IPLFS: Log-Structured File System without Garbage Collection

Juwon Kim, Minsu Jang, Muhammad Danish Tehseen, Joontaek Oh, and YouJip Won, *KAIST*

https://www.usenix.org/conference/atc22/presentation/kim-juwon

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

NetApp®

IPLFS: Log-Structured File System without Garbage Collection

Juwon Kim Minsu Jang Muhammad Danish Tehseen Joontaek Oh Youjip Won
Department of Electrical Engineering, KAIST

Abstract

In this work, we develop the log-structured filesystem that is free from garbage collection. There are two key technical ingredients: *IPLFS*, a log-structured filesystem for infinite partition, and *Interval Mapping*, a space-efficient LBA-to-PBA mapping for infinite filesystem partition. In IPLFS, we separate the filesystem partition size from the physical storage size and set the size of the logical partition large enough so that there is no lack of free segments in the logical partition during SSD's lifespan. This allows the filesystem to write the updates in append-only fashion without reclaiming the invalid filesystem blocks. We revise the metadata structure of the baseline filesystem, F2FS, so that it can efficiently handle the storage partition with 2^{64} sectors. We develop Interval Mapping to minimize the memory requirement for the LBA-to-PBA translation in FTL. Interval Mapping is a three level mapping tree. It maintains mapping only for actively used filesystem region. With Interval Mapping, the FTL can maintain the mapping for the 2^{64} sector range with almost identical memory requirement with the page mapping whose LBA range is limited by the size of the storage capacity. We implement the IPLFS on Linux kernel 5.11.0 and prototype the Interval Mapping in OpenSSD. By eliminating the filesystem level garbage collection, IPLFS outperforms F2FS by up to 12.8× (FIO) and 3.73× (MySQL YCSB A), respectively.

1 Introduction

Log-structured filesystem [49] has become a popular storage management system due to its unique append-only nature. This append-only nature brings significant performance advantage against the journaling-based counterparts in practically all types of storage devices including hard disk [49, 50], flash storage [35], shingled magnetic recording (SMR) drive [44] and even persistent memory [55].

The append-only nature of the log-structured filesystem brings another dimension of complexity to the filesystem: the *garbage collection*. As the log-structured filesystem ages, the filesystem runs out of free segments and needs to reclaim the obsolete filesystem blocks to make more free segments. The activity of reclaiming the invalid blocks is called *garbage collection*. When the filesystem performs the garbage collection in the foreground, it freezes the entire filesystem till it completes [2]. The garbage collection exposes the underlying log-structured filesystem under excessive tail latency and lowers the throughput of the filesystem. The garbage collection also generates extra write traffics that increase flash wears.

A fair amount of works have been dedicated to mitigate the overhead of garbage collection in the log-structured filesystem. They include performing the garbage collection in the idle period [10, 43, 47], performing the garbage collection in a preemptive way [37], selecting the right victim segment to maximize the garbage collection efficiency, e.g. greedy, cost-benefit, age, etc [27, 41]. Some works proposed to cluster the file blocks with a similar lifespan together to improve the efficiency of the garbage collection [31, 35, 48]. The overhead of device-level garbage collection is also of serious concern. A large body of works are dedicated to mitigate the overhead of the device-level garbage collection [12, 36, 61]. When the log-structured filesystem is used with flash storage, the overhead of garbage collection may compound due to the garbage collection activities at the filesystem as well as at the device [58]. A number of works proposed to enable the host filesystem to directly manage the flash pages in the storage and eliminate the device-level garbage collection [39, 59, 60]. Recently proposed ZNS (zoned name space) treats the storage device as append-only log so that address mapping and device-level garbage collection become much simpler [5, 20]. Despite all these efforts, the root cause of garbage collection still remains neglected; the filesystem needs to reclaim the invalid filesystem blocks to make the free segments.

For several decades, the operating systems have been separating the logical entity from the associated physical entity for various types of physical resources. The typical examples include virtualizing the CPU [13], virtualizing the memory [15] or virtualizing the entire computer system [11, 17]. Unlike the other physical resources, modern operating system still tightly couples the logical storage partition from the physical storage and the size of the logical partition is bounded by the capacity of the physical storage.

In this work, we design the log-structured filesystem without garbage collection. We separate the logical partition from the physical storage and allow the filesystem to define very large logical partition independent of the capacity of the physical storage. We make the size of the logical partition large enough so that the filesystem never runs out of free segments during the lifespan of the flash storage. With very large logical partition, the filesystem does not have to reclaim the invalid filesystem blocks and therefore can eliminate the critical drawbacks of the log-structured filesystem, the garbage collection. In the absence of the garbage collection, we can greatly simplify the log-structured filesystem design.

Our work consists of two key ingredients. First is Log-structured filesystem for Infinite Partition, *IPLFS*. The second is the space-efficient LBA-to-PBA mapping, *Interval Mapping*. For IPLFS, we use F2FS as the baseline filesystem. Modern IO subsystem uses 64 bit `unsigned long` to represent the sector number. IPLFS allows that the logical storage partition for the log-structured filesystem can grow as large as 2^{64} sectors. When the filesystem writes the disk block in an append-only manner, the lifespan of flash storage is going to expire long before it reaches the end of the filesystem partition. IPLFS ensures that the number of valid blocks does not exceed the physical storage partition. IPLFS eliminates the allocation bitmap and the reverse mapping information from the log-structured filesystem. We develop *Discard Bitmap* and *Discard Logging* to discard the invalid filesystem blocks. To support prohibitively large LBA space, we develop a space efficient LBA-to-PBA mapping, Interval Mapping FTL. Interval Mapping maintains the LBA-to-PBA mapping information only for the actively used filesystem regions. The contribution of our work can be summarized as follows.

- We successfully eliminate the key complication, garbage collection, from the log-structured filesystem. We analyze the metadata structures of the log-structured filesystem and redesign the log-structured filesystem to handle very large filesystem partition.
- In the absence of garbage collection, we greatly simplify the log-structured filesystem design. We eliminate the block allocation bitmap and reverse mapping from the log-structured filesystem. We develop space-efficient and crash-safe data structures to represent the invalidated filesystem blocks that need to be discarded at the storage, *Discard Bitmap* and *Discard Log*.
- We develop space efficient mapping scheme, *Interval Mapping*. With tree-based structure, Interval Mapping maintains the mapping only for the actively used filesystem region and can handle the LBA-to-PBA mapping for 8 ZByte logical storage partition.
- We develop *fixed-region mapping* and *map node compaction* to reduce the size of the mapping tree. With map node compaction, the fixed-region mapping periodically reorganizes the structure of the map node in the Interval Mapping tree to exclude the invalidated filesystem blocks from the map node.

Via eliminating the garbage collection overhead from the log-structured filesystem, IPLFS increases the benchmark performance by up to 12.8× (FIO) and 3.7× (MySQL YCSB-A) against F2FS, respectively. With fixed-region mapping, the memory overhead of Interval Mapping is limited by the size of the physical storage, not by the logical partition size. The memory overhead of Interval Mapping is similar to that of page mapping.

2 Background

2.1 Flash Translation Layer

Flash Translation Layer is mainly responsible for three tasks: LBA-to-PBA mapping, the garbage collection, and the wear-leveling. All these three features are for hiding the physical characteristics of the flash storage media: inability to overwrite, asymmetry between the read latency and write latency and limited erase/write cycles. FTL maintains a table that holds the physical locations of the individual logical blocks in the storage device. This data structure is called *mapping table*. The size of the mapping table is proportional to the number of LBA's which it needs to map and again it is linearly proportional to the capacity of the storage device visible to the host. Page mapping maintains mapping information in page granularity [7]. While it exhibits superior random write performance [34], it suffers from excessive memory pressure for the mapping table. Block mapping maintains a mapping in the granularity of the flash block, e.g. 2 MByte. It reduces the memory pressure for the mapping table but it leaves the SSD under block thrashing [28, 34, 38]. Hybrid mapping applies block mapping for data blocks and page mapping for log blocks to reduce the mapping table size and to avoid block thrashing in the block mapping [18, 28, 29, 42, 45].

