Application-Managed Flash

Sungjin Lee, Ming Liu, Sangwoo Jun, and Shuotao Xu, *MIT CSAIL;*
Jihong Kim, *Seoul National University;* Arvind, *MIT CSAIL*

https://www.usenix.org/conference/fast16/technical-sessions/presentation/lee

This paper is included in the Proceedings of the
14th USENIX Conference on
File and Storage Technologies (FAST '16).

February 22–25, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-28-7

Application-Managed Flash

Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim* and Arvind

Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology

**Department of Computer Science and Engineering, Seoul National University*

Abstract

In flash storage, an FTL is a complex piece of code that resides completely inside the storage device and is provided by the manufacturer. Its principal virtue is providing interoperability with conventional HDDs. However, this virtue is also its biggest impediment in reaching the full performance of the underlying flash storage. We propose to refactor the flash storage architecture so that it relies on a new block I/O interface which does not permit overwriting of data without intervening erasures.We demonstrate how high-level applications, in particular file systems, can deal with this restriction efficiently by employing *append-only* segments. This refactoring dramatically reduces flash management overhead and improves performance of applications, such as file systems and databases, by permitting them to directly manage flash storage. Our experiments on a machine with the new block I/O interface show that DRAM in the flash controller is reduced by 128X and the performance of the file system improves by 80% over conventional SSDs.

1 Introduction

NAND flash SSDs have become the preferred storage device in both consumer electronics and datacenters. Flash has superior random access characteristics to speed up many applications and consumes less power than HDDs. Thanks to Moore's law and advanced manufacturing technologies like 3D NAND [27], the use of flash-based devices is likely to keep rising over the next decade.

SSDs employ a flash translation layer (FTL) to provide an I/O abstraction of a generic block device so that HDDs can be replaced by SSDs without the software being aware of flash characteristics. An FTL is a complex piece of software because it must manage the overwriting restrictions, wear- leveling and bad-block management.

Implementing these management tasks requires significant hardware resources in the flash controller. In particular, tasks like address remapping and garbage collection require large amounts of DRAM and powerful CPUs (e.g., a 1 GHz quad-core CPU with 1 GB DRAM [19, 48, 45]). The FTL makes important decisions affecting storage performance and lifetime, without any awareness of the high-level application, and consequently the resulting performance is often suboptimal [15, 28, 9, 17]. Moreover, the FTL works as a black box – its inner-workings are hidden behind a block I/O layer, which makes the behavior of flash storage unpredictable for higher-level applications, for example, unexpected invocation of garbage collection [14] and swap-in/out of mapping entries [44, 23].

Another serious drawback of the FTL approach is the duplication of functionality between the FTL and the host. Many host applications already manage underlying storage to avoid in-place updates for several reasons such as better performance, efficient versioning, data consistency and so on. Log-structured or copy-on-write file systems always append new data to the device, mostly avoiding in-place updates [47, 20, 46, 31, 33, 50]. Similar log-structured systems are used in the database community [49, 1, 5]. The LSM-Tree is also a well known data structure based on a log-structured approach [42, 12, 2, 6]. Since the FTL is not aware of this avoidance of overwriting, it employs its own log-structured technique to manage flash. Running log-structured applications on top of a log-structured FTL wastes hardware resource and incurs extra I/Os. This double logging problem is also reported by empirical studies conducted by industry [53].

In this paper, we present a different approach to managing flash storage, which is called an **A**pplication-**M**anaged **F**lash (AMF). As its name implies, AMF moves the intelligence of flash management from the device to applications, which can be file systems, databases and user applications, leaving only essential management parts on the device side. For various applications to easily use AMF, we define a new block I/O interface which does not support overwrites of data unless they were explicitly deallocated (i.e., an attempt to overwrite data without a proper deallocation generates an error). This dramatically simplifies the management burden inside the device because fine-grained remapping and garbage collection do not have to be done in the device. The application using the flash device is completely responsible for avoiding in-place updates and issuing trim commands to indicate that the data has been deallocated and the device can erase and reuse the associated flash blocks. This direct flash management by applications has several advantages. For example, it can (i) efficiently schedule both regular and cleaning I/Os (e.g., copying and compacting) based on the states of the processes; (ii) accurately separate hot and cold data according to its properties (e.g., metadata versus data); and (iii) directly map objects (e.g., files) to physical locations without maintaining a huge mapping table.

In AMF, the device responsibility is reduced to providing error-free storage accesses and efficient parallelism support to exploit multiple storage chips on buses or channels. The device also keeps track of bad blocks and performs wear-leveling. It is preferable to do these operations in the device because they depend upon the specific circuit designs and manufacturing process of the device. Understandably, device manufacturers are reluctant to disclose such details. In our system, management tasks in the device are performed at the block granularity as opposed to the page granularity, therefore mapping tables required are considerably smaller.

One may think that avoiding in-place updates places too much burden on applications or users. However, this is not the case because many applications that use HDDs already employ append-only strategies as we pointed out earlier, and these are our target applications. For such applications, the only additional burden in using our interface is to avoid rare in-place updates. Moreover, forcing host applications to write data sequentially is not an extreme constraint either. For example, shingled magnetic recording (SMR) HDDs have already adopted a similar approach for the management of overlapped sectors [11].

To demonstrate the advantages of AMF, we used an open FPGA-based flash platform, BlueDBM [25, 36], as our testbed which provides error-free accesses to raw flash chips. We implemented a new lightweight FTL called an Application-managed FTL (AFTL) to support our block I/O interface. For our case study with applications, we have selected a file system because it is the most common application to access flash storage. We have implemented a new Application-managed Log-structured File-System (ALFS). The architecture of ALFS is exactly the same as the conventional LFS, except that it appends the metadata as opposed to updating it in-place. It should be noted that applying AMF

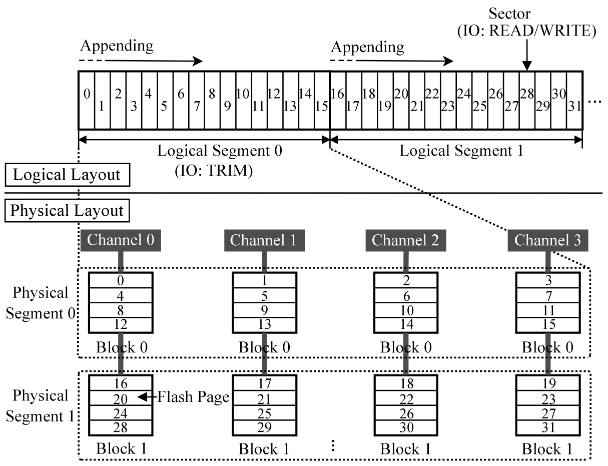


Figure 1: An AMF block I/O abstraction: It shows two logical segments (logical segments 0 and 1) and two corresponding physical ones on the device side (physical segments 0 and 1). A logical segment is composed of 16 sectors which are *statically* mapped to flash pages. A physical segment is organized with four flash blocks belonging to four channels and one way.

is not limited to a file system only. Many real world applications can benefit from AMF. For example, key-value stores based on LSM-Trees (e.g., LevelDB [12] and RocksDB [2]), logical volume managers combined with CoW file systems (e.g., WAFL [20] and Btrfs [46]) and log-structured databases (e.g., RethinkDB [1]) are candidate applications for AMF.

Our experiments show that AMF improves I/O performance and storage lifetime by up to 80% and 38%, respectively, over file systems implemented on conventional FTLs. The DRAM requirement for the FTL was reduced by a factor of 128 because of the new interface while the additional host-side resources (DRAM and CPU cycles) required by AMF were minor.

This paper is organized as follows: Section 2 explains our new block I/O interface. Sections 3 and 4 describe AMF. Section 5 evaluates AMF with various benchmarks. Section 6 reviews related work. Section 7 concludes with a summary and future directions.

2 AMF Block I/O Interface

Figure 1 depicts the block I/O abstraction of AMF, showing both logical and physical layouts. The block I/O interface of AMF is based on conventional block I/O – it exposes a linear array of fixed size blocks or sectors (e.g., 4 KB), which are accessed by three I/O primitives, `READ`, `WRITE` and `TRIM`. To distinguish a logical block from a flash block, we call it a *sector* in the remainder of this paper. Continuous sectors are grouped into a larger extent (e.g., several MB), called a *segment*.

A segment is allocated when the first write is performed and its size grows implicitly as more writes are performed. A segment is deallocated by issuing a `TRIM` command. Therefore, `TRIM` is always conducted in the unit of a segment. A sector of a segment can be read once it has been written. However, a sector can be written only once; an overwrite generates an error. To avoid this overwrite problem, the host software should write the sectors of a segment in an append-only manner, starting from the lowest sector address. This sector can be reused after the segment it belongs to has been deallocated by `TRIM`.

