upcall involves defining a new interrupt, modifying an interrupt vector table, and writing a new interrupt handler, if we are lucky to find an unused interrupt number available. Migration Upcall is a device-centric IO interface mechanism (Fig. 4). Migration upcall is carefully designed to minimize any changes in the existing IO stack. Migration upcall does not introduce any new data structures, interrupt handlers nor new interfaces.

Migration Upcall consists of an upcall ID, an array of migration records, and the number of migration records. The

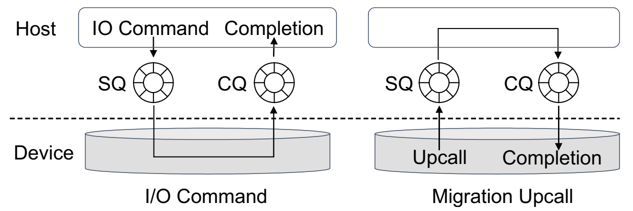


Figure 4: IO Command vs. Migration Upcall. (SQ: Submission Queue, CQ: Completion Queue)

control flow of the Migration Upcall is opposite to the control flow of IO command in the legacy IO stack. In Migration Upcall, the device submits an upcall to the host. The host processes it. Then, the host informs the storage device when it finishes servicing the upcall.

We define Migration Upcall based upon the existing NVMe standard. Migration upcall uses the command data structure of NVMe [31]. Migration Upcall uses NVMe queue pair (submission queue and completion queue) to send and receive the migration records to and from the host, respectively. We call it a Migration Queue pair. In submitting an upcall to the host, the storage device places the upcall at the SQ of the migration queue pair and notifies the host that the upcall is ready (Fig. 4). When the host finishes processing the upcall, it places a completion signal at the CQ of the migration queue and rings a doorbell.

5.2 Upcall Piggybacking

The storage device needs to signal the host when it places an upcall at the migration SQ. We do not use interrupt nor polling in signaling the host about the upcall. We develop a new mechanism to inform the host that the upcall is ready at the SQ, *Upcall Piggybacking*. In Upcall Piggybacking, the storage device piggybacks the upcall notification on the NVMe command completion signal [31]. In the legacy NVMe interface, the IO device places the command completion signal at the CQ when it finishes servicing the command.

When the storage device places the upcall at the migration SQ, it looks for any command completion signal to be sent to the host. When there is one, the storage device sets the newly defined `UPCALL` flag at the command completion signal that is going to be sent to the host. When the NVMe device driver at the host finds the `UPCALL` flag at the incoming completion signal, the device driver wakes up the upcall handler. The upcall handler removes the migration upcall from the migration SQ and services it.

With Upcall Piggybacking, the storage device can signal the host about the migration upcall without significant interference on the host. Despite piggybacking, we do not find any significant delay in delivering the migration records to the host. This is because the device-level garbage collection is triggered mostly (if not all) when there exist the frequent

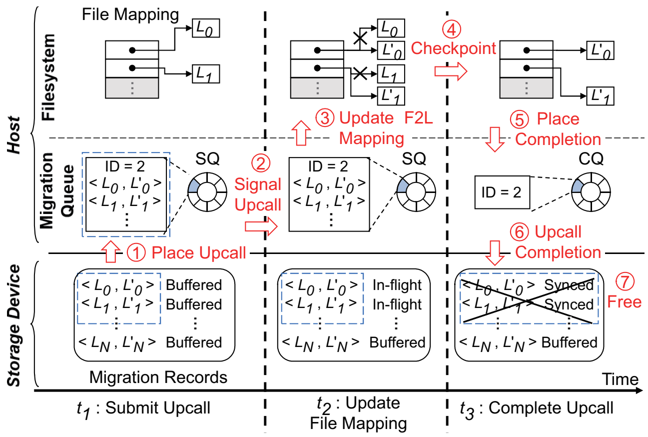


Figure 5: Synchronizing the F2L mapping through Migration Upcall (L: old LBA, L': new LBA); Checkpoint(④) makes updated F2L mapping persistent. Then, the filesystem completes the checkpointed upcall (⑤).

write requests from the host. There is no lack of IO commands to piggyback the migration records when the storage device needs to send the migration records. In the current implementation of D2FS, a single migration upcall can carry up to 256 migration records.

Fig. 5 illustrates the process of handling Migration Upcall. The device inserts the upcall at the SQ of Migration Queue (step ①). Then, by Upcall Piggybacking, the device informs the host that the upcall is ready (step ②).

5.3 Handling a Migration Upcall

Handling a Migration Upcall consists of three tasks: (i) Update the filesystem state with respect to the migration records, (ii) Checkpoint the migration records that are reflected to the filesystem, and (iii) Notify the storage about the completed migration upcalls. We allocate a dedicated thread for handling the migration upcalls. Unless noted otherwise, the description here is based upon F2FS.

Filesystem Update. The upcall handler reads the migration upcall from the SQ of the migration queue and extracts the migration records from the migration upcall. For each migration record, the upcall handler updates the file mapping, block bitmap and reverse mapping (step ③ in Fig. 5). Block bitmap (segment information table in F2FS) is a data structure to keep track of the allocation status of the blocks in a segment. Reverse mapping (segment summary area in F2FS) is a data structure to store the owner information of the filesystem block, such as the inode ID and the file offset of the associated filesystem block. The upcall handler repeats updating the filemap, block bitmap, and reverse mapping information for each migration record in the migration upcall. The upcall handler locates the filemap for the old LBA and updates it to refer to the new LBA. Then, the upcall handler updates the block bitmaps and the reverse mappings of an old and a new LBA. The upcall handler also updates the number of valid blocks in the sections associated with old LBA and new LBA of the migration record, respectively. The upcall handler may find that the old LBA of the migration record is invalid. This can happen if the file block is invalidated, e.g. `unlink()` while the migration record is in transit. If the old LBA of the migration record is invalid, the upcall handler marks it as serviced without updating the filesystem state.