`DISCARD` (or `TRIM`) command [52] informs the SSD that a given set of blocks in the storage are no longer needed by the filesystem. It is proposed to prohibit the garbage collection module of FTL from blindly migrating the flash pages whose contents are invalid [25].

2.2 Lifespan of the Flash Storage

Flash storage can be erased and programmed only a limited number of times [23]. Table 1 illustrates the TBW (TeraBytes Written) of flash storage products released between 2019 to 2020. TBW is the amount of data that

Manufacturer	Model	Release	TBW
Adata	XPG Gammix S50	2019	3,600
Samsung	970 EVO Plus	2019	1,200
Patriot	Viper VPR100	2019	3,115
Sabrent	Rocket Q	2019	1,800
Samsung	S980 PRO	2020	600
Samsung	870 QVO	2020	2,880
T-Force	Cardea Zero Z340	2020	1,665
WD	Black SN850	2020	1,200
SK hynix	Gold P31	2020	750

Table 1: TBW(terabytes written) of the SSD products

Dataset	Write volume (GB/day)	Time to exhaust address space (M year)
YCSB SSD [56]	1,159	20
Systor [51]	57	422
Nexus 5 [21]	13	1,853

Table 2: Daily write volume and estimated SSD lifespan, traces are from SNIA open dataset [4]

can be written to the SSD over the lifespan of the drive. In Table 1, the largest TBW corresponds to 3.6 PByte.

The actual amount of data that is written to the storage device is much smaller than what the storage device can sustain. We compute the per-day write volume from the real IO traces [4](Table 2). They correspond to IO traces in the Key-value storage [56], SYSTOR [51], and smartphone [21]. Key-value storage engine generates the largest amount of write among the three traces; it writes 1.13 TByte per day to the storage.

The modern OS uses 64 bit, `unsigned long` type, variable to represent the sector number (LBA). With `unsigned long` type variable, the host can create the logical partition of 2^{64} sectors. The size of the logical partition corresponds to 8 ZByte. With 8 ZByte logical storage partition, the log-structured filesystem can keep appending the updated blocks at the end of the active filesystem region without reclaiming the invalid filesystem blocks. The lifespan of the underlying SSD will expire long before the filesystem reaches the end of the logical partition of 8 ZByte. For YCSB-A workload in Table 2, it will take 20 million years to exhaust the 8 ZByte filesystem partition.

2.3 F2FS, a log-structured file system

F2FS is the log-structured filesystem specifically developed for the flash storage. Despite the promising characteristics, the preceding log-structured filesystems [49, 50] have failed to be widely deployed in the commodity hardware with the prime cause for the failure being the overhead of reclaiming the invalid blocks. F2FS is the first log-structured filesystem that has gained the publicity successfully. It is the default filesystem for wide variety of Android devices ranging from smartphone to automotive [1].

F2FS divides the filesystem partition into two areas: metadata area and data area. F2FS updates the contents in the metadata area in an in-place manner and writes the data area in an append-only manner. The metadata area contains the filesystem metadata such as a superblock, a checkpoint pack, a block allocation bitmap for each segment (a segment information table, SIT), and reverse mapping information (the file id and the file offset) for each segment (segment summary area, SSA). To avoid the wandering tree problem of the out-of-place update associated with the log-structured filesystem [8], F2FS clusters the file index blocks for all files in the filesystem together in the metadata area (a node address table, NAT) and updates it in an in-place manner. F2FS organizes the data area as a set of zones. A zone consists of a set of sections and a section consists of a set of segments. The segment is the unit of disk write and the section is the unit of garbage collection. In most (if not all) deployment of F2FS, a zone and a section consist of a single segment.

F2FS defines two block types (node and data) and three hotness levels (hot, warm, and cold) to represent the update frequency of a filesystem block. The node block corresponds to the inode or the index block of the file. There are total six combinations of block type and hotness level pair. F2FS maintains the six active segments in memory for each combination. F2FS places the blocks of the same type and temperature at the same active segment. When the active segment is full, it is flushed to the storage device. F2FS clusters the filesystem blocks with the same type and hotness level together at the flash storage. This is to reduce write-amplification caused by the device level garbage collection.

F2FS reclaims the invalid filesystem blocks either when the filesystem is idle (background garbage collection) or when there runs out of free segments (foreground garbage collection). In background and foreground garbage collection, F2FS uses cost-benefit policy [41] and greedy policy [27], respectively, in selecting the victim section.

For crash recovery, F2FS reads the most recent checkpoint pack from the disk and recovers the filesystem state with respect to the time of the most recent checkpoint. Then, F2FS scans the node area of the filesystem, and identifies the files that have been made durable via `fsync()` after the most recent checkpoint. For each such file, F2FS compares the node block at the time of checkpoint and the node block synchronized to the disk via `fsync()` and identifies the newly allocated blocks and the invalidated blocks in the file. For the newly allocated blocks, F2FS updates the associated filemap. For invalidated blocks, F2FS invalidates the block allocation bitmap in the segment information table in memory. After reconstructing the allocation bitmap, the recovery

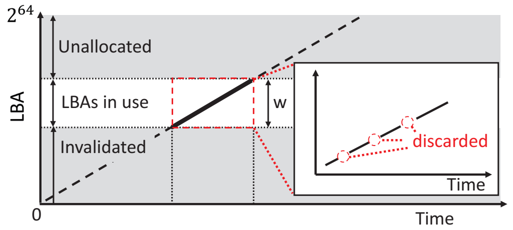


Figure 1: Active Region w in the logical partition

module creates the discard commands for the invalidated blocks.

When a filesystem operation invalidates one or more blocks, e.g. `truncate()`, or `unlink()`, F2FS updates the associated bitmap in the segment information table. F2FS checkpoints the filesystem state either periodically or when the garbage collection is triggered. In checkpoint, F2FS constructs the discard commands for the invalidated filesystem blocks. When the checkpoint finishes, F2FS wakes up the discard thread. The discard thread dispatches the discard commands in a regular interval (default = 50 msec). It limits the number of dispatch commands that are sent at a time (default = 8). It dispatches the discard command only when the system is idle.

3 Design Overview

3.1 Design Philosophy

The fundamental design philosophy behind IPLFS is the *separation of the logical storage partition from the physical storage*. The log-structured filesystem [49] offers natural ground to separate the logical partition from the physical partition due to its level of indirection inherent in the out-place update filesystem. In the legacy in-place update filesystems, a file block is bound to the fixed location in the physical storage when the file block is allocated. It is non-trivial to separate the logical partition size from the physical partition size [24]. On the other hand, the log-structured filesystem dynamically updates the file mapping information to keep track of the location of the most recent version of the file block.

The existing log-structured filesystem [32, 35] limits the size of the logical partition to the size of the associated storage device and reclaims the invalid filesystem blocks when it runs out of free blocks in the logical partition. In this work, we set the size of the logical partition large enough so that there is no lack of free LBAs during SSD's lifespan. Fig. 1 visualizes the usage pattern of the log-structured filesystem in the very large logical partition. X and Y axes denote time and LBA, respectively. As the filesystem ages, file blocks get invalidated. The window of actively used filesystem region, w, moves towards the higher end of its filesystem partition. Actively

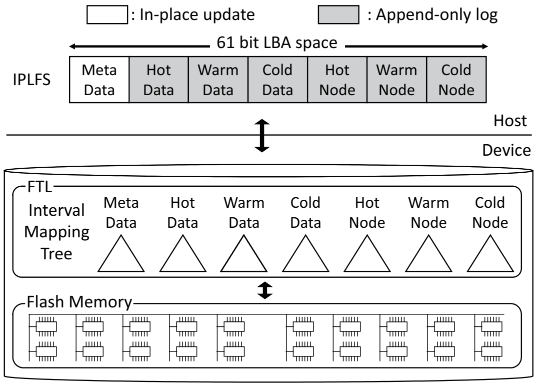


Figure 2: Concept: Log-structured filesystem for Infinite Partition, IPLFS, and Interval Mapping

used filesystem region starts at the lowest valid LBA and ends at the highest valid (allocated) LBA. Within the actively used filesystem region, w, some blocks are invalid and discarded at the storage device. When the size of the logical partition is very large, only a small fraction of the logical partition, w, is being accessed by the filesystem. The storage controller needs to maintain LBA-to-PBA mapping only for the actively used filesystem region, w.