A segment exposed to upper layers is called a *logical segment*, while its corresponding physical form is called a *physical segment*. Segmentation is a well known concept in many systems. In particular, with log-structured systems, a logical segment is used as the unit of free space allocation, where new data is *sequentially* appended and free space is reclaimed later. A physical segment on the storage device side is optimized for such a sequential access by software. In Figure 1, a physical segment is composed of a group of blocks spread among different channels and ways, and sectors within a logical segment are statically mapped to flash pages within a physical one. This mapping ensures the maximum bandwidth of the device when data is read or written sequentially. It also provides predictable performance unaffected by the firmware's behavior.

Our block I/O interface expects the device controller to take the responsibility for managing bad-blocks and wear-leveling so that all the segments seen by upper layers are error-free. There are two main reasons for our decision. First, it makes the development of systems and applications easier since bad-block management and wear-leveling are relatively simple to implement at the device level and do not require significant resources. Second, the lifetime of NAND devices can be managed more effectively at lower levels where device-level information is available. Besides P/E cycles, the lifetime of NAND devices are affected by factors such as recovery effects [13] and bit error rates [43]; SSD vendors take these factors into consideration in wear-leveling and bad-block management. This information is proprietary and confidential – for example, some vendors do not reveal even P/E cycles on datasheets. Hiding these vendor and device-specific issues inside the flash controller also makes the host software vendor independent.

Compatibility Issue: Our block I/O interface maintains good compatibility with existing block I/O subsystems – the same set of I/O primitives with fixed size sectors (i.e., `READ`, `WRITE` and `TRIM`). The only new restrictions introduced by the AMF block I/O interface are (i) non-rewritable sectors before being trimmed, (ii) a linear array of sectors grouped to form a segment and (iii) the unit of a `TRIM` operation. Note that a segment size is easily shared by both applications and devices through interfaces like S.M.A.R.T and procfs. In our Linux implementation, for example, the existing block I/O layer is not changed at all. This allows us to convert existing software to run on AMF in an easy manner.

The architecture of AFTL is similar to block or segment-level FTLs and requires minimal functions for flash management, thus SSD vendors can easily build AMF devices by removing useless functions from their devices. For better compatibility with conventional systems, SSD vendors can enhance their SSD products to support two different modes: device-managed flash and application-managed flash. This allows us to choose the proper mode according to requirements. The addition of AFTL to existing SSDs may not require much efforts and hardware resources because of its simplicity.

3 AMF Log-structured File System

In this section, we explain our experience with the design and implementation of ALFS. We implement ALFS in the Linux 3.13 kernel based on an F2FS file system [33]. Optimizing or enhancing the fundamental LFS design is not a goal of this study. For that reason, most of the data structures and modules of F2FS are left unchanged. Instead, we focus on two design aspects: (i) where in-place updates occur in F2FS and (ii) how to modify F2FS for the AMF block I/O interface. The detailed implementation of F2FS is different from other LFSs [38, 31], but its fundamental design concept is the same as its ancestor, Sprite LFS [47]. For the sake of simplicity and generality, we explain the high-level design of ALFS using general terms found in Sprite LFS.

3.1 File Layout and Operations

Figure 2 shows the logical segments of ALFS, along with the corresponding physical segments in AFTL. All user files, directories and inodes, including any modifications/updates, are appended to free space in logical segments, called *data segments*. ALFS maintains an inode map to keep track of inodes scattered across the storage space. The inode map is stored in reserved logical segments, called *inode-map segments*. ALFS also maintains check-points that point to the inode map and keep the consistent state of the file system. A check-point is written periodically or when a flush command (e.g., `fsync`) is issued. Logical segments reserved for check-points are called *check-point segments*.

ALFS always performs out-of-place updates even for the check-point and the inode-map because of the requirements of the AMF block I/O interface. Hence, their locations are not fixed. This makes it difficult to find the

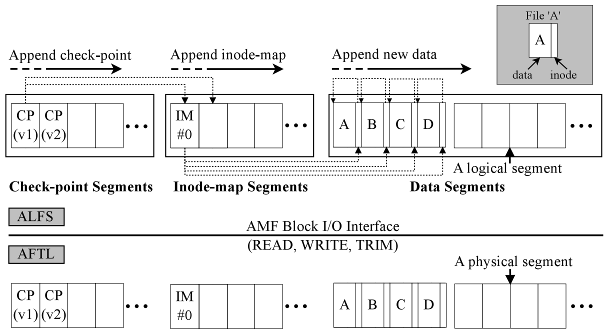


Figure 2: The upper figure illustrates the logical layout of ALFS. There is an initial check-point CP(v1). Four files are appended to data segments along with their inodes in the following order: A, B, C and D. Then, an inode map IM#0 is written which points to the locations of the inodes of the files. Finally, the check-point CP(v2) is written to check-point segments. The bottom figure shows the physical segments corresponding to the logical segments. The data layout of a logical segment perfectly aligns with its physical segment.

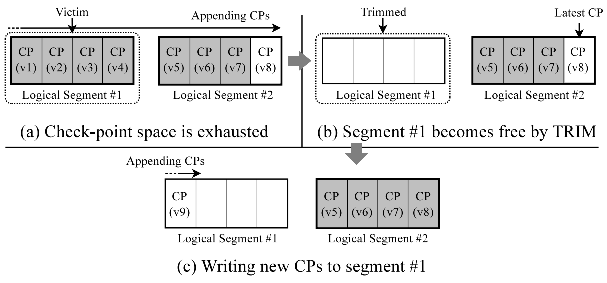


Figure 3: Check-point segment handling

latest check-point and the locations of inodes in inode-map segments after mounting or power failure. Next, we explain how ALFS manages check-point segments for quick mount and recovery, and show how it handles inode-map segments for fast searches of inodes.

3.2 Check-Point Segment

The management of check-point segments is straightforward. ALFS reserves *two fixed* logical segments #1 and #2 for check-points. (Note: a logical segment #0 is reserved for a superblock). Figure 3 shows an example of check-point management. ALFS appends new check-points with incremental version numbers using the available free space. If free space in segments is exhausted, the segment containing only old check-point versions is selected as a victim for erasure (see Figure 3(a)). The latest check-point is still kept in the other segment. ALFS sends TRIM commands to invalidate and free the victim (see Figure 3(b)). Then, it switches to the freed segment and keeps writing new check-points (see Figure 3(c)). Even though ALFS uses the same logical segments repeatedly, it will not unevenly wear out flash because AFTL performs wear-leveling.

When ALFS is remounted, it reads all the check-point segments from AFTL. It finds the latest check-point by comparing version numbers. This brute force search is efficient because ALFS maintains only two segments for check-pointing, regardless of storage capacity. Since segments are organized to maximize I/O throughput, this search utilizes full bandwidth and mount time is short.

3.3 Inode-Map Segment

The management of inode-map segments is more complicated. The inode map size is decided by the maximum number of inodes (i.e., files) and is proportional to storage capacity. If the storage capacity is 1 TB and the minimum file size is 4 KB, 2^{28} files can be created. If each entry of the inode map is 8 B (4 B for an inode number and 4 B for its location in a data segment), then the inode map size is 2 GB (= 8 B$\times 2^{28}$). Because of its large size, ALFS divides the inode map into 4 KB blocks, called *inode-map blocks*. There are 524,288 4-KB inode-map blocks for the inode map of 2 GB, each of which contains the mapping of 512 inodes (see Table 1). For example, IM#0 in Figure 2 is an inode-map block.

ALFS always appends inode-map blocks to free space, so the latest inode-map blocks are scattered across inode-map segments. To identify the latest valid inode-map blocks and to quickly find the locations of inodes, we need to develop another scheme.

Inode-Map Block Management: Figure 4 illustrates how ALFS manages inode-map blocks. To quickly find the locations of inodes, ALFS maintains a table for inode-map blocks (TIMB) in main memory. TIMB consists of 4 B entries that point to inode-map blocks in inode-map segments. Given an inode number, ALFS finds its inode-map block by looking up TIMB. It then obtains the location of the inode from that inode-map block. The TIMB size is 2 MB for 524,288 inode-map blocks (= 4 B$\times 524,288$), so it is small enough to be kept in the host DRAM. The in-memory TIMB should be stored persistently; otherwise, ALFS has to scan all inode-map segments to construct the TIMB dur-

Data structure	Unit size	Count	Storage
Inode-map block	4 KB	524K	Flash (inode-map segs)
In-memory TIMB	2 MB	1	DRAM
TIMB block	4 KB	512	Flash (inode-map segs)
TIMB-blocks list	2 KB	1	Flash (a check-point)

Table 1: An example of data structures sizes and locations with a 1 TB SSD. Their actual sizes vary depending on ALFS implementation (e.g., an inode-map size) and storage capacity.