Filesystem Checkpoint. D2FS checkpoints its state under the same condition as F2FS does. D2FS executes the checkpoint when the number of dirty metadata entries or the number of the invalid sections reach a certain threshold [1] (step ④ in Fig. 5). Migration Upcall is failure-atomic and yet preemptive. The filesystem checkpoint routine can preempt the upcall handler and can store the partially updated filesystem state. Migration upcall ensures the failure-atomicity with two key technical ingredients. First, D2FS processes the migration records in the order in which they are created. D2FS stores the (record id, upcall id) of the latest migration record which it has processed at the checkpoint pack. Second, D2FS processes a single migration record in failure-atomic manner. The upcall handler disables the checkpoint routine when it processes a migration record. The upcall handler acquires a shared lock on a global `rwsemaphore` [2] so that it blocks the checkpoint and yet it allows the other threads to use the filesystem service.

Notify the Upcall Completion. The migration upcall is said to be completed if all of its migration records are reflected at the filesystem and the updated filesystem state is checkpointed. When the checkpoint completes, the completion thread notifies the storage device about the completed migration upcalls (step ⑤ and ⑥ in Fig. 5). The completion thread places the completion signal that contains the ids of completed upcalls at the CQ of the migration queue. Then it rings a doorbell. When the storage device receives the completion signal, it changes the state of the migration records in the completed migration upcalls from in-flight to synced. The migration records at synced state are deallocated asynchronously by the storage FTL (step ⑦).

Preserving the Order. The filesystem should process the migration records in order. A flash page may be migrated multiple times by CGC and the storage device can send the associated migration records as a single batch. For the consistency of the filesystem state, the upcall handler services the migration records in a migration upcall in the order in which CGC performs the associated migration. To this end, the storage device places the migration records at the migration upcall object sequentially with respect to the order in which the migration records are created. The storage device inserts the migration upcall at the SQ of the single migration queue in the order the migration upcalls are created. At the host side, the filesystem receives the migration upcalls in the

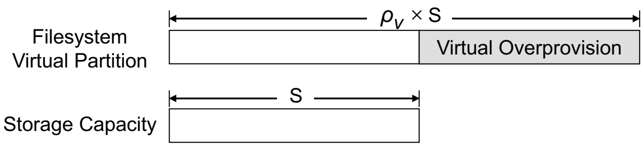


Figure 6: Virtual Overprovisioning: A Concept (S: size of storage device, ρ_v: virtual overprovisioning degree)

order the migration upcalls arrive at the SQ. For each migration upcall, the filesystem processes the migration records one by one by scanning the array of migration records in it.

6 Virtual Overprovisioning

6.1 A Concept

Coupled Garbage Collection should reclaim the free flash blocks on time so that the filesystem can be supplied with the free sections without any interruption. Our approach to achieve this objective is simple and yet effective. We separate the size of the filesystem partition from the capacity of the underlying storage [33, 40]. We make the filesystem partition size sufficiently larger than the capacity of the underlying storage so that the storage runs out of free blocks before the filesystem runs out of free sections. This ensures that the CGC runs and reclaims the free sections for the filesystem before the filesystem runs out of free sections. We call this technique Virtual Overprovisioning. We use the term "virtual" since the overprovisioned space does not physically exist. The ratio of the filesystem partition size to the storage device capacity is referred to as the virtual overprovisioning degree, ρ_v. For example, for 2 TByte virtual partition on 1 TByte storage, ρ_v is 2. Fig. 6 illustrates the virtual overprovisioning. The filesystem runs out of free sections less frequently as its partition size gets larger. IPLFS [40] is an extreme form of virtual overprovisioning. IPLFS's filesystem partition size is fixed to 8 ZByte. For 1 TByte storage, ρ_v is of 2^{33} in IPLFS.

D2FS uses the best-effort policy in issuing the discard command to the storage device unlike the existing filesystems. When using virtual overprovisioning, it is important to inform the storage device about invalid filesystem blocks as early as possible. In virtual overprovisioning, the filesystem utilization is represented by the number of valid blocks in its partition. As a result, the storage device can run out of free flash blocks while the filesystem is not full. We call this situation a *free section fault*. A free section fault occurs because the filesystem does not inform the invalid filesystem blocks on time and because there is no invalid flash pages for the storage device to reclaim. If the free section fault occurs, the storage device cannot start reclaiming the free flash blocks until the filesystem informs the storage device about the invalidated filesystem blocks. To prevent the free section fault, the filesystem needs to inform the storage device about the invalid filesystem blocks as early as possible.

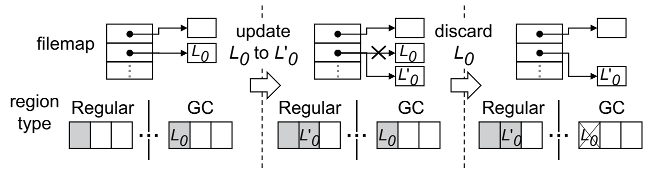


Figure 7: Updating file block on garbage collection region

Via a small degree of virtual overprovisioning combined with the Coupled Garbage Collection, we save the log-structured filesystem from the garbage collection.

6.2 Garbage Collection Region

In D2FS, not only the filesystem but also the storage device can allocate the LBA from the filesystem partition. We devise a mechanism to prohibit the storage device from conflicting with the filesystem in allocating LBA. We set aside a region in the filesystem partition and have this region exclusively allocated by the storage device. We call this region *garbage collection region*. We call it the garbage collection region since this region is used by the Coupled Garbage Collection to allocate the LBA for the destination flash block. To distinguish the rest of the filesystem partition from the garbage collection region, we refer to the rest of the filesystem partition as regular region. The filesystem can read, update, or discard the file blocks in the garbage collection region, but it cannot allocate the LBA from the garbage collection region. When the file block in the garbage collection region is updated, the updated content is allocated from the regular region. Fig. 7 illustrates this situation. The file block at LBA L_0 in the garbage collection region is updated. The updated content is written to LBA L'_0 from the regular region. The LBA for the updated file block is changed from L_0 to L'_0. The filesystem discards L_0 so that the Coupled Garbage Collection can recycle it later.