3.2 Organization

Fig. 2 illustrates the main components of our system. Eliminating the garbage collection in the log-structured filesystem is achieved by two ingredients: Log-structured filesystem for Infinite Partition, *IPLFS* and FTL for very large logical partition, *Interval Mapping*. The first component is IPLFS. IPLFS uses F2FS as a baseline filesystem. The in-memory and on-disk structures of F2FS are carefully trimmed and modified so that it can handle the logical partition of 2^{61} blocks and that it can dispense with filesystem level garbage collection.

The second component is a space-efficient FTL, *Interval Mapping*. In most existing LBA-to-PBA mapping techniques, the number of entries in the mapping table corresponds to the number of blocks in the logical storage partition. These techniques become practically infeasible due to its prohibitive mapping table size when it needs to map 2^{61} blocks. Interval Mapping maintains an LBA-to-PBA mapping only for the actively used region in the logical storage partition. Interval Mapping is multi-level tree based mapping. By using the multi-level mapping, Interval Mapping tries to avoid allocating the mapping table entries for the invalid filesystem blocks.

4 IPLFS

IPLFS never recycles the blocks in the filesystem partition. This very nature enables IPLFS to dispense with garbage collection at the filesystem layer and yet can

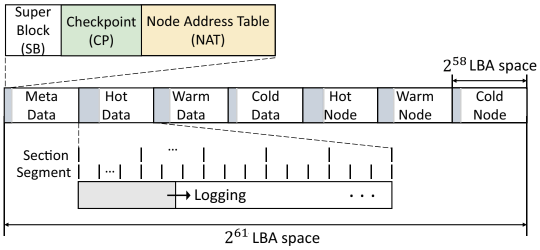


Figure 3: Multi-area Partition of IPLFS

maintain its append-only update nature. IPLFS consists of three key design ingredients: (i) multi-area partition layout, (ii) garbage collection-less metadata design, and (iii) discard map and discard logging.

4.1 Multi-area Partition Layout

We partition the entire filesystem partition of IPLFS into seven areas of the same size (Fig. 3). One area (the first one) is used for hosting the metadata of IPLFS. It holds the filesystem metadata information such as superblock and the node address table. Existing log-structured filesystems treat the filesystem partition as a single log [49, 50] or two logs [35]. IPLFS has six logs each of which accommodates the filesystem blocks of the same type and hotness level. Via clustering the filesystem blocks with similar update frequency together, IPLFS maintains the size of the actively used filesystem region small. We use MSB 3 bits of the LBA address as the area identifier. The size of each area is 2^{58} blocks, 1 ZByte.

4.2 Metadata Design

We carefully design the metadata structure of IPLFS so that it can handle the very large filesystem partition. Particular care has been taken to minimize any changes in the on-disk layout of its baseline filesystem, F2FS. Log-structured filesystem provides two essential metadata: reverse mapping and block allocation bitmap. These data structures have two main usages: the filesystem level garbage collection and the block discard. These two data structures are no longer used for the filesystem-level garbage collection because IPLFS does not perform garbage collection. IPLFS cannot use the reverse mapping and block allocation bitmap for block discard purpose, either, due to the prohibitively large logical partition. The size of the block allocation bitmap and the size of the reverse mapping information are linearly proportional to the size of the filesystem partition. Given the filesystem partition of 2^{64} sectors, 8 ZByte, the size of the block allocation bitmap and the reverse mapping corresponds to 512 PByte and 8 EByte, respectively. IPLFS cannot afford the storage space for block allocation bitmap and reverse mapping.

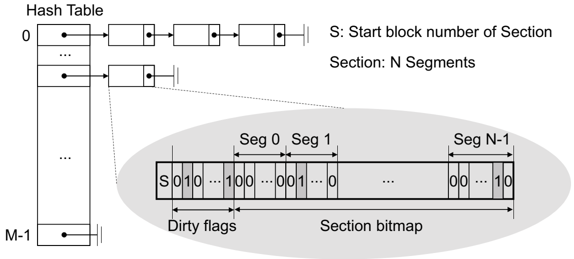

Hash Table
0
...
M-1
S: Start block number of Section
Section: N Segments
Seg 0
Seg 1
Seg N-1
S 0 1 0 ··· 1 0 0 ··· 0 0 1 ··· 0 ··· 0 0 ··· 1 0
Dirty flags
Section bitmap

Figure 4: Discard Bitmap

IPLFS retains the node address table (NAT) in F2FS as is. The number of entries of the node address table corresponds to the maximum number of inodes in the filesystem partition. The number of inodes in the filesystem is limited by the capacity of the physical storage, not by the size of the logical filesystem partition. The size of node address table does not increase even though the size of the logical partition is very large.

In IPLFS, we remove the block allocation bitmap (Segment Information Table) and reverse mapping information (Segment Summary Area) from its baseline filesystem, F2FS and develop a new metadata structure for discarding the filesystem blocks, *Discard Bitmap* and *Discard Log*.

4.3 Discarding the Invalid Blocks

The log-structured filesystem maintains the block allocation bitmap for two reasons: to represent the space utilization of the individual segments and to keep track of the newly invalidated filesystem blocks. Former is for the filesystem-level garbage collection purpose and the latter is for discarding the filesystem blocks. In IPLFS, the former reason for maintaining the allocation bitmap disappears but the latter reason remains outstanding.

Eliminating the block allocation bitmap, we develop a new data structure, *discard bitmap*, that represents the newly invalidated filesystem blocks since the last checkpoint. IPLFS maintains the discard bitmap in per-section basis. In IPLFS, a section consists of more than one segments. A discard bitmap consists of two components: the start LBA of the section and the bitmap itself. When the filesystem invalidates the block, it sets the associated discard bit at the discard bitmap. IPLFS organizes a set of the discard bitmaps using the hash table. Fig. 4 illustrates the structure of the set of discard bitmaps. M and N correspond to the number of hash buckets in the hash table and the number of segments in a section. The hash table uses the section number as a hash key.

When a filesystem block is invalidated, IPLFS searches the hash table for the associated discard bitmap. If the discard bitmap is found, IPLFS updates the discard bitmap with the newly invalidated block. If the associated discard bitmap does not exist, IPLFS allocates the discard bitmap for the section which the newly invalid block belongs to, and sets the associated bit of the block that needs to be invalidated. Then, IPLFS inserts the newly created discard bitmap at the hash table.

There is a trade-off between the section size and the filesystem performance. With a larger section size, the hash table for the discard bitmaps becomes larger. With a smaller section size, there exists more discard bitmaps in the hash table and the latency for searching the hash table becomes longer. Through experiment, we find that the section size of 1 GByte renders the reasonable balance between the filesystem performance and the memory pressure. In the later part of this paper, the section size is set as 1 GByte, i.e. 512 segments.

In each checkpoint, IPLFS scans the hash table and constructs the discard commands for each discard bitmap. After constructing the discard commands, it removes the discard bitmap from the hash table. IPLFS issues the discard commands periodically, e.g. in 50 msec interval. As in F2FS, IPLFS allocates a separate thread for dispatching the discard commands. IPLFS issues the discard commands in a more aggressive fashion than F2FS does. IPLFS dispatches the discard commands no matter whether there is a pending I/O or not. In F2FS, the dispatch thread issues the discard commands only when there is no pending I/O. In IPLFS, the dispatch thread issues up to sixteen discard commands each time when it wakes up. F2FS takes particular care to prohibit the discard command from interfering with the foreground IO requests [3]. We find in our platform (OpenSSD), the aggressive discard policy renders better benchmark performance since it makes the SSD garbage collection more efficient and reduces the write amplification.