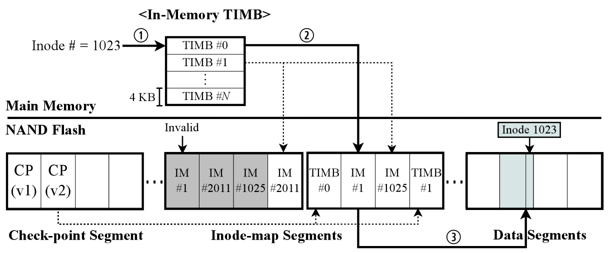


Figure 4: To find an inode, ALFS first looks up in-memory TIMB to find the location of inode-map blocks that points to the inode in flash. Each 4 KB TIMB block indicates 1,024 inode-map blocks in inode-map segments (e.g., TIMB#0 points to IM#0∼IM#1023 in flash). Since each inode-map block points to 512 inodes, TIMB#0 block indicates inodes ranging from 0-524,288 in flash. If ALFS searches for an inode whose number is 1023, it looks up TIMB#0 in the in-memory TIMB (①) and finds the location of IM#1 that points to 512∼1023 inodes (②). Finally, the inode 1023 can be read from a data segment (③). Note that the latest check-point points to all of the physical locations of TIMB blocks in flash.

ing mount. ALFS divides the TIMB into 4 KB blocks (TIMB blocks) and keeps track of dirty TIMB blocks that hold newly updated entries. ALFS appends dirty TIMB blocks to free space in inode-map segments just before a check-point is written.

TIMB blocks themselves are also stored in non-fixed locations. To build the in-memory TIMB and to safely keep it against power failures, a list of all the physical locations of TIMB blocks (TIMB-blocks list) is written to check-point segments together with the latest check-point. Since the size of the in-memory TIMB is 2 MB, the number of TIMB blocks is 512 (= 2 MB/4 KB). If 4 B is large enough to point to locations of TIMB blocks, the TIMB-blocks list is 2 KB (= 4 B$\times$512). The actual size of check-point data is hundred bytes (e.g., 193 bytes in F2FS), so a check-point with a TIMB-block list can be written together to a 4 KB sector without extra writes.

Remount Process: The in-memory TIMB should be reloaded properly whenever ALFS is mounted again. ALFS first reads the latest check-point as we described in the previous subsection. Using a TIMB-blocks list in the check-point, ALFS reads all of the TIMB blocks from inode-map segments and builds the TIMB in the host DRAM. The time taken to build the TIMB is negligible because of its small size (e.g., 2 MB for 1 TB storage).

Up-to-date TIMB blocks and inode-map blocks are written to inode-map segments before a new check-point is written to NAND flash. If the check-point is successfully written, ALFS returns to the consistent state after power failures by reading the latest check-point. All the TIMB blocks and inode-map blocks belonging to an incomplete check-point are regarded as obsolete data. The recovery process of ALFS is the same as the remount process since it is based on LFS [47].

Garbage Collection: When free space in inode-map segments is almost used up, ALFS should perform garbage collection. In the current implementation, the least-recently-written inode-map segment is selected as a victim. All valid inode-map blocks in the victim are copied to a free inode-map segment that has already been reserved for garbage collection. Since some of inode-map blocks are moved to the new segment, the in-memory TIMB should also be updated to point to their new locations accordingly. Newly updated TIMB blocks are appended to the new segment, and the check-point listing TIMB blocks is written to the check-point segment. Finally, the victim segment is invalidated by a TRIM command and becomes a free inode-map segment.

To reduce live data copies, ALFS increases the number of inode-map segments such that their total size is larger than the actual inode-map size. This wastes file-system space but greatly improves garbage collection efficiency because it facilitates inode-map blocks to have more invalid data prior to being selected as a victim. ALFS further improves garbage collection efficiency by separating inode-map blocks (i.e., hot data) in inode-map segments from data segments (i.e., cold data). Currently, ALFS allocates inode-maps segments which are four times larger than its original size (e.g., 8 GB if the inode map size is 2 GB). The space wasted by extra segments is small (e.g., 0.68% = 7 GB / 1 TB).

All of the I/O operations required to manage inode-map blocks are extra overheads that are not present in the conventional LFS. Those extra I/Os account for a small portion, which is less than 0.2% of the total I/Os.

3.4 Data Segment

ALFS manages data segments exactly the same way as in the conventional LFS – it buffers file data, directories and inodes in DRAM and writes them all at once when their total size reaches a data segment size. This buffering is advantageous for ALFS to make use of the full bandwidth of AFTL. ALFS performs segment cleaning when free data segments are nearly exhausted. The procedure of segment cleaning is illustrated in Figure 5.

Besides issuing TRIM commands after segment cleaning, we have not changed anything in F2FS for management of data segments because F2FS already manages data segments in an append-only manner. This is a good example of how easily AMF can be used by other log-structured systems. It also allows us to automatically borrow advanced cleaning features from F2FS [33] without any significant effort.

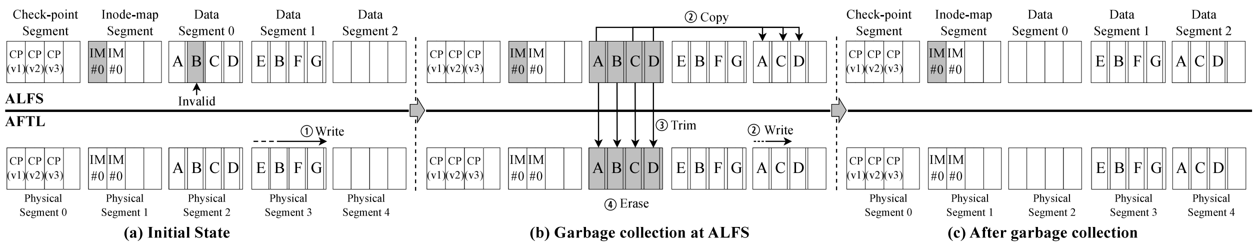


Figure 5: Writes and garbage collection of ALFS with AFTL: (a) Four new files E, B, F and G are written to data segment 1. The file B is a new version. ALFS appends IM#0 to the inode-map segment because it points to the locations of the files. A new check-point CP(v3) is appended. (b) Free space in ALFS is nearly exhausted, so ALFS triggers garbage collection. ALFS copies the files A, C and D to data segment 2. Since data segment 0 has only invalid data, ALFS sends TRIM commands to AFTL, making it free. Finally, AFTL erases physical segment 2. There are 3 page copies and 2 block erasures for garbage collection.

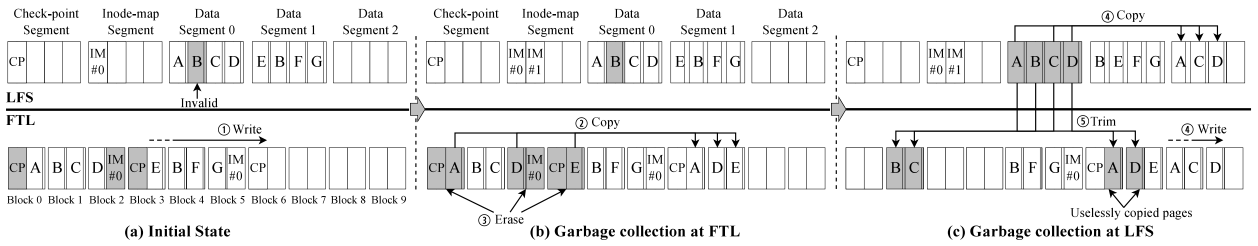


Figure 6: Writes and garbage collection of LFS with FTL: FTL sequentially writes all the sectors to NAND flash using a mapping table. (a) The files E, B, F, and G are appended to free pages. IM#0 and CP are overwritten in the same locations in LFS. FTL maps them to free pages, invaliding old versions. (b) FTL decides to perform garbage collection. It copies flash pages for A, D and E to free pages and gets 3 free blocks (Blocks 0, 2, 3). (c) LFS is unaware of FTL, so it also triggers garbage collection to create free space. It moves the files A, C and D to free space and sends TRIM commands. For garbage collection, there are 6 page copies and 3 block erasures. The files A and D are moved uselessly by FTL because they are discarded by LFS later.