6.3 Filesystem Implementation

Coupled Garbage Collection, Migration Upcall and Virtual Overprovisioning can be used in any log-structured filesystem. Currently, we implement D2FS on F2FS. There are a few implementation-specific issues that deserve attention.

Filesystem Organization. The D2FS partition consists of two regions: the regular region and the garbage collection region. Fig. 8 illustrates the filesystem organization. The regular region is divided into two areas: metadata area where the filesystem stores metadata, and main area where the filesystem logs file blocks. D2FS has the same metadata structure as F2FS. Further interested readers are kindly referred to [46]. The size of the entire filesystem partition corresponds to $\rho_v \times (\textit{storage device capacity})$. The amount of blocks in the garbage collection region cannot exceed the storage device capacity. We set the size of the garbage collection region

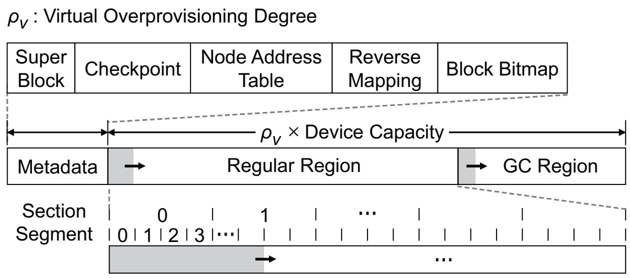
ρ


Figure 8: Partition Layout of D2FS

equal to the storage device capacity. The size of the regular region corresponds to $(\rho_v - 1) \times (\mathit{storage\ device\ capacity})$.

Immediate Discard. Existing filesystems, e.g. F2FS, BTRFS, send the discard commands in batched manner [46, 63]. This is to prohibit the discard commands from interfering with the latency-sensitive commands from the host, e.g. read or flush [6, 7]. For example, F2FS dispatches the discard command only when the IO is idle. Unlike the existing approaches, D2FS dispatches the discard command right after it invalidates the filesystem blocks. This is to invalidate the flash pages that contain the invalid filesystem blocks as early as possible. We carefully suspect that immediate discard policy is not infeasible primarily because the overhead of servicing a discard command becomes much lighter in recent SSD products than before [8, 16, 26, 30, 65].

Specifying Block Type in IO Command D2FS defines two block types for the blocks in its filesystem partition; data block and filemap block. In sending the write command to the storage, D2FS specifies the block type using *stream* interface [31, 35]. Coupled Garbage Collection clusters the flash pages of the same block type at the same flash block.

7 Crash Recovery

The system may crash when the filesystem is in the middle of servicing the migration upcall. In this case, the partial subset of migration records for the migration upcall may have been made durable. D2FS ensures the failure-atomicity of servicing a migration upcall using *redo* semantics; in the filesystem recovery, D2FS recovers the outstanding migration records for the migration upcall and replays them.

There are three key ingredients to make the migration upcall failure-atomic. First, the outstanding migration records, i.e. the migration records with buffered or in-flight state, are protected under unexpected power failure. Commodity SSD products adopt the various techniques to protect the state of the storage device against the unexpected power failure, e.g. using supercap [23] or spare area of flash page [34]. We assume that the outstanding migration records in device's memory are protected by these techniques. Second, D2FS guarantees the failure-atomicity for processing a single migration record. The filesystem disables the checkpoint when it processes a migration record. Via disabling the checkpoint during this period, the filesystem ensures that while processing a migration record, the transient state of the filesystem is not made durable and is not externally visible. Third is the re-submission of the outstanding migration records. In crash recovery, the recovery routine of the filesystem examines the checkpoint pack to identify the most recently serviced migration record and instructs the storage device to re-submit the outstanding migration records, i.e. the migration records that are in in-flight state. When the host receives the migration records from the storage device, it checks if the migration record is not serviced. By processing the remaining outstanding migration records for the migration upcall, the recovery routine redoes the migration upcall.

8 Evaluation

8.1 Experiment Setup

We implement D2FS and block IO layer in Linux (v5.11.0). We use F2FS [46] as our baseline filesystem. The total LoC in the Linux kernel and the storage firmware is approximately 5.4K. We use a server (Intel Xeon, 2.10 GHz, 40 cores, 512 GByte DRAM) for our experiment. We use the SSD emulator NVMeVirt [41] for our experiment. With this emulator, we model Samsung 970 Pro [74] (8 channels, 16 chips, 16 KByte flash pages, 2 MByte flash blocks, 32 MByte superblock, and 7% OVP ratio). We set the size of the emulated SSD to 256 GByte and limit the host main memory size to 64 GByte.

We compare D2FS with F2FS [46], zoned F2FS [4] , and IPLFS [40]. Zoned F2FS is a version of F2FS ported on ZNS SSD. The zone size is set to 32 MByte, the size of a superblock used in the prior works [28, 29]. IPLFS is the latest log-structured filesystem without garbage collection known to the public. IPLFS is paired with Interval Mapping FTL [40], which maintains a space-efficient L2P mapping structure for the extremely large filesystem partition. We examine the following: the effect of virtual overprovisioning degree (§8.2), the effect of eliminating filesystem-level garbage collection (§8.3 and §8.4), comparison between the filesystem-level garbage collection and device-level garbage collection (§8.5), and memory pressure (§8.6).