4.4 Discard Logging

In the absence of the block allocation bitmap, IPLFS is subject to the *Storage Leak*. Storage Leak denotes the situation where the flash page contains invalid filesystem block and the filesystem never reclaims the associated flash page. Assume that the system crashes while there are outstanding discard commands. As a result of the system crash, the outstanding discard commands are lost. In F2FS, the recovery routine creates the discard commands based upon the recovered allocation bitmap. Unlike F2FS, IPLFS does not have allocation bitmap and cannot reconstruct the discard commands that are lost due to crash. The flash pages associated with the lost discard commands will remain valid permanently even though they will no longer be used.

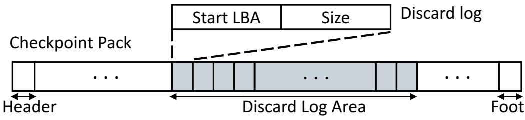


Figure 5: Checkpoint Pack with Discard Logs

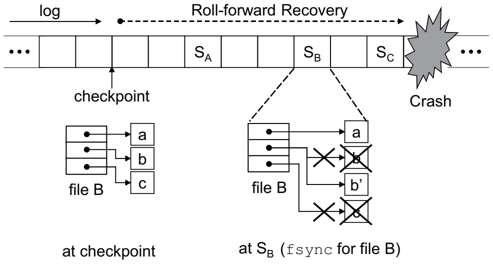


Figure 6: Roll-forward Recovery in IPLFS

To save IPLFS from Storage Leak, we develop a mechanism called *Discard Logging*. In Discard Logging, IPLFS checkpoints the information associated with the discard commands prior to issuing the discard commands to the storage. With discard logging, IPLFS guarantees that discard command is made durable at the storage before it is issued to the storage. With Discard logging, IPLFS can recover the outstanding discard commands when the system crashes unexpectedly.

IPLFS allocates a certain region, *Discard Log Area* at the checkpoint pack (Fig. 5). At each checkpoint, IPLFS scans the discard bitmap and creates the discard commands. After it finishes creating the discard commands, IPLFS logs the information associated with the discard commands, [startLBA, Length], at the discard log area of the in-memory checkpoint pack. After it finishes preparing the checkpoint pack, it synchronizes the checkpoint pack to the disk. After the checkpoint, IPLFS wakes up the discard thread for issuing the discard commands.

When the system crashes, IPLFS recovers the discard commands in two phases. In roll-backward recovery, the recovery module reads the most recent checkpoint pack and reconstructs the discard commands with respect to the discard logs. In roll-forward recovery, IPLFS identifies the fsynced files after the most recent checkpoint. IPLFS compares the node block that is found at the roll-forward recovery phase and the node block at the time of the checkpoint. IPLFS then identifies the changes in the block allocation and updates the filemap with respect to the newly allocated node blocks. Based upon the difference on the block allocation, IPLFS identifies the invalidated filesystem blocks and constructs the discard commands for the invalidated blocks. Fig. 6 illustrates how IPLFS reconstructs the discard commands. At the time of the checkpoint, the node block of file B consists of three file blocks, a, b and c. After the checkpoint, the file block b is updated to b' and file block c is truncated in file B. Then, file B is synchronized to disk through

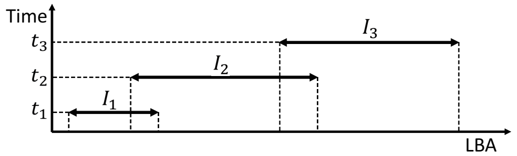


Figure 7: active region of the filesystem, I_j: active region at time t_j

`fsync()`. After `fsync()`, node block of file B refers to two blocks, a and b'. Block b' is newly allocated and block b and block c are invalidated. In roll-forward recovery, IPLFS updates the filemap of file B to refer to a and b' and creates the discard command for discarding b and c.

5 Interval Mapping

5.1 Design

We develop a space-efficient LBA-to-PBA mapping, called *Interval Mapping*. It is similar to interval tree [16] in that each leaf node has an interval of LBA's associated with it. Unlike interval tree, the height of the interval mapping tree is fixed to three. Limiting the height of the tree, the interval mapping increases the fan-out degree of the root node to accommodate the new leaf nodes. In designing the LBA-to-PBA mapping, we exploit the fact that in IPLFS, the actively used filesystem region moves towards the higher end of the logical storage partition as the filesystem ages. Fig. 7 illustrates how the active region moves with time. At t_1, the active region corresponds to I_1. At t_2, the active region corresponds to I_2.

In IPLFS, there are 2^{61} blocks in the logical storage partition. With 16 KByte flash page size, the page mapping table size for this storage partition corresponds to $4EByte$. None of the existing mapping techniques such as block mapping [18], hybrid mapping [29,34,38,40,46], or superblock-based mapping [26] can reduce the mapping table size for LBA space of 2^{61} blocks to a manageable one. Interval Mapping addresses the prohibitive mapping table size requirement in IPLFS. Flash storage for IPLFS uses the page mapping for the metadata area and Interval Mapping for each of six data areas. Storage controller identifies the interval mapping tree for the incoming LBA using the most significant three bits of LBA.

Interval Mapping is organized as three level tree (Fig. 8). We limit the height of the mapping tree to three to reduce the number of memory accesses associated with the address translation. Interval Mapping organizes a storage area as an array of *zones*. The zone is an array of *mapping segments*. The size of the zone and the size of mapping segment correspond to 16 GByte and 16 MByte, respectively.

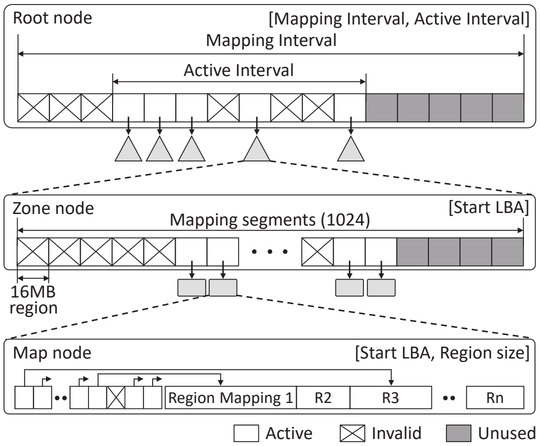


Figure 8: Structure of Interval Mapping Tree

A root node has a number of *zone nodes* as children. The sub-tree rooted at each zone node maintains a mapping for a single zone. When the interval tree is first created, the fan-out degree of the root node is set to 32. We increase the root node size to accommodate more child nodes when it is necessary. The maximum size of the root node is currently set to 1 MByte. With 1 MByte root node, the root node can have 2^{18} child nodes and can map up to 4 PByte of logical storage partition. It can be increased if the root node needs to map larger LBA region.

A single Zone node has 1024 *Map Nodes* as its child nodes. A map node maintains the LBA-to-PBA mapping for a single mapping segment. In map node, we avoid using plain table based mapping structure. Instead, for the compact mapping organization of the map node, we develop a new technique, *fixed-region mapping*, for the LBA-to-PBA mapping.

5.2 Mapping Interval and Active Interval

Interval Mapping defines two important concepts associated with mapping: the *Mapping Interval* and the *Active Interval*. Mapping Interval is a region of the logical partition that the interval mapping tree needs to map. Mapping interval is represented by the start LBA of the first zone and the start LBA of the last zone in the mapping interval, respectively. When the storage partition is created, Interval mapping creates six interval mapping trees for individual data areas of the IPLFS filesystem partition. Mapping interval is initialized when the mapping tree is first created. The start LBA of each mapping interval corresponds to the first LBA of associated filesystem area. Initially, each mapping interval consists of thirty-two zones, 512 GByte.