3.5 Comparison with Conventional LFS

In this section, we compare the behavior of ALFS with a conventional LFS that runs on top of a traditional FTL. For the same set of operations, Figure 5 illustrates the behaviors of ALFS, while Figure 6 shows those of the conventional LFS with the FTL. For the sake of simplicity, we assume that ALFS and LFS have the same file-system layout. The sizes of a sector and a flash page are assumed to be the same. LFS keeps check-points and an inode-map in a fixed location and updates them by overwriting new data.[1] On the storage device side, LFS runs the page-level FTL that maps logical sectors to any physical pages in NAND flash. In AFTL, a physical segment is composed of two flash blocks. AFTL just erases flash blocks containing only obsolete pages.

Figures 5 and 6 demonstrate how efficiently ALFS manages NAND flash compared to LFS with the FTL. LFS incurs a larger number of page copies for garbage collection than ALFS. This inefficiency is caused by (i) in-place updates to check-point and inode-map regions by LFS. Whenever overwrites occur, the FTL has to map up-to-date data to new free space, invalidating its old version that must be reclaimed by the FTL later (see Blocks 0, 2, 3 in Figure 6(a)). Other versions of LFS that overwrite only check-points (e.g., Sprite LFS) also have the same problem. (ii) The second is unaligned logging by both LFS and the FTL which results in data from different segments being mixed up in the same flash blocks. To lessen FTL-level garbage collection costs, LFS discards the entire logical segment (i.e., data segment 0) after cleaning, but it unintentionally creates dirty blocks that potentially cause page copies in the future (see Blocks 6 and 7 in Figure 6(c)).

In ALFS, in-place updates are never issued to the device and the data layout of a logical segment perfectly aligns with a corresponding physical segment. Thus, the problems with LFS do not occur in ALFS.

4 AMF Flash Translation Layer (AFTL)

In this section, we explain the design and implementation of AFTL. We implement AFTL in a device driver because our SSD prototype does not have a processor, but it can be implemented in the device if a processor is avail-

[1]This is somewhat different depending on the design of LFS. Sprite LFS overwrites data in a check-point region only [47], while NILFS writes segment summary blocks in an in-place-update fashion [31]. F2FS overwrites both a check-point and an inode map [33]. Since ALFS is based on F2FS, we use the design of F2FS as an example.

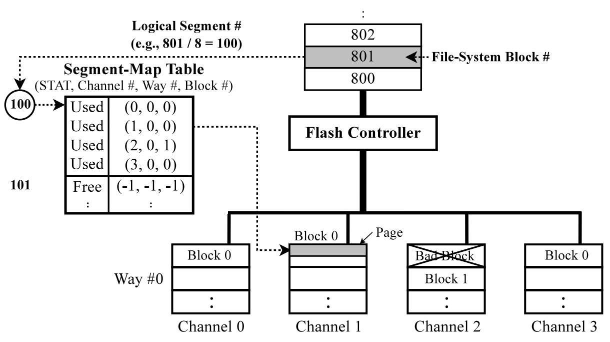


Figure 7: An example of how AFTL handles writes: There are four channels and one way in AFTL, and each block is composed of two pages. A physical segment has 8 pages. When a write request comes, AFTL gets a logical segment number (i.e., 100 = 801/8) using the logical sector number. It then looks up the segment-map table to find a flash block mapped to the logical segment. In this example, the logical block '801' is mapped to 'Block 0' in 'Channel #1'. Finally, AFTL writes the data to a corresponding page offset in the mapped block.

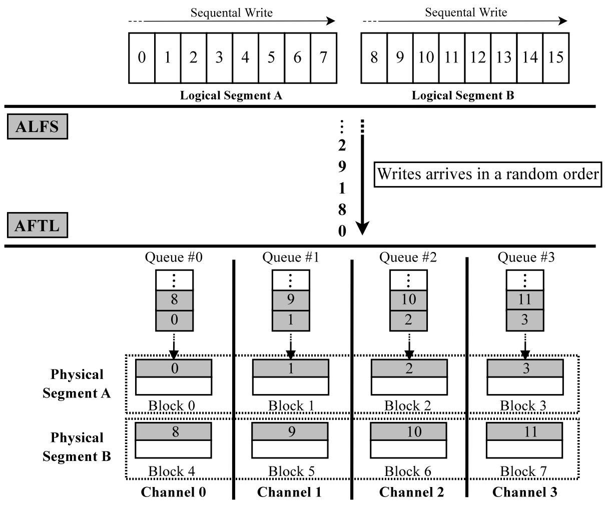


Figure 8: An example of how AFTL handles write requests when ALFS appends data to two segments A and B simultaneously: Numbers inside rectangles indicate a file-system sector address. ALFS sequentially writes data to segments A and B, but write requests arrive at AFTL in a random order (i.e., 0, 8, 1, ...). They are sorted in multiple I/O queues according to their destined channels and are written to physical segments in a way of fully utilizing four channels. If a single queue with FIFO scheduling is used, the sector '1' is delayed until '0' and '8' are sent to flash blocks '0' and '4' through the channel 0.

able. The architecture of AFTL is similar to a simplified version of the block-level FTL [4], except that AFTL does not need to run address remapping to avoid in-place updates, nor does it need to perform garbage collection. For the sake of clarity, we focus on describing the minimum requirements for AFTL implementation rather than explaining how to improve existing FTLs to support the new block I/O interface.

Wear-Leveling and Bad-Block Management: As discussed in Section 2, sectors in a logical segment are statically mapped to flash pages. For wear-leveling and bad-block management, AFTL only needs a small segment-map table that maps a logical segment to a physical segment. Each table entry contains the physical locations of flash blocks mapped to a logical segment along with a status flag (`STAT`). Each entry in the table points to blocks of a logical segment that is striped across channels and ways. `STAT` indicates Free, Used or Invalid.

Figure 7 shows how AFTL handles write requests. If any physical blocks are not mapped yet (i.e., `STAT` is Free or Invalid), AFTL builds the physical segment by allocating new flash blocks. A bad block is not selected. AFTL picks up the least worn-out free blocks in the corresponding channel/way. To preserve flash lifetime and reliability, AFTL can perform static wear-leveling that exchanges the most worn-out segments with the least worn-out ones [7]. If there are previously allocated flash blocks (i.e., `STAT` is Invalid), they are erased. If a logical segment is already mapped (i.e., `STAT` is Used), AFTL writes the data to the fixed location in the physical segment. ALFS informs AFTL via TRIM commands that the physical segments have only obsolete data. Then, AFTL can figure out which blocks are out-of-date. Upon receiving the TRIM command, AFTL invalidates that segment by changing its `STAT` to Invalid. Invalid segments are erased on demand or in background later.

I/O Queueing: AFTL employs per-channel/way I/O queues combined with a FIFO I/O scheduler. This multiple I/O queueing is effective in handling multiple write streams. ALFS allocates several segments and writes multiple data streams to different segments at the same time. For example, a check-point is often written to check-point segments while user files are being written to data segments. Even if individual write streams are sent to segments sequentially, multiple write streams arriving at AFTL could be mixed together and be random I/Os, which degrades I/O parallelism. Figure 8 shows how AFTL handles random writes using multiple queues.

This management of multiple write streams in AMF is more efficient than conventional approaches like multi-streamed SSDs [28]. In multi-streamed SSDs, the number of segments that can be opened for an individual write stream is specified at the configuration time. There is no such a limitation in AMF; ALFS opens as many logical segments as needed to write multiple streams. All the data are automatically separated to different physical segments according to the segment-level mapping. This enables applications to more efficiently separate data according to their properties.

Write skews do not occur for any channel or way in

Capacity	Block-level FTL	Hybrid FTL	Page-level FTL	AMF	
				AFTL	ALFS
512 GB	4 MB	96 MB	512 MB	4 MB	5.3 MB
1 TB	8 MB	186 MB	1 GB	8 MB	10.8 MB

Table 2: A summary of memory requirements

AFTL. This is because ALFS allocates and writes data in the unit of a segment, distributing all the write requests to channels and ways uniformly. Moreover, since FTL garbage collection is never invoked in AFTL, I/O scheduling between normal I/Os and GC I/Os is not required. Consequently, simple multiple I/O queueing is efficient enough to offer good performance, and complex firmware algorithms like load-balancing [8] and out-of-ordering [39, 16] are not required in AFTL.

5 Experimental Results

We begin our analysis by looking at memory requirements for AFTL and ALFS, comparing against commonly used FTL schemes. We then evaluate the performance of AMF using micro-benchmarks to understand its behavior under various I/O access patterns. We benchmark AMF using realistic applications that have more complex I/O access patterns. Finally, we measure lifetime, I/O latency and CPU utilization of the system.