We use FIO [13], TPC-C [60], YCSB [22] workload A and F, and Fileserver [73] benchmarks. In FIO, a single thread performs 4 KByte random write on a 180 GByte file for 1200 secs. In TPC-C, we fill the disk with 60 GByte cold file and perform database operation on 155 GByte MySQL DB. In TPC-C, 15 threads run 1700 warehouses on MySQL DB for 1800 secs. In YCSB, we create 50 GByte cold file and 175 GByte MySQL DB, and 15 threads perform 10M operations on 16M of 8 KByte records. YCSB-A consists of 20% read and 80% update. YCSB-F consists of 20% read and 80% read-modify-write. In Fileserver, 45,000 files are initially created

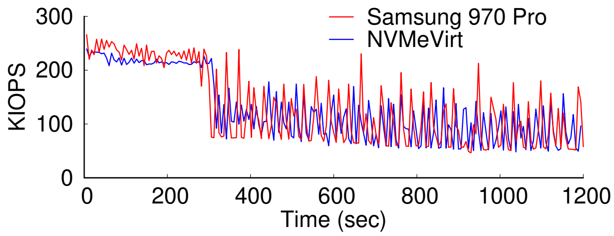


Figure 9: Emulator Fidelity Test (FIO, zoned F2FS)

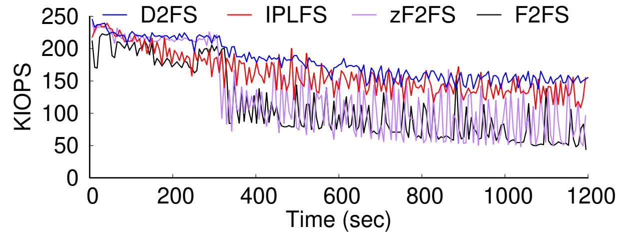


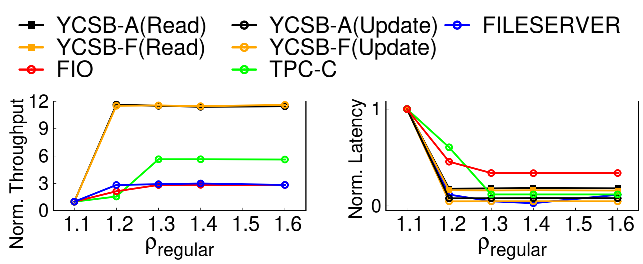
Figure 12: FIO Throughput (zF2FS: zoned F2FS)

and each file size is 4 MByte in average (total 180 GByte). Fifteen threads perform create, write (4 MByte), append (4 KByte), and delete on those files randomly for 400 secs.

We examine the fidelity of the SSD emulator [41] used in this work. We compare the performance of the SSD emulator against the real SSD product, Samsung 970 Pro, which it emulates. We run FIO benchmark with zoned F2FS. Fig. 9 illustrates the result. We carefully believe that the emulator renders reasonably accurate performance to the real device.

8.2 Virtual Overprovisioning Degree

We vary the virtual overprovisioning degree, and examine the performance of each workload (Fig. 10). The filesystem partition consists of the regular region and the garbage collection region. We vary the virtual overprovisioning degree of regular region, $\rho_{regular}$. Recall that the size of garbage collection region is set to the device capacity. When $\rho_{regular}$ is less than 1.4, the filesystem occasionally runs out of the free sections. The filesystem temporarily freezes free block allocation until the CGC reclaims the free sections. With $\rho_{regular}$



(a) Throughput (b) Average Latency

Figure 10: Performance with varying Virtual Overprovisioning Degree. (the value at Y-axis is normalized to the value at $\rho_{regular} = 1.1$.)

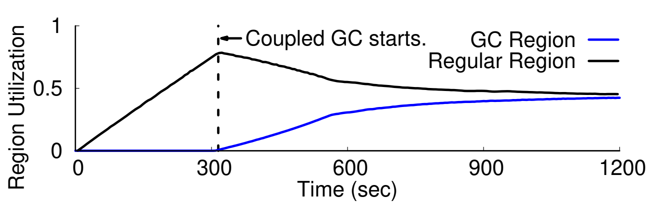


Figure 11: Region Utilization, FIO (ρ_v: 2.4, $\rho_{regular}$:1.4, ρ_{gc}:1)

of 1.4 or larger, CGC always leaves sufficient free sections in the filesystem. In all subsequent experiments, we set $\rho_{regular}$ to 1.4, resulting in the virtual overprovisioning degree of the entire filesystem partition, ρ_v, being 2.4.

We examine how CGC reclaims the free flash blocks and how the filesystem utilization changes as CGC proceeds (Fig. 11). We use FIO in this experiment. The filesystem region utilization is the ratio of filesystem blocks that are not free in the filesystem partition. From the beginning till the CGC starts, the utilization of regular region increases to 70%. At this point, the storage device runs out of the free flash pages and it starts CGC. The CGC reallocates the valid filesystem blocks from the regular region to the garbage collection region. The utilization of regular region decreases while that of the garbage collection region increases. After 1000 seconds, the utilization of each region reaches a steady state. This is because CGC creates the free sections at the same rate at which the free sections are consumed by FIO benchmark.

8.3 Throughput

FIO. We examine the throughput for each filesystem (Fig. 12). In D2FS, the FIO throughput decreases by 30% when the device starts to reclaim the free flash blocks. F2FS is subject to the filesystem-level as well as the device-level garbage collection. It renders 80% performance drop when garbage collections start. Zoned F2FS renders 75% performance drop when the garbage collection start. The performance drop is slightly less severe since the device-level garbage collection does not run. D2FS outperforms zoned F2FS by about 1.7×. This is because D2FS is free from expensive filesystem-level garbage collection. As same as D2FS, IPLFS dispenses with filesystem-level garbage collection. However, D2FS outperforms IPLFS by about 15%. This is because IPLFS suffers from larger FTL overhead [40]. FTL in IPLFS periodically reorganizes the L2P mapping structure to minimize the FTL memory footprint. This activity hinders the FTL from servicing the incoming IO commands, reducing the system performance. We observe that D2FS, IPLFS and zoned F2FS exhibit similar WAF (about 1.4). The WAF of F2FS is slightly higher, 1.5. We carefully suspect that F2FS renders higher WAF since both the filesystem and the storage device run garbage collection.