Active Interval is a window of actively used zones within the associated mapping interval. Active interval

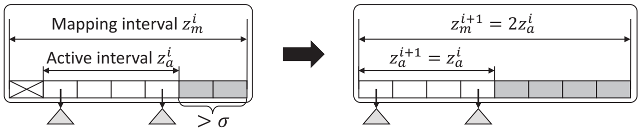


Figure 9: Updating the Mapping Interval

is similar to the active region in Fig. 7. The active interval starts at the first valid zone of the mapping interval and ends at the last valid zone of the associated mapping interval. If all filesystem blocks in a zone become invalid, the zone becomes invalid. The start of the active interval is updated to the following valid zone in the active interval when the first zone of the active interval becomes invalid. The end of the active interval is extended to the newly allocated zone if the new zone is appended to the active interval to accommodate more blocks.

As the filesystem ages, active interval moves towards the higher end of the logical partition. When the end of active interval nearly reaches the end of the mapping interval and there is little room to grow, Interval Mapping creates the new root node with the new mapping interval which can better accommodate the current active interval. The details of updating the mapping interval are as follows. First, we compute the mapping interval for the newly created root node. If the length of the current active interval is less than 1024 zones, the length of the new mapping interval is set as twice the length of the current active interval. Otherwise, it is initialized as the length of the current active interval plus sixteen zones. Second, we allocate the new root node with updated mapping interval. Third, we copy the child pointers of the old root node to the new root node.

The start of the mapping interval of the newly created root node is initialized to the start of the current active interval. The active interval of the new root node inherits the current active interval. Fig. 9 illustrates an example of creating the new root node with updated mapping interval. σ is the minimum number of free zones which the interval tree needs to maintain. If the number of free zones in the mapping interval becomes less than σ, Interval Mapping updates the mapping interval creating the new root node. For an interval mapping tree, the mapping interval can be updated multiple times. When the mapping interval is updated, IPLFS allocates the new root node each time. Let i be the number of times when the mapping interval is updated for the associated mapping tree. z_m^i and z_a^i denote the number of zones in the mapping interval and the number of zones at the active interval of the i^{th} version of the root node, respectively. The new mapping interval starts at the same zone as the start of the current active interval. The length of the new mapping interval is twice the length of the current active interval, $z_m^{i+1} = 2z_a^i$. The length of the new active interval is the same as the length of the current active interval, $z_a^{i+1} = z_a^i$.

FTL creates the new root node and updates the mapping interval in non-blocking way so that it can minimize the interference with the foreground IO request for address translation or for allocating the new zone. When a new zone node needs to be inserted at the root node while the new root node is being created, the newly created zone node is appended at the new root node and the active interval of the new root node is updated accordingly. For address translation, FTL uses the old root node if the incoming LBA belongs to the active interval of the old root node. Otherwise, it uses the new root node for address translation.

5.3 Fixed-Region Mapping

A Map Node maintains the mapping for single mapping segment, 16 MByte by default. The total size of the map nodes accounts for 99.9% of the entire mapping tree. It is critical that map node data structure is carefully designed to minimize the memory requirement as well as the mapping latency in Interval Mapping. To address the two objectives, we develop a new mapping technique called *fixed-region mapping*. Fixed-region mapping partitions a mapping segment into the same size *regions* with a given *region size*. Map node maintains the LBA-to-PBA mapping in per-region basis. Fixed-region mapping allocates the mapping table only for the valid regions, i.e. the region that has one or more valid blocks. Map node maintains a region directory which has the location of the per-region mapping tables. If the region is invalid, the associated entry in the region directory is NULL. To reduce the size of mapping table, each mapping table specifies the start LBA of the active region and excludes the mapping for invalid blocks at the beginning of the associated region.

The *region size* plays a key role in the mapping efficiency of the map node. Mapping efficiency is the ratio of the number of the valid mapping entries against the total number of mapping entries. As the region size gets smaller, the mapping segment is partitioned into smaller regions and the number of invalid regions is likely to increase. As the region size becomes smaller, the mapping efficiency improves but the region directory becomes larger. As the region size becomes larger, the mapping segment consists of smaller number of regions. With larger size region, the region directory becomes smaller but the mapping efficiency becomes worse. We need to find the right region size that can maximize the mapping efficiency and minimizes the map node size. Interval Mapping sets the region size as *the size of the smallest hole in the mapping segment.* To avoid that the region size becomes too small, we set the minimum region size, 256 KByte. When the region size corresponds to the size of the smallest hole, it is guaranteed that there can be

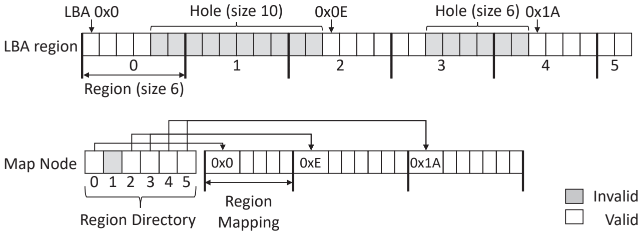


Figure 10: Mapping Segment and Map Node

only one active region in the associated region. Active region is a consecutive array of valid blocks in a region.

A map node consists of the three components: the range of the mapping segment, the region directory, and the array of mappings for individual regions. The number of entries in the region directory corresponds to the number of regions of the map node. The mapping information for each region consists of the start address of the active region and associated LBA-to-PBA mapping. Fig. 10 illustrates the mapping segment and the organization of the associated map node. There are 32 blocks in the mapping segment. There are two holes with 10 blocks and 6 blocks, respectively. The region size of this map node is set to 6. Map node partitions the mapping segment into six regions. In Region 0, there are four valid blocks. Region 1 does not have any valid blocks. The directory entry for region 1 is NULL since it does not have any valid blocks. There are three active regions in the mapping segment. The second active region spans across region 2 and region 3. The map node allocates a single region map for the active region that spans region 2 and region 3.

Interval Mapping periodically reorganizes the map node. We call it *map node compaction*. It updates the region size for the mapping segment and reconstructs the per-region mapping tables with respect to the updated regions. This is to reduce the map node size by eliminating the mappings for the invalid flash pages. When the map node is first created, the region size is set to size of the mapping segment. When the FTL invalidates the mapping table entry at the map node, it examines the mapping efficiency. If the mapping efficiency becomes smaller than a certain threshold, (50%), the FTL inserts the map node to the compaction candidate list. The compaction thread periodically scans the compaction candidate list (default 30s), and estimates the size of the reorganized map node with the updated region size. If the map node can become smaller by more than 30% after compaction, FTL reorganizes the map node. Fig. 11 illustrates the map node compaction. Before the compaction, the original map node has a single region that has 16 mapping entries and three active regions. There are two holes of four blocks and five blocks. For compaction, the region size is updated to four (the minimum hole size). The mapping region is partitioned into four regions based upon the new region size, four. After compaction, the map node has three region mappings, each of which includes two, two and three mapping entries, respectively. As a result of map node compaction, the map node size decreases by approximately half.

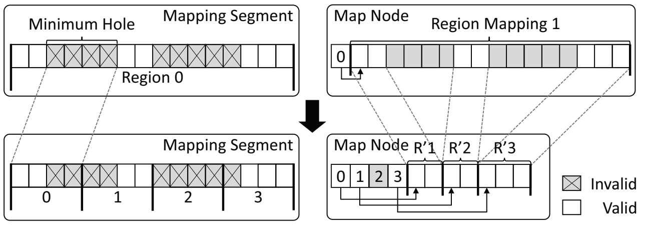


Figure 11: Reorganizing the Map Node

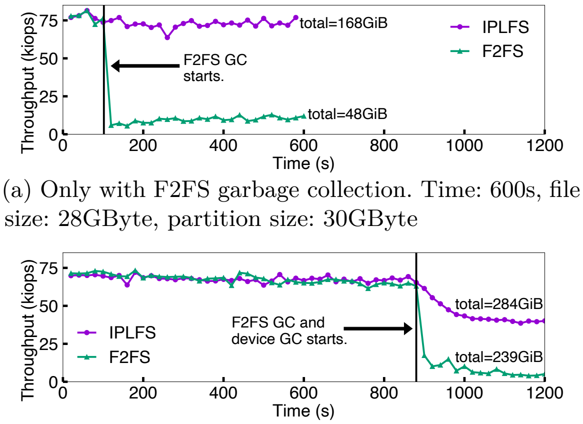
The memory overhead of Interval mapping is almost the same as the memory overhead for page mapping. Assume that the storage size is 512 GByte and flash page size is 16 KByte. The page mapping consists of a 4 KByte block bitmap (for subpage mapping [30]) and a mapping table. The page mapping requires 144 MByte (16 MByte for bitmap and 128 MByte for mapping table). The interval mapping consists of a root node, 32 zone nodes, and 2^{15} map nodes with sizes 128Byte, 4096Byte, and 4624Byte, respectively. The size of Interval Mapping corresponds to 144.6 MByte.