5.1 Memory Requirements

We compare the mapping table sizes of AFTL with three FTL schemes: block-level, hybrid and page-level FTLs. Block-level FTL uses a flash block (512 KB) as the unit of mapping. Because of its low performance, it is rarely used in production SSDs. Page-level FTL performs mapping on flash pages (4-16KB). Hybrid FTL is a combination of block-level and page-level FTLs – while the block-level mapping is used to manage the storage space offered to end-users, the page-level mapping is used for an over-provisioning area. For the hybrid FTL, 15% of the total capacity is used as the over-provisioning area. AFTL maintains the segment-map table pointing to flash blocks for wear-leveling and bad-block management.

Table 2 lists the mapping table sizes of 512 GB and 1 TB SSDs. For the 512 GB SSD, the mapping table sizes are 4 MB, 96 MB, 512 MB and 4 MB for block-level, hybrid, page-level FTLs and AFTL, respectively. The mapping table sizes increase in proportional to the storage capacity – when the capacity is 1 TB, block-level, hybrid, page-level FTLs and AFTL require 8 MB, 62 MB, 1 GB and 8 MB memory, respectively. AFTL maintains a smaller mapping table than the page-level FTL, enabling us to keep all mapping entries in DRAM even for the 1 TB SSD. Table 2 shows the host DRAM requirement for

Category	Workload	Description
File System	`FIO`	A synthetic I/O workload generator
	`Postmark`	A small and metadata intensive workload
Database	`Non-Trans`	A non-transactional DB workload
	`OLTP`	An OLTP workload
	`TPC-C`	A TPC-C workload
Hadoop	`DFSIO`	A HDFS I/O throughput test application
	`TeraSort`	A data sorting application
	`WordCount`	A word count application

Table 3: A summary of benchmarks

ALFS, including tables for inode-map blocks (TIMB) as well as other data structures. As listed in the table, ALFS requires a tiny amount of host DRAM.

5.2 Benchmark Setup

To understand the effectiveness of AMF, we compared it with two file systems, EXT4 and F2FS [33], running on top of two different FTL schemes, page-level FTL (PFTL) and DFTL [14]. They are denoted by `EXT4+PFTL`, `EXT4+DFTL`, `F2FS+PFTL`, and `F2FS+DFTL`, respectively.

PFTL was based on pure page-level mapping that maintained all the mapping entries in DRAM. In practice, the mapping table was too large to be kept in DRAM. To address this, DFTL stored all the mapping entries in flash, keeping only popular ones in DRAM. While DFTL reduced the DRAM requirement, it incurred extra I/Os to read/write mapping entries from/to NAND flash. We set the DRAM size so that the mapping table size of DFTL was 20% of PFTL. Since DFTL is based on the LRU-based replacement policy, 20% hot entries of the mapping table were kept in DRAM. For both PFTL and DFTL, greedy garbage collection was used, and an over-provisioning area was set to 15% of the storage capacity. The over-provisioning area was not necessary for AFTL because it did not perform garbage collection. For all the FTLs, the same dynamic wear-leveling algorithm was used, which allocated youngest blocks for writing incoming data.

For EXT4, a default journaling mode was used and the discard option was enabled to use TRIM commands. For F2FS, the segment size was always set to 2 MB which was the default size. For ALFS, the segment size was set to 16 MB which was equal to the physical segment size. ALFS allocated 4x larger inode-map segments than its original size. For both F2FS and ALFS, 5% of file-system space was used as an over-provisioning area which was the default value.

5.3 Performance Analysis

We evaluated AMF using 8 different workloads (see Table 3), spanning 3 categories: file-system, DBMS and

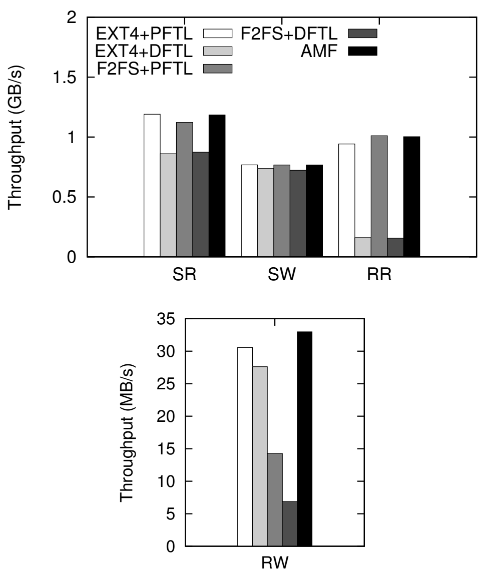


Figure 9: Experimental results with `FIO`

	`EXT4+` `PFTL`	`EXT4+` `DFTL`	`F2FS+` `PFTL`		`F2FS+` `DFTL`		`AMF`
	FTL	FTL	FS	FTL	FS	FTL	FS
FIO(SW)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FIO(RW)	1.41	1.45	1.35	1.82	1.34	2.18	1.38
Postmark(L)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Postmark(H)	1.12	1.35	1.17	2.23	1.18	2.89	1.16
Non-Trans	1.97	2.00	1.58	2.90	1.59	2.97	1.59
OLTP	1.45	1.46	1.23	1.78	1.23	1.79	1.24
TPC-C	2.33	2.21	1.81	2.80	1.82	5.45	1.87
DFSIO	1.0	1.0	1.0	1.0	1.0	1.0	1.0
TeraSort	1.0	1.0	1.0	1.0	1.0	1.0	1.0
WordCount	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 4: Write amplification factors (WAF). For F2FS, we display WAF values for both the file system (FS) and the FTL. In `FIO`, the WAF values for the read-only workloads FIO (RR) and FIO (SR) are not included.

Hadoop. To understand the behaviors of AMF under various file-system operations, we conducted a series of experiments using two well known file system benchmarks, `FIO` [3] and `Postmark` [30]. We also evaluated AMF using response time sensitive database workloads: `Non-Trans`, `OLTP` and `TPC-C`. Finally, we assessed AMF with Hadoop applications from HiBench [21], `HFSIO`, `TeraSort` and `WordCount`, which required high I/O throughput for batch processing.

For performance measurements, we focused on analyzing the effect of extra I/Os by the FTL on performance specifically caused by garbage collection and swap-in/out of mapping entries. There were no extra I/Os from wear-leveling since dynamic wear-leveling was used. EXT4, F2FS and AMF all performed differently from the perspective of garbage collection. Since EXT4 is a journaling file system, only the FTL in the storage device performed garbage collection. In F2FS, both F2FS and the FTL did garbage collection. In AMF, only ALFS performed garbage collection. There were no extra swapping I/Os in PFTL and AFTL for fetching/evicting mapping entries from/to flash because their tables were always kept in DRAM. Only DFTL incurred extra I/Os to manage in-flash mapping entries. Note that our implementation of PFTL and DFTL might be different from that of commercial FTLs. Two technical issues related to PFTL and DFTL (i.e., cleaning and swapping costs), however, are well known and common problems. For this reason, our results are reasonable enough to understand the benefits of AMF on resolving such problems.

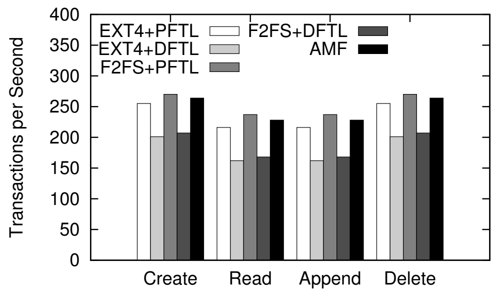
Our experiments were conducted under the same host and flash device setups. The host system was Intel's Xeon server with 24 1.6 GHz cores and 24 GB DRAM. The SSD prototype had 8 channels and 4 ways with 512 GB of NAND flash, composed of 128 4 KB pages per block. The raw performance of our SSD was 240K IOPS (930 MB/s) and 67K IOPS (260 MB/s) for reads and writes, respectively. To quickly emulate aged SSDs where garbage collection occurs, we set the storage capacity to 16 GB. This was a feasible setup because SSD performance was mostly decided by I/O characteristics (e.g., data locality and I/O patterns), not by storage capacity. The host DRAM was set to 1.5 GB to ensure that requests were not entirely served from the page cache.

5.3.1 File System Benchmarks

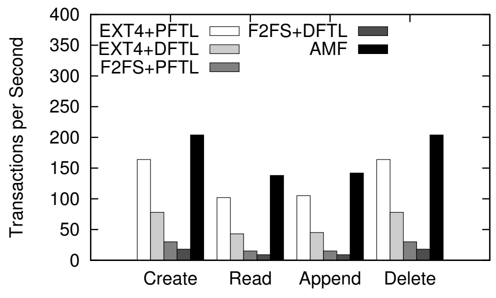
FIO: We evaluate sequential and random read/write performance using the `FIO` benchmark. `FIO` first writes a 10 GB file and performs sequential-reads (SR), random-reads (RR), sequential-writes (SW) and random-writes (RW) on it separately. We use a libaio I/O engine, 128 io-depth, and a 4 KB block, and 8 jobs run simultaneously. Except for them, default parameters are used.