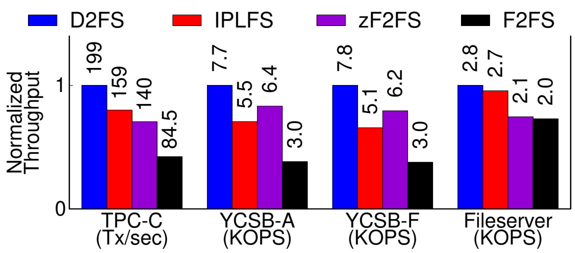


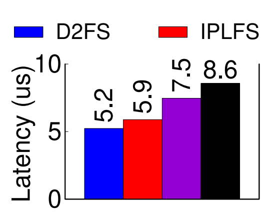
Figure 13: Macrobenchmark: Throughput (Tx: Transaction)

Macrobenchmark. D2FS yields better performance than the other filesystems in macrobenchmarks (Fig. 13). D2FS outperforms zoned F2FS by up to 1.4× in TPC-C. IPLFS shows lower performance compared to zoned F2FS in YCSB. The Interval Mapping FTL of IPLFS periodically reorganizes its L2P mapping table to reduce its size [40]. During this compaction, the FTL pauses to handle incoming I/O commands. In fsync-intensive YCSB workloads, the compaction overhead directly impacts the user application, as it waits for I/O operations to complete.

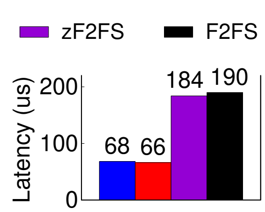
8.4 Latency

FIO. We examine the impact of eliminating the filesystem-level garbage collection on the 4 KByte write latency (Fig. 14). Eliminating the filesystem-level garbage collection brings notable improvements in tail latency, which is crucial for performance predictability [38, 50, 79]. D2FS reduces the 99.99th tail latency to $\frac{1}{3}$ compared to zoned F2FS and F2FS.

Macrobenchmark. We measure the latency of each benchmark operation (TPC-C: Payment Transaction, YCSB: 8 KByte record update and read, Fileserver: 4 MByte write). By eliminating the garbage collection from the filesystem, the average and tail latency improves significantly (Fig. 15). Compared to F2FS, D2FS reduces the average latency and the tail latency at 99.95% up to $\frac{1}{5}$ and $\frac{1}{11}$ (YCSB-F update), respectively. D2FS reduces average latency to about $\frac{1}{5}$ (Fileserver) and $\frac{1}{2}$ (YCSB-F update) compared to zoned F2FS. Compared to IPLFS, D2FS reduces the average latency and tail latency at 99.95% up to $\frac{1}{2}$ and $\frac{1}{8}$ (YCSB-F), respectively. This is because the flash storage in D2FS is free from the compaction overhead of the Interval Mapping FTL.

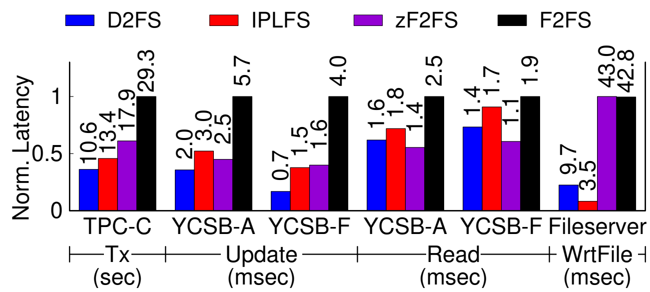


(a) Average

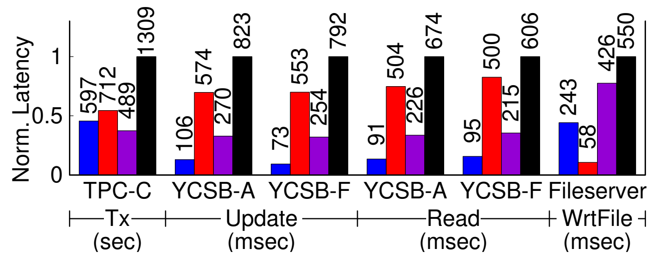


(b) 99.99th

Figure 14: FIO: Latency



(a) Average Latency



(b) Tail Latency at 99.95th

Figure 15: Macrobenchmark: Latency (Tx: Payment Transaction, WrtFile: Writing 4 MByte file)

8.5 Device-level GC vs. Filesystem-level GC

Both D2FS and zoned F2FS aim at mitigating the garbage collection overhead in IO stack. However, their approaches lie at the other end of the extreme; ZNS SSD removes the garbage collection from the storage device, and D2FS removes it from the host filesystem. We examine the latencies of reclaiming a single filesystem section via the filesystem-level garbage collection (zoned F2FS) and via device-level garbage collection (D2FS) (Fig. 16), respectively. Unsurprisingly, the filesystem-level garbage collection is much more expensive than the device-level garbage collection in reclaiming a filesystem section. The latency of filesystem-level garbage collection is longer than that of device-level garbage collection by 3× - 10× in reclaiming a single section in the filesystem.

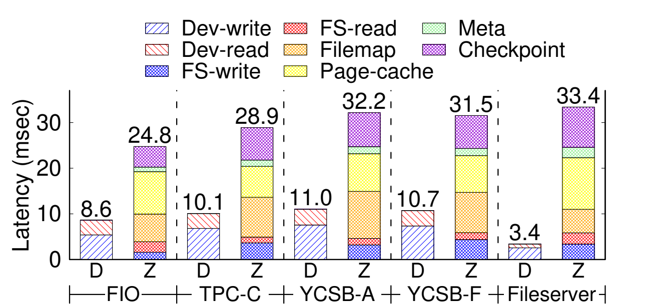


Figure 16: Garbage Collection Latency Breakdown on a single 32 MByte superblock (Dev: Device, FS: Filesystem, D: D2FS, Z: zoned F2FS), (Meta: read metadata, Page-cache: allocate page cache, Filemap: traverse filemap blocks)

Filesystem	F2FS	IPLFS	zF2FS	D2FS
Memory (MByte)	256.0	392.0	10.4	27.2

Table 1: FTL Memory Overhead in FIO (256 GByte SSD)

The latency of filesystem-level garbage collection deserves a closer look. In filesystem-level garbage collection, the overhead of allocating the page cache entry and the overhead of navigating through the filemap blocks account for 60 % of total garbage collection latency. Also, the overhead of dispatching write command to the storage device is significant, upto 13 % (YCSB-F) even though they are asynchronous write. On the other hand, the overhead of reading the valid file blocks from the storage device accounts for only about 5 % of the garbage collection latency. We find that most of the valid file blocks are already in the page cache. The overhead of checkpointing the filesystem state including the overhead of writing the newly consolidated file section to the storage corresponds to about 20 % of the garbage collection latency.