6 Evaluation

We implement IPLFS on F2FS (Linux 5.11.0) and Interval Mapping on OpenSSD (230GByte, 8 channels) [33], respectively. A default FTL in OpenSSD uses page mapping and maps LBA to PBA of different channels in a round-robin way. So does Interval Mapping FTL. We use a PC server with Intel CPU i7-4770K (3.50GHz, 4 cores), and 8 GByte DRAM for the experiment.

6.1 Eliminating the Garbage Collection

FIO 1. We examine the performance benefit of eliminating the filesystem-level garbage collection. We use FIO [6]. The logical storage partition is 30 GByte. In this experiment, four threads perform random write on 28GByte file. We measure the throughput in every 2 sec. Since the F2FS partition is almost full at the beginning of the experiment, it quickly runs out of free segment. On the other hand, the logical partition size is set to be much smaller than the physical storage size. This is to prohibit the storage device from running FTL garbage collection. The effect of eliminating the filesystem level garbage collection is substantial. In Fig. 12a, the performance of F2FS drops to 1/10 when it starts to perform garbage collection. IPLFS performance remains steady at its full speed till the experiment finishes.



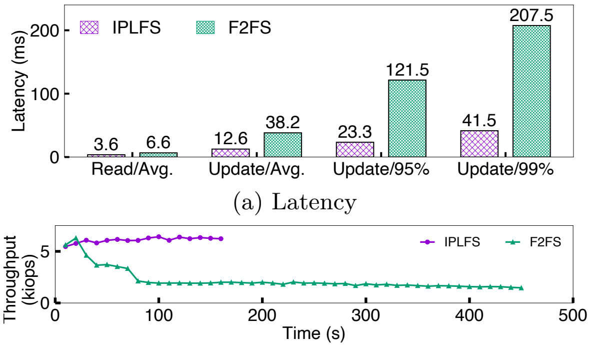
(a) Only with F2FS garbage collection. Time: 600s, file size: 28GByte, partition size: 30GByte

(b) F2FS garbage collection and FTL garbage collection. Time: 1200s, file size: 210GByte, partition size: 230GByte.

Figure 12: FIO (random write) Throughput: IPLFS vs. F2FS. The number indicates the total write volume.

FIO 2. We examine the performance impact of filesystem-level garbage collection as well as device-level garbage collection. Fig. 12b illustrates the result. We set the size of the logical partition to 230 GByte, which is the physical storage capacity of OpenSSD. We perform the random write on 210 GByte file. When F2FS starts garbage collection, the throughput decreases to nearly 1/10. While IPLFS is free from filesystem level garbage collection, the underlying flash storage is not. When OpenSSD starts device-level garbage collection, the FIO performance of IPLFS decreases to 60%. The filesystem-level garbage collection bears more significant impact on the benchmark performance than the device-level garbage collection does.

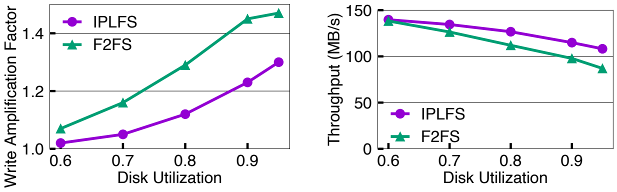
MySQL. We run YCSB-A workload with MySQL and examine how database operations are interfered by the filesystem garbage collection. YCSB A workload [14] consists of the same amount of reads and updates. To quickly trigger the filesystem level garbage collection, the filesystem is 90% full at the beginning of the experiment. Fig. 13a illustrates the average read and update latencies of IPLFS and F2FS, respectively. The average read latency and the average update latency of IPLFS are 1/2 and 1/3 of those of F2FS, respectively. The absence of garbage collection improves the tail latency of the filesystem significantly. The tail latencies of update at 95% and at 99% in IPLFS are 5.2× and 5× lower than those of F2FS. Fig. 13b illustrates the throughput. IPLFS's throughput remains steady throughout the experiment. F2FS renders the similar performance to IPLFS at the beginning but the performance decreases substantially when it starts running the garbage collection.



(a) Latency

(b) Throughput

Figure 13: MySQL latency and throughput: YCSB-A, record size: 1KB, record count: 5M, operation count: 1M (read:update=1:1), threads: 50, partition size: 18GB.



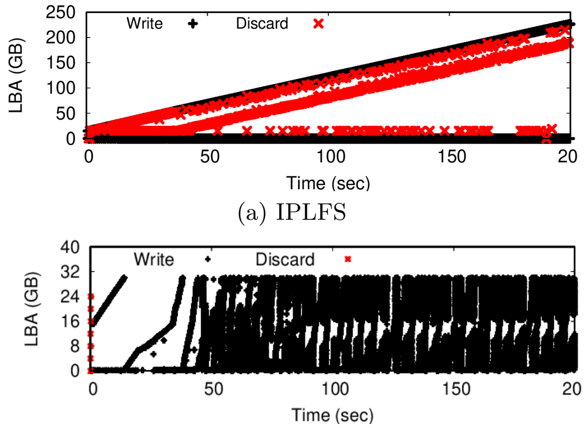
(a) WAF

(b) Throughput

Figure 14: IPLFS vs. F2FS: fileserver benchmarks under varying disk utilization, file size: 2 MByte, partition size: 230GByte

6.2 Discard Policy of IPLFS

We examine how the more aggressive discard policy affects the FTL garbage collection and the application performance. We use fileserver workload in Filebench [53], where 50 threads create, update and delete 2 MByte files. Fig. 14a shows the write amplifications in IPLFS and F2FS. IPLFS exhibits lower write amplification than F2FS in all disk utilizations. Fig. 14b depicts the throughput under varying disk utilization. IPLFS improves the



(a) IPLFS

(b) F2FS

Figure 15: Block trace: Filebench fileserver workload, Partition size: 30GB

throughput by as much as 24% against F2FS. Aggressive discard policy of IPLFS saves the FTL garbage collection from migrating the invalid filesystem blocks. As a result, IPLFS renders substantial improvement in write amplification and the benchmark performance.

We examine the IO traces in IPLFS and F2FS, respectively. In IPLFS and F2FS, the logical partition sizes correspond to 1 ZByte and 30 GByte, respectively. Fig. 15a and 15b illustrate the results. IPLFS never recycles the filesystem blocks and keeps appending the blocks throughout the experiment. On the other hand, F2FS recycles the invalid filesystem blocks in round-robin manner. We observe that IPLFS issues the discard command a lot more frequently than F2FS does. This is because F2FS issues discard command only when there is no pending IO. In this experiment, F2FS rarely finds that there is no pending IO.

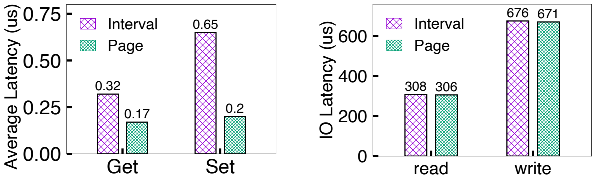
6.3 Address Translation Overhead

Address Translation Latency. We examine the overhead of address translation in Interval Mapping and page mapping (Fig. 16a). We run FIO with four threads. They perform random write on 10 GByte file. Interval Mapping yields 88% longer mapping latency than the page mapping. This is because Interval Mapping performs multiple index lookups for address translation. When creating a new mapping entry, Interval Mapping exhibits 3.3× longer latency than the page mapping.