Figure 9 shows our experimental results. For SR and SW, `EXT4+PFTL`, `F2FS+PFTL` and `AMF` show excellent performance. For sequential I/O patterns, extra live page copies for garbage collection do not occur (see Table 4). Moreover, since all the mapping entries are always kept in DRAM, there are no overheads to manage in-flash mapping entries. Note that these performance numbers are higher than the maximum performance of our SSD prototype due to the buffering effect of `FIO`.

`EXT4+DFTL` and `F2FS+DFTL` show slower performance than the others for SR and SW. This is caused by extra I/Os required to read/write mapping entries from/to NAND flash. In our measurements, only about 10% of them are missing in the in-memory mapping table, but its effect on performance is not trivial. When a mapping entry is missing, the FTL has to read it from flash and to



(a) Postmark(L)



(b) Postmark(H)

Figure 10: Experimental results with Postmark

evict an in-memory entry if it is dirty. While the FTL is doing this task, an incoming request has to be suspended. Moreover, it is difficult to fully utilize I/O parallelism when reading in-flash mapping entries because their locations were previously decided when they were evicted.

The performance degradation due to missing entries becomes worse with random-reads (RR) patterns because of their low hit ratio in the in-memory mapping table – about 67% of mapping entries are missing. For this reason, `EXT4+DFTL` and `F2FS+DFTL` show slow performance for RR. On the other hand, `EXT4+PFTL`, `F2FS+PFTL` and `AMF` exhibit good performance.

RW incurs many extra copies for garbage collection because of its random-writes patterns. `AMF` outperforms all the other schemes, exhibiting the highest I/O throughput and the lowest write amplification factor (WAF) (see Table 4). `EXT4+PFTL` shows slightly lower performance than `AMF`, but its performance is similar to that of `AMF`. In particular, `F2FS+PFTL` shows lower performance than `AMF` and `EXT4+PFTL`. This is because of duplicate storage management by F2FS and the FTL. F2FS has a similar WAF value as `AMF`, performing segment cleaning efficiently. However, extra writes for segment cleaning are sent to the FTL and trigger additional garbage collection at the FTL level, which results in extra page copies.[2]

EXT4 and F2FS with DFTL show worse performance than those with PFTL because of extra I/Os for in-flash mapping-entries management.

Postmark: After the assessments of `AMF` with various I/O patterns, we evaluate `AMF` with `Postmark`, which is a small I/O and metadata intensive workload. To understand how garbage collection affects overall performance, we perform our evaluations with two different scenarios, light and heavy, denoted by `Postmark(L)` and `Postmark(H)`. They each simulate situations where a storage space utilization is low (40%) and high (80%) – for `Postmark(L)`, 15K files are created; for `Postmark(H)`, 30K files are created. For both cases, file sizes are 5K-512KB and 60K transactions run.

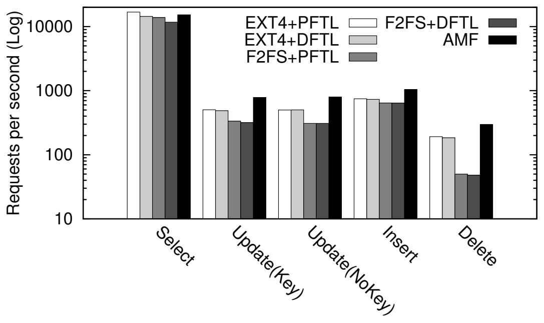
Figure 10 shows experimental results. `F2FS+PFTL` shows the best performance with the light workload, where few live page copies occur for garbage collection (except for block erasures) because of the low utilization of the storage space. `EXT4+PFTL` and `AMF` show fairly good performance as well. For the heavy workload where many live page copies are observed, `AMF` achieves the best performance. On the other hand, the performance of `F2FS+PFTL` deteriorates significantly because of the duplicate management problem. F2FS and EXT4 with DFTL perform worse because of overheads caused by in-flash mapping-entries management.

From the experimental results with `Postmark`, we also confirm that extra I/Os required to manage inode-map segments do not badly affect overall performance. `Postmark` generates many metadata updates, which requires lots of inode changes. Compared with other benchmarks, `Postmark` issues more I/O traffic to inode-map segments, but it accounts for only about 1% of the total I/Os. Therefore, its effect on performance is negligible. We will analyze it in detail Section 5.5.

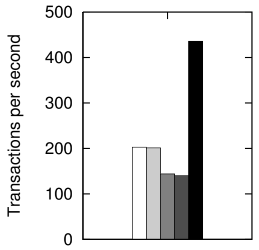
5.3.2 Application Benchmarks

Database Application: We compare the performance of `AMF` using DBMS benchmarks. MySQL 5.5 with an Innodb storage engine is selected. Default parameters are used for both MySQL and Innodb. `Non-Trans` is used to evaluate performance with different types of queries: Select, Update (Key), Update (NoKey), Insert and Delete. The non-transactional mode of a SysBench benchmark is used to generate individual queries [32]. `OLTP` is an I/O intensive online transaction processing (OLTP) workload generated by the SysBench tool. For both `Non-Trans` and `OLTP`, 40 million table entries are created and 6 threads run simultaneously. `TPC-C` is a well-known OLTP workload. We run `TPC-C` on 14 warehouses with 16 clients each for 1,200 seconds.

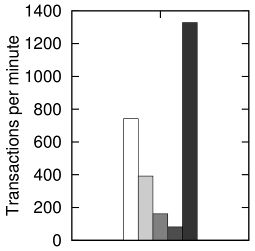
[2]The segment size could affect performance of F2FS – F2FS shows better performance when its segment size is equal to the physical segment (16 MB). However, F2FS still suffers from the duplicate management problem, so it exhibits worse performance than AMF, regardless of the segment size. For this reason, we exclude results with various segment sizes and use the default segment size (2 MB).



(a) Non-Trans



(b) OLTP



(c) TPC-C

Figure 11: Experimental results with database apps.

Figure 11 shows the number of transactions performed under the different configurations. AMF outperforms all other schemes. Compared with the micro-benchmarks, database applications incur higher garbage collection overheads because of complicated I/O patterns. As listed in Table 4, AMF shows lower WAFs than EXT4+PFTL and EXT4+DFTL thanks to more advanced cleaning features borrowed from F2FS. F2FS+PFTL and F2FS+DFTL show similar file-system-level WAFs as AMF, but because of high garbage collection costs at the FTL level, they exhibit lower performance than AMF. The state-of-the-art FTLs used by SSD vendors maybe work better with more advanced features, but it comes at the price of more hardware resources and design complexity. In that sense, this result shows how efficiently and cost-effectively flash can be managed by the application.

Hadoop Application: We show measured execution times of Hadoop applications in Figure 12. Hadoop applications run on top of the Hadoop Distributed File System (HDFS) which manages distributed files in large clusters. HDFS does not directly manage physical storage devices. Instead, it runs on top of regular local disk file systems, such as EXT4, which deal with local files. HDFS always creates/deletes large files (e.g., 128 MB) on the disk file system to efficiently handle large data sets and to leverage maximum I/O throughput from sequentially accessing these files.

This file management of HDFS is well-suited for NAND flash. A large file is sequentially written across multiple flash blocks, and these flash blocks are invalidated together when the file is removed from HDFS. Therefore, FTL garbage collection is done by simply erasing flash blocks without any live page copies. Moreover, because of its sequential access patterns, the effect of missing mapping entries on performance is not significant. This is the reason why all five storage configurations show similar performance for Hadoop applications. The results also indicate that existing flash storage is excessively over-designed. With the exception of error management and coarse-grain mapping, almost all storage management modules currently implemented in the storage device are not strictly necessary for Hadoop.

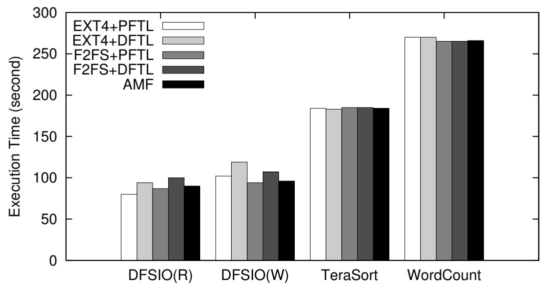


Figure 12: Experimental results with Hadoop apps.