D2FS's device-level garbage collection can dispense with all these filesystem-level overhead and yet can reclaim the free sections at the filesystem.

8.6 FTL Memory Overhead

We examine the FTL memory overhead of individual filesystems we tested (Table 1). The page mapping FTL of F2FS requires 256 MByte memory since it allocates 4 Byte for 4 KByte logical block.

IPLFS uses a page granularity mapping FTL. It uses Interval Mapping [40] to reduce the memory pressure for mapping. The memory requirement for Interval Mapping is expected to be as small as that of page mapping. In reality, it renders 1.5× larger L2P mapping table than page mapping table. Its memory saving mechanism does not work well under small random write workload [40]. This memory overhead poses challenges for deploying IPLFS in real-world storage devices with limited power and hardware budgets [24, 53]. For 1 PByte SSD, for example, IPLFS requires 1.5 TByte memory for mapping.

The FTL of zoned F2FS requires 10.4 MByte in total for page mapping for the metadata area and the block mapping for the main area. Its size is the smallest among all filesystems we tested. The FTL of D2FS requires 2.4× larger memory space than zoned F2FS's due to the Virtual Overprovisioning (total 25.2 MByte). The FTL of D2FS maintains the outstanding migration records in its buffer. We examine how the buffer size changes with respect to the frequency of the migration upcall to the host. Fig. 17 depicts the buffer size for migration records during running FIO benchmark. We observe that it is sufficient to allocate 2 MByte for the buffer in all benchmarks.

9 Related Work

A number of log-structured filesystems have been crafted to exploit an append-only nature of flash memory [28, 29, 40, 46, 51, 56]. ZNS+ [28] exploits SSD's internal parallelism-aware `copyback` command to mitigate the garbage collection of ZNS SSD. IPLFS proposed to use a extremely large logical partition to eliminate the filesystem garbage collection [40].

Several studies proposed that the host filesystem directly manages the L2P mapping [16, 17, 49, 58, 82, 83]. These works are dedicated to eliminate the redundant effort between the garbage collection at the host filesystem and the garbage collection at the storage device. With this feature, the host filesystem can directly consolidate the valid flash pages of the flash device without relying on FTL. This approach, however, suffers from costly filesystem garbage collection overhead.

Several works have been proposed to reduce the overhead of the device-level garbage collection, such as hot/cold separation [21, 35], pre-emptive garbage collection [42, 48], and parallel garbage collection [68]. These techniques complement our FTL and can be combined with our FTL. Some works use the learned index for L2P mapping to reduce the space overhead of mapping table [71, 76]. Learned-index based L2P mapping accompanies the overhead of model training. Our L2P mapping can significantly minimize the mapping table without that computational overhead.

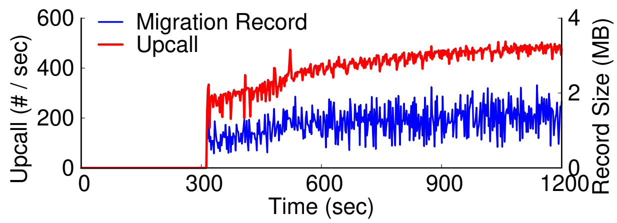


Figure 17: FIO: Upcall Submission Rate and Memory Footprint of Migration Record.

10 Concluding Remarks

We propose yet another way to eliminate the garbage collection from the log-structured filesystem. Log-structured filesystems were originally designed for hard disk drives, which lack built-in garbage collection capabilities. In contrast, a flash storage inherently supports garbage collection. We believe that the log-structured filesystems on flash storage can dispense with the filesystem-level garbage collection when we can use the garbage collection functionality of the flash storage to reclaim the free filesystem section. In this paper, we demonstrate a method to achieve this.

Acknowledgements We are deeply indebted to our shepherd Ming-Chang Yang for helping us to shape the final version of this paper. We are grateful to the anonymous reviewers for their valuable comments and feedback. This work was supported by SK hynix Inc., IITP (RS-2024-00459026), and NRF (NRF-2020R1A2C3008525 and NRF-2021R1A2C2095125).

References

[1] F2FS checkpoint policy. https://elixir.bootlin.com/linux/latest/source/fs/f2fs/segment.c#L462, 2012.

[2] F2FS checkpoint rwsemaphore. https://elixir.bootlin.com/linux/latest/source/fs/f2fs/f2fs.h#L1571, 2012.

[3] F2FS garbage collection. https://elixir.bootlin.com/linux/latest/source/fs/f2fs/gc.c#L1797, 2012.

[4] F2FS support for Zoned Storage. https://zonedstorage.io/docs/filesystems/f2fs, 2012.

[5] Garbage collection semaphore in F2FS. https://elixir.bootlin.com/linux/latest/source/fs/f2fs/f2fs.h#L1706, 2012.

[6] SSD Trim Maintenance | The FreeBSD Forums. https://forums.freebsd.org/threads/ssd-trim-maintenance.56951/#post-328912, 2016.

[7] There are two very different types of trim command | lwn.net. https://lwn.net/Articles/787736/, 2019.

[8] F2FS Patch: Aggressive discard option added. https://www.mail-archive.com/linux-f2fs-devel@lists.sourceforge.net/msg27923.html, 2023.