End-to-end Latency. We measure the micro-benchmark performance under Interval mapping and the page mapping. Fig. 16b shows the latencies of read and write (direct IO) in FIO benchmark. The read latency and the write latency of Interval Mapping and page mapping are almost identical. This result shows that the overhead of accessing the NAND flash and the overhead of transferring the data blocks between the host and storage device account for dominant fraction of IO latency and FTL overhead is not significant.

6.4 Map Node Size

We examine the memory overhead of fixed-region mapping. We run fileserver workload with 50 threads. Each thread creates, updates and deletes 2 MByte files.



(a) FTL mapping latency (b) Direct I/O latency.

Figure 16: FTL Overhead: Interval Mapping vs. page mapping

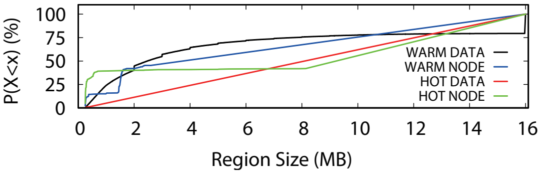


Figure 17: CDF of region size

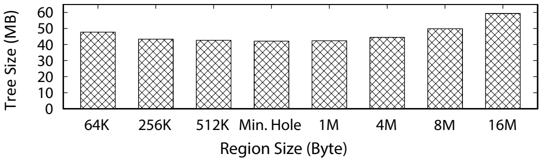


Figure 18: Mapping Tree size under varying region sizes

Region Size. We examine the IO volume associated with write and discard. We also examine the size of holes for each filesystem area (Table 3). The median hole size varies widely subject to the area type. Median hole size of the warm data area is 2860 KByte while that of the hot data area is 8 KByte. IO's for warm data area (write and discard) account for dominant fraction of all IO's (99% of write, 99% of discard).

We examine the region size distribution (Fig. 17). Region size is set as the size of the smallest hole in the associated mapping segment. For the warm data are, 3/4 of the region sizes are greater than 1 MByte. Subsequently, most of the map nodes have sixteen or less number of regions. Recall that the size of mapping segment is 16 MByte. The size of region directory accounts for approximately 1% of the map node size. In map node design, the size of the region directory is negligible.

Minimum Hole Size for region size. We compare the sizes of the mapping trees when the region size is fixed and when the region size is chosen dynamically to the size of the smallest hole in mapping segment. Fig. 18 shows results. The mapping tree becomes the smallest when we use the the minimum hole size as the region size. For small region size (64 KByte), the tree size is 13% larger than the tree size when we use the minimum hole size as region size. This is due to the increase in the region directory size. In a map node with 64 KByte region size, the region directory accounts for 13% of the total size of the mapping tables. The mapping tree size becomes larger when we use large fixed region size

Log type	W/D volume (GB)	Median (KB)	75% (KB)	90% (KB)
Warm Node	1.99 / 1.69	32	32	80
Warm Data	305.9 / 170.4	2860	6352	11764
Hot node	0.4 / 0.38	48	232	816
Hot Data	0.87 / 0.36	8	24	40

Table 3: Statistics on the hole size, Filebench fileserver workload. Average file size: 2MByte, runtime: 1600s, W/D volume: total volume of Write/Discard

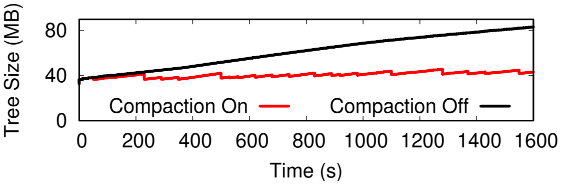


Figure 19: Mapping Tree size

(1 MB or larger) than when we use the minimum hole size as a region size. This is because with the larger region size, the fixed-region mapping fails to exclude the mappings for invalid flash pages and the mapping efficiency becomes worse.

Reorganizing the Map Node. We examine the effectiveness of map node compaction. We configure the compaction period to 30s and a compaction ratio threshold to 0.7. Total size of the files is 160GByte. When the benchmark finishes, the active interval in the filesystem partition corresponds 305GByte. Fig. 19 illustrates the sizes of mapping trees in two cases: when the Interval Mapping periodically reorganizes the map node and when it does not. Without compaction, the mapping tree size increases as the filesysem ages and reaches over 80 MByte. This is because the active interval becomes larger and the Interval Mapping allocates new map nodes as the active interval expands. With map node compaction, the mapping tree size stays at around 40 MByte. Mapping table size remains almost the same as the mapping table size for 160 GByte even though the active interval expands to 305 GByte.

7 Related Works

IO stack largely consists of two layers: the host (filesystem and block device layer) and the device (SSD). A body of works try to migrate the FTL overhead from the storage device to the host. They include DFS [24], ParaFS [60], Application Managed Flash [39], and OrcFS [59]. In these works, the software overhead at the host side increases; the host side software directly manages the flash pages and performs essential managerial activities such as garbage collection and wear-leveling. Nameless Write eliminates the address translation layer from the IO stack [62]. Migrating the device's functionality to the host has its cost. Flash storage needs expose its physical nature, e.g. physical flash page location, page size or block size, to the host. ZNS saves the device from the FTL overhead [9, 20]. On the contrary to these works, IPLFS aims at reducing the host side overhead, the filesystem level garbage collection with reasonable increase in the device firmware complexity.

IPLFS shares the same idea with DFS [24] in that IPLFS separates the logical filesystem partition from the physical storage capacity and fully exploits 2^{64} logical partition size. Despite of the similarity, the underlying philosophies and the approaches of the two lie at the other ends of spectrum. DFS migrates the garbage collection from the device to the host whereas IPLFS migrates the garbage collection from the host to the device. DFS introduces new indirection layer to separate the logical partition from the physical storage. Host side's IO stack becomes heavier to handle LBA-to-PBA mapping, garbage collection, and etc. It requires the flash device to expose physical details to the host. IPLFS does not require new layer nor any physical information of the flash storage. Separating the logical partition from the physical storage, IPLFS becomes simpler and lighter-weight than its original counter part, F2FS.

A number of works proposed to make the garbage collection more efficient. Kim et al. [31] proposed to distinguish the hot data and the cold data according to the program context. Wu et al. [54] proposed to optimize background segment cleaning scheduler based on Q-learning algorithm. Gwak et al. [19] optimized a foreground segment cleaning. A few works proposed to trigger background segment cleaning during system idle time. [10, 22, 47] Lee et al. [37] proposed preemptive garbage collection. Yan et al. [57] proposed copying valid pages in victim block to another block so that the copies handle IO operations to victim block.

To reduce the mapping table size, Kang et al. proposed to use larger granularity mapping [26]. Zhou et al. [63] increased the cache hit ratio of the page-level mapping information by employing a two-level LRU list. Liu et al. [42] proposed the FTL, which enables partial erase operation in 3D NAND flash storage.

8 Conclusion

In this work, we propose IPLFS, a log-structured filesystem for infinite partition. Separating the logical filesystem partition size from the physical storage size and making the logical filesystem partition size virtually infinite, we free the log-structured filesystem from recycling the invalid filesystem blocks. To maintain the mapping information for the prohibitively large logical filesystem partition, we develop Interval-Mapping which maintains the LBA to PBA mapping only for the actively used filesystem region. With IPLFS and Interval mapping combined together, we relieve the log-structured filesystem from the overhead of reclaiming the invalid blocks, the garbage collection.

Acknowledgements We are deeply indebted to our shepherd Youyou Lu for helping us to shape the final version of this paper. We are also grateful to the anonymous reviewers for their valuable comment and feedback. We like to thank Jay Hyun for his inspiring comment at the inception stage of this work. This work was supported by IITP, Korea (grant No. 2018-0-00549), and by NRF, Korea (grant No. NRF-2020R1A2C3008525).