5.4 Lifetime Analysis

We analyze the lifetime of the flash storage for 10 different write workloads. We estimate expected flash lifetime using the number of block erasures performed by the workloads since NAND chips are rated for a limited number of program/erase cycles. As shown in Figure 13, AMF incurs 38% fewer erase operations overall compared to F2FS+DFTL.

5.5 Detailed Analysis

We also analyze the inode-map management overheads, CPU utilizations and I/O latencies.

Inode-map Management Overheads: I/O operations

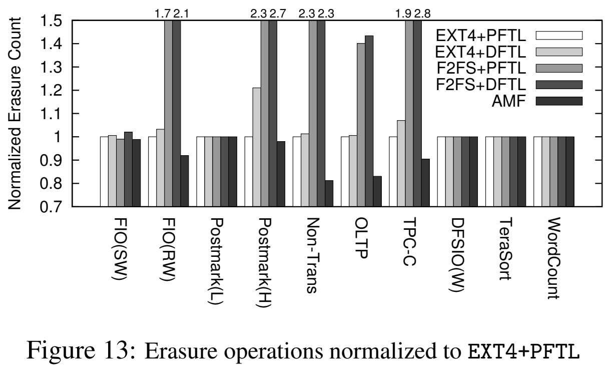
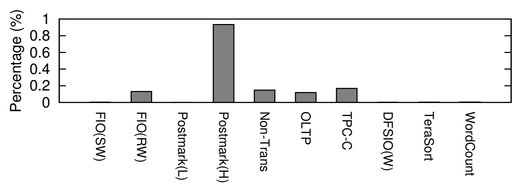
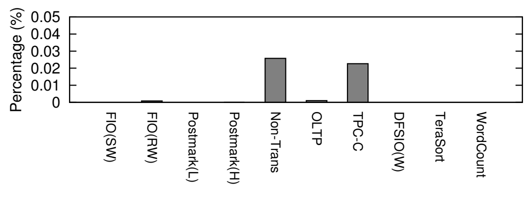


Figure 13: Erasure operations normalized to EXT4+PFTL



(a) TIMB writes



(b) Inode-map garbage collection

Figure 14: Inode-map management overheads analysis

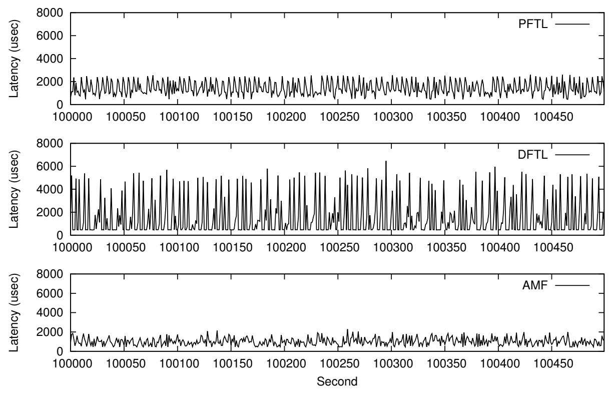


Figure 16: Write latency (μsec)

required to manage inode-map segments in ALFS are extra overheads. Figure 14(a) shows the percentage of TIMB writes to flash storage. We exclude read-only workloads. TIMB writes account for a small proportion of the total writes. Moreover, the number of dirty TIMB blocks written together with a new check-point is small – 2.6 TIMB blocks are written, on average, when a check-point is written. Figure 14(b) illustrates how many extra copies occur for garbage collection in inode-map segments. Even though there are minor differences among the benchmarks, overall extra data copies for inode-map segments are insignificant compared to the total number of copies performed in the file system.

Host CPU/DRAM Utilization: We measure the CPU utilization of `AMF` while running `Postmark(H)`, and compare it with those of `EXT4+PFTL` and `F2FS+PFTL`. As depicted in Figure 15, the CPU utilization of `AMF` is similar to the others. `AMF` does not employ any additional layers or complicated algorithms to manage NAND flash. Only existing file-system modules (F2FS) are slightly modified to support our block I/O interface. As a result, extra CPU cycles required for `AMF` are negligible.

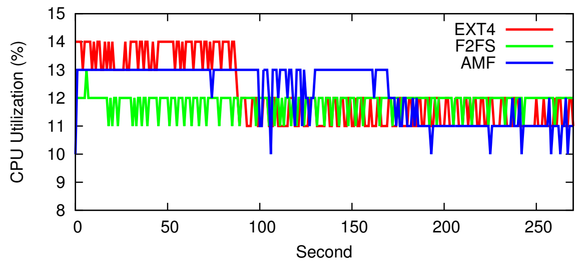


Figure 15: CPU utilization (%)

Host DRAM used by `AMF` is trivial. `AMF` with 16 GB flash requires 180 KB more DRAM than F2FS. Almost all of host DRAM is used to keep AMF-specific data structures (e.g., in-memory TIMB). The host DRAM requirement increases in proportion to the storage capacity, but, as shown in Table 1, it is small enough even for a large SSD (e.g., 10.8 MB for a 1 TB SSD).

I/O Latency: We measure I/O response times of three different FTLs, AMF, PFTL and DFTL, while running `Postmark(H)`. We particularly measure write latencies that are badly affected by both garbage collection and missing mapping entries. As shown in Figure 16, AMF has the shortest I/O response times with small fluctuations since only block erasures are conducted inside the FTL. On the other hand, PFTL and DFTL incur large fluctuations on response times because of FTL garbage collection and in-flash mapping-entries management.

6 Related Work

FTL Improvement with Enhanced Interfaces: Delivering system-level information to the FTL with extended I/O interfaces has received attention because of its advantage in device-level optimization [15, 28, 9, 17]. For example, file access patterns of applications [15] and multi-streaming information [28] are useful in separating data to reduce cleaning costs. Some techniques go one step further by offloading part or all of the file-system functions onto the device (e.g., file creations or the file-system itself) [29, 35, 54]. The FTL can exploit rich file-system information and/or effectively combine its internal operations with the file system for better flash management. The common problem with those approaches is that they require more hardware resources and greater design complexity. In AMF, host software directly manages flash devices, so the exploitation of system-level information can be easily made without additional interfaces or offloading host functions to the device.

Direct Flash Management without FTL: Flash file systems (FFS) [52, 37] and NoFTL [18] are designed to directly handle raw NAND chips through NAND-specific interfaces [51, 22]. Since there is no extra layer, it works efficiently with NAND flash with smaller memory and less CPUs power. Designing/optimizing systems for various vendor-specific storage architectures, however, is in fact difficult. The internal storage architectures and NAND properties are both complex to manage and specific for each vendor and semiconductor-process technology. Vendors are also reluctant to divulge the internal architecture of their devices. The decrease in reliability of NAND flash is another problem – this unreliable NAND can be more effectively managed inside the storage device where detailed physical information is available [13, 43]. For this reason, FFS is rarely used these days except in small embedded systems. AMF has the same advantages as FFS and NoFTL, however, by hiding internal storage architectures and unreliable NAND behind the block I/O interface, AMF eliminates all the concerns about architectural differences and reliability.

Host-Managed Flash: Host-based FTLs like DFS [24, 26, 41] are different from this study in that they just move the FTL to a device driver layer from storage firmware. If log-structured systems like LFS run on top of the device driver with the FTL, two different software layers (i.e., LFS and the FTL in the device driver) run their own garbage collection. As a result, host-based FTLs still have the same problems that the conventional FTL-based storage has.

A software defined flash (SDF) [40] exposes each flash channel to upper layers as individual devices with NAND I/O primitives (e.g., block erasure). Host applications are connected to channels each through a custom interface. In spite of the limited performance of a single channel, it achieves high aggregate throughput by running multiple applications in parallel. SDF is similar to our study in that it minimizes the functionality of the device and allows applications to directly manage the device. This approach, however, is suitable for special environments like the datacenter where aggregate I/O throughput is important and applications can easily access specialized hardware through custom interfaces. AMF is more general – because of compatibility with the existing I/O stacks, if modules that cause overwrites are modified to avoid it, any application can run on AMF.

REDO [34] shows that the efficient integration of a file system and a flash device offers great performance improvement. However, it does not consider important technical issues, such as metadata management affecting performance and data integrity, efficient exploitation of multiple channels, and I/O queueing. REDO is based on a simulation study, so it is difficult to know its feasibility and impact in real world systems and applications.

7 Conclusion

In this paper, we proposed the Application-Managed Flash (AMF) architecture. AMF was based on a new block I/O interface exposing flash storage as append-only segments, while hiding unreliable NAND devices and vendor-specific details. Using our new block I/O interface, we developed a file system (ALFS) and a storage device with a new FTL (AFTL). Our evaluation showed that AMF outperformed conventional file systems with the page-level FTL, both in term of performance and lifetime, while using significantly less resources.