[9] Anant Agarwal and Stephen D Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proc. of 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[10] Gene M Amdahl, Gerrit A Blaauw, and Frederick P Brooks. Architecture of the ibm system/360. *IBM Journal of Research and Development*, 8(2):87–101, 1964.

[11] Ashok Anand, Sayandeep Sen, Andrew Krioukov, Florentina I Popovici, Aditya Akella, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Suman Banerjee. Avoiding File System Micromanagement with Range Writes. In *Proc. of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[12] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. of ACM SIGOPS 17th Symposium on Operating Systems Principles (SOSP)*, 1991.

[13] Jens Axboe. FIO-flexible i/o tester synthetic benchmark. https://github.com/axboe/fio, 2005.

[14] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. ZNSwap: Un-block your swap. *ACM Transactions on Storage (TOS)*, 19(2):1–25, 2023.

[15] Matias Bjørling. Zone Append: A new way of writing to zoned storage. In *Proc. of USENIX Linux Strorage and Filesystem Conference (VAULT)*, 2020.

[16] 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 *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[17] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proc. of 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[18] Trevor Blackwell, Jeffrey Harris, and Margo I Seltzer. Heuristic Cleaning Algorithms in Log-Structured File Systems. In *Proc. of USENIX 1995 Winter Technical Conference (TCON)*, 1995.

[19] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A fast, Cost-Effective LSM-tree based KV store on hybrid storage. In *Proc. of 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[20] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proc. of 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[21] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, 29(3):267–290, 1999.

[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[23] Intel Corporation. Power Loss Imminent (PLI) Technology. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/ssd-power-loss-imminent-technology-brief.pdf, 2014.

[24] Niv Dayan, Philippe Bonnet, and Stratos Idreos. GeckoFTL: Scalable flash translation techniques for very large flash devices. In *Proc. of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[25] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. Idle time garbage collection scheduling. In *Proc. of 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[26] Western Digital. Western Digital Champions Zoned Storage, An Open Initiative to Redefine Data Centers for Zettabyte Scale. https://www.westerndigital.com/company/newsroom/press-releases/2019/2019-06-11-western-digital-champions-zoned-storage, 2019.

[27] Cagdas Dirik and Bruce Jacob. The Performance of PC Solid-State Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization. *ACM SIGARCH Computer Architecture News*, 37(3):279–289, 2009.

[28] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Proc. of 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[29] Joo-Young Hwang, Seokhwan Kim, Daejun Park, Yong-Gil Song, Junyoung Han, Seunghyun Choi, Sangyeun Cho, and Youjip Won. ZMS: Zone abstraction for mobile flash storage. In *Proc. of 2024 USENIX Annual Technical Conference (ATC)*, 2024.

[30] SK Hynix. SK hynix Demonstrates Industrys First ZNS-based SSD Solution for Data Centers. https://news.skhynix.com/sk-hynix-demonstrates-industrys-first-zns-based-ssd-solution-for-data-centers-2/, 2019.

[31] NVM EXPRESS INC. NVM express base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf, 2022.

[32] NVM EXPRESS INC. NVMe Zoned Namespaces (ZNS) Command Set Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-Zoned-Namespace-Command-Set-Specification-1.1e-2024.07.05-Ratified.pdf, 2024.

[33] William K Josephson, Lars A Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 6(3):1–25, 2010.

[34] Sanghyuk Jung and Yong Ho Song. Data loss recovery for power failure in flash memory storage systems. *Journal of Systems Architecture (JSA)*, 61(1):12–27, 2015.

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

[36] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A Flash-memory Based File System. In *Proc. of USENIX 1995 Winter Technical Conference (TCON)*, 1995.

[37] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, (2):223–235, 1962.

[38] Jaeho Kim, Kwanghyun Lim, Young-Don Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating Garbage Collection Interference through Spatial Separation in All Flash Arrays. In *Proc. of 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[39] 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 (TCE)*, 48(2):366–375, 2002.

[40] Juwon Kim, Minsu Jang, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. IPLFS: Log-Structured File System without Garbage Collection. In *Proc. of 2022 USENIX Annual Technical Conference (ATC)*, 2022.

[41] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. NVMeVirt: A Versatile Software-defined Virtual NVMe Device. In *Proc. of 21st USENIX Conference on File and Storage Technologies (FAST)*, 2023.

[42] Shine Kim, Jonghyun Bae, Hakbeom Jang, Wenjing Jin, Jeonghun Gong, Seungyeon Lee, Tae Jun Ham, and Jae W. Lee. Practical Erase Suspension for Modern Low-Latency SSDs. In *Proc. of 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[43] 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 *Proc. of 17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[44] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G Andersen, Gregory R Ganger, George Amvrosiadis, and Matias Bjørling. RAIZN: Redundant array of independent zoned namespaces. In *Proc. of 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[45] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review (OSR)*, 40(3):102–107, 2006.

[46] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[47] Jongsung Lee, Donguk Kim, and Jae W Lee. Waltz: Leveraging zone append to tighten the tail latency of lsm tree on zns ssd. In *Proc. of the VLDB Endowment*, volume 16, 2023.

[48] Junghee Lee, Youngjae Kim, Galen M Shipman, Sarp Oral, and Jongman Kim. Preemptible I/O scheduling of garbage collection for solid state drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 32(2):247–260, 2013.

[49] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind Arvind. Application-Managed Flash. In *Proc. of 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[50] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proc. of ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[51] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A multicore-accelerated file system for flash storage. In *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[52] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[53] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. Deepstore: In-storage acceleration for intelligent queries. In *Proc. of 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[54] Jeanna Neefe Matthews, Drew Roselli, Adam M Costello, Randolph Y Wang, and Thomas E Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM SIGOPS Operating Systems Review (OSR)*, 31(5):238–251, 1997.

[55] Micron. TN-29-28: Memory Management in NAND Flash Arrays Overview. https://www.micron.com/-/media/client/global/documents/products/technical-note/nand-flash/tn2928.pdf, 2016.