References

[1] crosshatch: switch userdata filesystem from ext4 to f2fs. https://android.googlesource.com/device/google/crosshatch/+/a0d74ba2c0b943c6370288b13ade0cf6c4868da2.

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

[3] Idle checking code in the f2fs discard procedure. https://elixir.bootlin.com/linux/v5.11/source/fs/f2fs/segment.c#L1548.

[4] SNIA Block I/O Traces. http://iotta.snia.org/traces/block-io.

[5] Zoned Namespaces (ZNS) SSDs. https://zonedstorage.io/introduction/zns/.

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

[7] Amir Ban. Flash file system, U.S. Patent 5404485, Apr. 1995.

[8] Artem B Bityutskiy. JFFS3 design issues, 2005. http://www.linux-mtd.infradead.org.

[9] 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.

[10] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proc. of 1995 USENIX Technical Conference Proceedings*, 1995.

[11] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, November 1997.

[12] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 3(4):837–863, November 2004.

[13] Melvin E Conway. A multiprocessor system design. In *Proc. of the fall joint computer conference (AFIPS)*, 1963.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM symposium on Cloud computing*, 2010.

[15] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, 1968.

[16] H. Edelsbrunner. *Dynamic Rectangle Intersection Searching*. Forschungsberichte: Institut für Informationsverarbeitung. Inst., 1980.

[17] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974.

[18] 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.

[19] Hyunho Gwak, Yunji Kang, and Dongkun Shin. Reducing garbage collection overhead of log-structured file systems with GC journaling. In *Proc. of 2015 International Symposium on Consumer Electronics (ISCE)*, 2015.

[20] 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.

[21] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *Proc. of 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[22] Martin Jambor, Tomas Hruby, Jan Taus, Kuba Krchak, and Viliam Holub. Implementation of a Linux Log-Structured File System with a Garbage Collectors. In *Proc. of ACM Special Interest Group on Operating Systems (SIGOPS)*, 2007.

[23] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proc. of 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[24] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: A file system for virtualized flash storage. In *Proc. of 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[25] Dong Hyun Kang and Young Ik Eom. iDiscard: enhanced Discard () scheme for flash storage devices. In *Proc. of IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2018.

[26] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for NAND flash memory. In *Proc. of the 6th ACM & IEEE International conference on Embedded software (EMSOFT)*, 2006.

[27] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proc. of USENIX Technical Conference (TCON)*, 1995.

[28] Bum Soo Kim and Gui Young Lee. Method of driving remapping in flash memory and flash memory architecture suitable therefore, U.S. Patent 6381176, Apr. 2002.

[29] Jesung Kim, Jong Min Kim, S.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, August 2002.

[30] Jung-Hoon Kim, Sang-Hoon Kim, and Jin-Soo Kim. Subpage programming for extending the lifetime of NAND flash memory. In *Proc. of 2015 Design, Automation* & *Test in Europe Conference* & *Exhibition (DATE)*, 2015.

[31] 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.

[32] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM Special Interest Group on Operating Systems (SIGOPS)*, 40(3):102–107, July 2006.

[33] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Transactions on Storage (TOS)*, 16(3):1–35, July 2020.

[34] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Janus-FTL: Finding the optimal point on the spectrum between page and block mapping schemes. In *Proc. of the 10th ACM international conference on Embedded software (EMSOFT)*, 2010.

[35] 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.

[36] 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*, 32(2):247–260, 2013.

[37] Junghee Lee, Youngjae Kim, Galen M Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. A semi-preemptive garbage collector for solid state drives. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[38] 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), 2007.

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

[40] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM Special Interest Group on Operating Systems (SIGOPS)*, 42(6):36–42, 2008.

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

[42] Chun-yi Liu, Jagadish Kotra, Myoungsoo Jung, and Mahmut Kandemir. PEN: Design and evaluation of partial-erase for 3d nand-based high density ssds. In *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[43] 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 Special Interest Group on Operating Systems (SIGOPS)*, 31(5):238–251, 1997.

[44] A. Palmer. SMR in Linux Systems. In *Proc. of 2020 Linux Storage and Filesystems Conference (VAULT)*, 2020.

[45] Yubiao Pan, Yongkun Li, Huizhen Zhang, Hao Chen, and Mingwei Lin. GFTL: Group-level mapping in flash translation layer to provide efficient address translation for NAND flash-based SSDs. *IEEE Transactions on Consumer Electronics (TCE)*, 66(3):242–250, April 2020.

[46] 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, July 2008.

[47] 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.

[48] 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.

[49] 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, February 1992.

[50] 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 Winter*, 1993.

[51] Amir Ali Semnanian, Jeffrey Pham, Burkhard Englert, and Xiaolong Wu. Virtualization technology and its impact on computer hardware architecture. In *Proc. of IEEE 8th International Conference on Information Technology: New Generations (ITNG)*, 2011.

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

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

[54] Chao Wu, Cheng Ji, and Chun Jason Xue. Reinforcement learning based background segment cleaning for log-structured file system on mobile devices. In *Proc. of 2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019.

[55] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. of 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[56] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. SSD-based Workload Characteristics and Their Performance Implications. *ACM Transactions on Storage (TOS)*, 17(1):1–26, 2021.

[57] 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. In *Proc. of 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

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

[59] 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.

[60] 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.

[61] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. Lazy-RTGC: A Real-Time Lazy Garbage Collection Mechanism with Jointly Optimizing Average and Worst Performance for NAND Flash Memory Storage Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(3):1–32, 2015.

[62] 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.

[63] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian Zhou. An efficient page-level FTL to optimize address translation in flash memory. In *Proc. of the 10th European Conference on Computer Systems (EuroSys)*, pages 1–16, 2015.

A Artifact Appendix

Abstract

Our artifact consists of two parts: IPLFS and Interval Mapping FTL. IPLFS is a log-structured filesystem with Infinite logical partition. Interval Mapping FTL is flash translation layer that maintains mapping for Infinite logical partition.

Scope

This artifact can be used to validate all experiments that measure throughput, latency and block trace in the paper.

Contents

IPLFS artifact is implemented on the top of F2FS in Ubuntu 5.11.0. The IPLFS artifact does not conduct garbage collection, and does not have metadata for garbage collection, such as a block allocation bitmap (a segment information table) and reverse mapping information (segment summary area). To replace the block allocation bitmap, discard bitmap is implemented in the IPLFS artifact. The IPLFS artifact also conducts discard logging to prevent storage leak. As specified in the paper, the IPLFS artifact partitions infinite logical space into seven areas.

Interval Mapping artifact is implemented on the top of OpenSSD. The design of the Interval Mapping artifact is three level tree, as written in the paper. The root node, zone node, and map node are all implemented in the artifact. We implement Expansion of root node and Compaction of map node in the artifact. In the artifact, there are total seven Interval Mapping trees to support for multi-area partition layout of IPLFS.

Hosting

The IPLFS artifact is uploaded in Github repository, https://github.com/ESOS-Lab/IPLFS. A branch named 'IPLFS-stable' contains IPLFS source code, and IPLFS format utility (f2fs-tools). A branch named 'original_kernel' is vanilla kernel which is compared with IPLFS for experiment in the paper.

The Interval Mapping artifact is uploaded in Github repository, https://github.com/ESOS-Lab/Interval_Mapping. A branch named 'main' is the artifact mainly used for the experiments in the paper. If you try measuring 'get' and 'set' latencies of Interval Mapping FTL, please use branches named 'exp-getlatency' and 'exp-setlatency', which print out the latencies of 'get' operation and 'set' operation, respectively. A branch named 'expansion+compaction' is Interval Mapping that conducts root node expansion and map node compaction in the foreground.

Requirements

Interval Mapping FTL is built on Cosmos+ OpenSSD, the PCIe-based SSD platform on which open source SSD firmware can be developed. To operate the Interval Mapping artifact, OpenSSD is required.