The idea of AMF can be extended to various systems, in particular, log-structured systems. Many DBMS engines manage storage devices in an LFS-like manner [49, 1, 6], so we expect that AMF can be easily adapted to them. A storage virtualization platform could be a good target application where log-structured or CoW file systems [20] coupled with a volume manager [10] manage storage devices with its own indirection layer. A key-value store based on log-structured merge-trees is also a good target application [42, 12, 2]. According to the concept of AMF, we are currently developing a new key-value store to build cost-effective and high-performance distributed object storage.

Acknowledgments

We would like to thank Keith A. Smith, our shepherd, and anonymous referees for valuable suggestions. Samsung (Res. Agmt. Eff. 01/01/12), Quanta (Agmt. Dtd. 04/01/05), and SK hynix memory solutions supported our research. We also thank Jamey Hicks, John Ankcorn and Myron King from Quanta Research Cambridge for their help in developing device drivers. Sungjin Lee was supported by the National Research Foundation of Korea (NRF) grant (NRF-2013R1A6A3A03063762). The work of Jihong Kim was supported by the NRF grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2013R1A2A2A01068260) and the Next-Generation Information Computing Development Program through the NRF funded by the MSIP (NRF-2015M3C4A7065645).

Source code

All of the source code including both software and hardware are available to the public under the MIT license. Please refer to the following git repositories: `https://github.com/chamdoo/bdbm_drv.git` and `https://github.com/sangwoojun/bluedbm.git`.

References

[1] RethinkDB. `http://rethinkdb.com`, 2015.

[2] RocksDB: A persistent key-value store for fast storage environments. `http://rocksdb.org`, 2015.

[3] AXBOE, J. FIO benchmark. `http://freecode.com/projects/fio`, 2013.

[4] BAN, A. Flash file system, 1995. US Patent 5,404,485.

[5] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder – A transactional record manager for shared flash. In *Proceedings of the biennial Conference on Innovative Data Systems Research* (2011).

[6] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*, 2 (2008), 4.

[7] CHANG, L.-P. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the Symposium on Applied Computing* (2007), pp. 1126–1130.

[8] CHANG, Y.-B., AND CHANG, L.-P. A self-balancing striping scheme for NAND-flash storage systems. In *Proceedings of the Symposium on Applied Computing* (2008), pp. 1715–1719.

[9] CHOI, H. J., LIM, S.-H., AND PARK, K. H. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage 4*, 4 (2009), 14:1–14:22.

[10] EDWARDS, J. K., ELLARD, D., EVERHART, C., FAIR, R., HAMILTON, E., KAHN, A., KANEVSKY, A., LENTINI, J., PRAKASH, A., SMITH, K. A., ET AL. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference* (2008), pp. 129–142.

[11] FELDMAN, T., AND GIBSON, G. Shingled magnetic recording areal density increase requires new data management. *USENIX issue 38*, 3 (2013).

[12] GHEMAWAT, S., AND DEAN, J. LevelDB. `http://leveldb.org`, 2015.

[13] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2012).

[14] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 229–240.

[15] HA, K., AND KIM, J. A program context-aware data separation technique for reducing garbage collection overhead in NAND flash memory. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/O* (2011).

[16] HAHN, S., LEE, S., AND KIM, J. SOS: Software-based out-of-order scheduling for high-performance NAND flash-based SSDs. In *Proceedings of the International Symposium on Mass Storage Systems and Technologies* (2013), pp. 1–5.

[17] HAHN, S. S., JEONG, J., AND KIM, J. To collect or not to collect: Just-in-time garbage collection for high-performance SSDs with long lifetimes. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implemenation (Poster)* (2014).

[18] HARDOCK, S., PETROV, I., GOTTSTEIN, R., AND BUCHMANN, A. NoFTL: Database systems on FTL-less flash storage. In *Proceedings of the VLDB Endowment* (2013), pp. 1278–1281.

[19] HITACHI. Hitachi accelerated flash, 2015.

[20] HITZ, D., LAU, J., AND MALCOLM, M. A. File system design for an NFS file server appliance. In *Proceedings of the Winter USENIX Conference* (1994).

[21] HUANG, S., HUANG, J., DAI, J., XIE, T., AND HUANG, B. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proceedings of the International Workshop on Data Engineering* (2010), pp. 41–51.

[22] HUNTER, A. A brief introduction to the design of UBIFS, 2008.

[23] JIANG, S., ZHANG, L., YUAN, X., HU, H., AND CHEN, Y. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies* (2011), pp. 1–12.

[24] JOSEPHSON, W. K., BONGO, L. A., LI, K., AND FLYNN, D. DFS: A file system for virtualized flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2010).

[25] JUN, S.-W., LIU, M., LEE, S., HICKS, J., ANKCORN, J., KING, M., XU, S., AND ARVIND. BlueDBM: An appliance for big data analytics. In *Proceedings of the Annual International Symposium on Computer Architecture* (2015), pp. 1–13.

[26] JUNG, M., WILSON, III, E. H., CHOI, W., SHALF, J., AKTULGA, H. M., YANG, C., SAULE, E., CATALYUREK, U. V., AND KANDEMIR, M. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), pp. 75:1–75:11.

[27] JUNG, S.-M., JANG, J., CHO, W., CHO, H., JEONG, J., CHANG, Y., KIM, J., RAH, Y., SON, Y., PARK, J., SONG, M.-S., KIM, K.-H., LIM, J.-S., AND KIM, K. Three dimensionally stacked NAND flash memory technology using stacking single crystal Si layers on ILD and TANOS structure for beyond 30nm node. In *Proceedings of the International Electron Devices Meeting* (2006), pp. 1–4.

[28] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (2014).

[29] KANG, Y., YANG, J., AND MILLER, E. L. Efficient storage management for object-based flash memory. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2010).

[30] KATCHER, J. PostMark: A new filesystem benchmark. *NetApp Technical Report TR3022* (1997).

[31] KONISHI, R., AMAGAI, Y., SATO, K., HIFUMI, H., KIHARA, S., AND MORIAI, S. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review 40*, 3 (2006), 102–107.

[32] KOPYTOV, A. SysBench: A system performance benchmark. `http://sysbench.sourceforge.net`, 2004.

[33] LEE, C., SIM, D., HWANG, J.-Y., AND CHO, S. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2015).

[34] LEE, S., KIM, J., AND ARVIND. Refactored design of I/O architecture for flash storage. *Computer Architecture Letters 14*, 1 (2015), 70–74.

[35] LEE, Y.-S., KIM, S.-H., KIM, J.-S., LEE, J., PARK, C., AND MAENG, S. OSSD: A case for object-based solid state drives. In *Proceedings of the International Symposium on Mass Storage Systems and Technologies* (2013), pp. 1–13.

[36] LIU, M., JUN, S.-W., LEE, S., HICKS, J., AND ARVIND. minFlash: A minimalistic clustered flash array. In *Proceedings of the Design, Automation and Test in Europe Conference* (2016).

[37] MANNING, C. YAFFS: Yet another flash file system, 2004.

[38] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. SFS: random write considered harmful in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2012).

[39] NAM, E. H., KIM, B., EOM, H., AND MIN, S. L. Ozone (O3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers 60*, 5 (2011), 653–666.

[40] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), pp. 471–484.

[41] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the International Symposium on High Performance Computer Architecture* (2011), pp. 301–311.

[42] ONEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[43] PAN, Y., DONG, G., AND ZHANG, T. Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes. *IEEE Transactions on Very Large Scale Integration Systems 21*, 7 (2013), 1350–1354.

[44] PARK, D., DEBNATH, B., AND DU, D. CFTL: A convertible flash translation layer adaptive to data access patterns. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2010), pp. 365–366.

[45] PHISON. PS3110 controller, 2014.

[46] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage 9*, 3 (2013), 9.

[47] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10* (1991), 1–15.

[48] SAMSUNG. Samsung SSD 840 EVO data sheet, rev. 1.1, 2013.

[49] VO, H. T., WANG, S., AGRAWAL, D., CHEN, G., AND OOI, B. C. LogBase: a scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment* (2012).

[50] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Bluesky: A cloud-backed file system for the enterprise. In *Proceedings of the USENIX conference on File and Storage Technologies* (2012).

[51] WOODHOUSE, D. Memory technology device (MTD) subsystem for Linux, 2005.

[52] WOODHOUSE, D. JFFS2: The journalling flash file system, version 2, 2008.

[53] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *Proceedings of the Workshop on Interactions of NVM/Flash with Operating Systems and Workloads* (2014).

[54] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2012).