[56] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proc. of 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[57] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *Proc. of 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[58] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proc. of 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[59] Dongil Park, Seungyong Cheon, and Youjip Won. Suspend-aware segment cleaning in log-structured file system. In *Proc. of 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2015.

[60] Percona. Percona-Lab/tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql, 2017.

[61] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proc. of ACM SIGOPS 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[62] 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 *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[63] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[64] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[65] Samsung. Samsung Introduces Its First ZNS SSD With Maximized User Capacity and Enhanced Lifespan. https://semiconductor.samsung.com/news-events/news/samsung-introduces-its-first-zn

s-ssd-with-maximized-user-capacity-and-enhanced-lifespan/, 2021.

[66] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An Implementation of a Log-Structured File System for UNIX. In *Proc. of USENIX 1993 Winter Technical Conference (TCON)*, 1993.

[67] André Seznec. A case for two-way skewed-associative caches. *ACM SIGARCH Computer Architecture News*, 21(2):169–178, 1993.

[68] Narges Shahidi, Mahmut T Kandemir, Mohammad Arjomand, Chita R Das, Myoungsoo Jung, and Anand Sivasubramaniam. Exploring the Potentials of Parallel Garbage Collection in SSDs for Enterprise Storage Systems. In *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[69] Frank Shu. Data set management commands proposal for ata8 acs2. https://studylib.net/doc/7497677/non-volatile-cache-command-proposal-for-ata8-acs, 2007.

[70] Darko Stefanović, Kathryn S McKinley, and J Eliot B Moss. Age-based garbage collection. In *Proc. of 14th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[71] 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 *Proc. of 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[72] Ying Y Tai. High performance FTL for PCIe/NVMe SSDs. *Flash Memory Summit*, 2016.

[73] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX Login*, 41(1):6–12, 2016.

[74] Tech Power Up. Samsung 970 Pro. https://www.techpowerup.com/ssd-specs/samsung-970-pro-512-gb.d54, 2018.

[75] 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 *Proc. of 20th USENIX Conference on File and Storage Technologies (FAST)*, 2022.

[76] 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 *Proc. of 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[77] Shunzhuo Wang, Fei Wu, Chengmo Yang, Jiaona Zhou, Changsheng Xie, and Jiguang Wan. WAS: Wear aware superblock management for prolonging SSD lifetime. In *Proc. of 56th Annual Design Automation Conference (DAC)*, 2019.

[78] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proc. of 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[79] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Transactions on Storage (TOS)*, 13(3):1–26, 2017.

[80] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *Proc. of 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[81] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proc. of 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[82] Jinsoo Yoo, Joontaek Oh, Seongjin Lee, Youjip Won, Jin-Yong Ha, Jongsung Lee, and Junseok Shim. OrcFS: Orchestrated File System for Flash Storage. *ACM Transactions on Storage (TOS)*, 14(2):1–26, 2018.

[83] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *Proc. of 2016 USENIX Annual Technical Conference (ATC)*, 2016.

[84] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-Indirection for Flash-Based SSDs with Nameless Writes. In *Proc. of 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

A Artifact Appendix

Abstract

D2FS is a log-structured filesystem that dispenses with filesystem-level garbage collection by leveraging the garbage collection functionality of the underlying flash storage. We release the source code and experiment scripts to facilitate further research on alleviating the garbage collection overhead inherent to the log-structured filesystem.

Scope

The artifact includes experimental scripts and source code to validate the following claims presented in our paper:

- Filesystem-level garbage collection has a significantly more negative impact on application performance compared to device-level garbage collection (Fig. 1).
- By offloading the garbage collection activity of the log-structured filesystem to the underlying flash storage, D2FS outperforms state-of-the-art log-structured filesystems, including IPLFS [40], F2FS with ZNS support [4] and F2FS [46] (Figs. 12, 13, 14, 15, and 16).
- D2FS significantly reduces the memory footprint of the FTL compared to both page-mapping FTL and Interval Mapping FTL [40] (Table. 1).
- In D2FS, the storage device seamlessly synchronizes LBA updates to the host via the device-centric IO interface, Migration Upcall, and Virtual Overprovisioning of the filesystem partition (Figs. 11 and 17).

Contents

The artifact includes the source code of D2FS and its corresponding SSD emulator, along with the scripts for running benchmarks and generating graphs. Additionally, we provide the source code of the other filesystems and their corresponding SSD emulators evaluated in the paper.

Our artifact repository has five branches: `D2FS`, `IPLFS`, `zoned_F2FS`, and `F2FS`. The branches `D2FS`, `IPLFS`, `zoned_F2FS`, and `F2FS` correspond to the filesystems evaluated in the paper. The `F2FS-vanilla` branch is used in the experiments presented in the Fig. 1. The difference between `F2FS` branch and `F2FS-vanilla` branch lies in their discard command submission policies. `F2FS-vanilla` branch follows the original F2FS behavior, where discard commands are submitted only when the system is idle. In contrast, the `F2FS` branch aggressively submits discard commands.

The source code of each branch consists of four directories: `kernel`, `ssd_emulator`, `mkfs-tools`, and `experiment_script`. The contents of each directory are as follows:

- The `kernel` directory contains the kernel source code and the kernel compilation script, `build_kernel.sh`.
- The `ssd_emulator` directory consists of the SSD emulator source code and the script, `start_nvmevirt.sh`, for compiling and executing the SSD emulator.
- The `mkfs-tools` directory includes the source code that makes the filesystem on the disk (`mkfs`) and the compilation script for `mkfs` tools, `build_tools.sh`.
- The `experiment_script` directory contains experiment scripts, graph-generating scripts, and experimental data. The `experiment_script` directory includes directories named `FIO`, `TPC-C`, `YCSB-A`, `YCSB-F`, and `Fileserver`. These directories contain experiment scripts and experimental data for each benchmark. The `experiment_script` directory also includes a directory `graph`. The directory `graph` contains the scripts to generate experimental graphs shown in the paper.

Hosting

The artifact is available at https://github.com/ESOS-Lab/D2FS.