A Case for Application-Managed Flash

Jinhyung Koo, Chanwoo Chung, Arvind, *Fellow, IEEE*, and Sungjin Lee

Abstract—We propose a new I/O architecture for NAND flash-based SSDs, called application-managed flash (AMF) and present two case studies to show its usefulness. In a typical SSD controller, an intermediate software layer, called the flash translation layer (FTL), is employed between NAND flash chips and a host interface. The main responsibility of an FTL is to provide interoperability with conventional HDDs, but this interoperability comes at the cost of extra hardware resources and degraded I/O performance. The proposed AMF refactors the flash storage architecture so that an SSD controller exposes append-only segments, which do not permit overwriting. This refactoring dramatically improves performance of applications and reduces hardware costs by allowing applications to directly manage flash storage with minimal supports from the SSD controller. In order to understand the benefits of AMF, we study two popular applications: a log-structured file system (F2FS) and a key-value store (RocksDB). Our experiments show that the DRAM in the flash controller is reduced by 128X and the performances of the file system and the key-value store improve by 80 and 54 percent, respectively, over conventional SSDs.

Index Terms—NAND flash, solid-state disks, file system, key-value store, flash translation layer

1 Introduction

NAND flash SSDs have become the popular storage media in enterprise systems because of its superior performance, low-power consumption, and high capacity. Thanks to the recent advances in semiconductor technologies such as 3D NAND [1] and ZNAND [2], the popularity of NAND flash-based SSDs is likely to grow.

NAND flash-based SSDs have unique characteristics, namely out-of-place updates, limited erasure cycles, and asymmetric read/write latencies. To provide an I/O abstraction of a generic block device, an SSD employs a flash translation layer (FTL) that emulates conventional block I/O operations and media geometry. The main virtue of an FTL is to provide interoperability of legacy applications developed for HDDs, but this interoperability comes at the expense of increased hardware cost and performance loss.

Implementing an FTL requires significant hardware resources in the SSD controller. In particular, tasks such as 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 [3], [4], [5]). An FTL makes important decisions that affects storage performance and lifetime without any awareness of the high-level application, and thus, the resulting performance is often suboptimal [6], [7], [8]. Moreover, an FTL works as a black box—its inner-workings are hidden behind a block interface, which makes the behavior of flash storage unpredictable, for example, unexpected invocation of garbage collection (GC) [9] and swapping of mapping entries [10], [11].

Several attempts have been made to address the fundamental limitations of the typical FTL-based SSD designs. One of the alternatives is an open-channel SSD (OCSSD) [12]. It exposes the unique geometry and operations of an SSD to a host machine via an extended NVMe interface, enabling system software or applications to manage NAND media directly. It addresses some limitations of the FTL approach but creates some other problems. First, it removes the FTL from a flash device, but requires us to run the same FTL functionality on the host. Thus, instead of solving the problems caused by the FTL, it simply shifts the resource burden from the controller to the host. Second, it puts too much burden on developers: application developers have to take into account the unusual physical properties of NAND, such as paired pages and asymmetric I/O units. Moreover, since those properties keep changing as NAND flash technology advances, software, including the kernel and the libraries using OCSSDs, has to adapt to such changes. Finally, semiconductor companies are reluctant to disclose any details of their devices, which has hindered acceptability of OCSSD by major SSD vendors.

We present a different approach to managing flash storage, which we call **A** pplication- **M** anaged **F** lash (AMF). The idea of AMF is in line with OCSSD in that it moves the intelligence of flash management from the device to applications, but unlike OCSSD, it leaves some essential management tasks to the device. In contrast to OCSSD, the I/O interface of AMF, *AMF IO*, is based on the existing block I/O interface; it exposes the same I/O operations (i.e., `READ`, `WRITE`, and `TRIM`) and device geometry (i.e., a linear array of 4-KB blocks). The only difference is that it does *not* support overwrites unless the region to be overwritten is explicitly deallocated first. This new restriction dramatically simplifies the management burden on the device because fine-grained

- *Jinhyung Koo and Sungjin Lee are with the Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu 42988, Republic of Korea. E-mail: {jhk5361, sungjin.lee}@dgist.ac.kr.*
- *Chanwoo Chung and Arvind are with Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: {cwchung, arvind}@mit.edu.*

Manuscript received 30 June 2019; revised 27 Jan. 2020; accepted 16 Mar. 2020. Date of publication 14 Apr. 2020; date of current version 14 Jan. 2021. (Corresponding author: Sungjin Lee.) Recommended for acceptance by M. Kandemir. Digital Object Identifier no. 10.1109/TC.2020.2987569

remapping and GC do not have to be done internally. An *AMF device* exposing AMF I/O is seen by applications as a usual block device, but the applications should take care of avoiding in-place updates and issuing trim commands to indicate that the data has been deallocated. This direct flash management of AMF enables applications to access data with minimal FTL involvement.

The AMF device is responsible for providing error-free storage accesses (e.g., bit-flip errors) and exploiting efficient parallelism of multiple I/O channels accessing NAND flash chips. The device also keeps track of bad blocks and performs wear-leveling. It is preferable to do these operations inside an SSD because they depend on the specific circuit designs and manufacturing process of NAND devices, which device manufacturers prefer not to disclose. In our system, management tasks in the device are performed at a block granularity as opposed to a page granularity, and therefore, the power requirement of the device and the size of mapping tables 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 important applications already employ "append-only" strategies. For example, log-structured or copy-on-write file systems always append new data to the device, mostly avoiding in-place updates [13], [14], [15]. Similar log-structured systems are used in the database community [16], [17]. The LSM-Tree is also a well-known data structure based on a log-structured approach [18], [19], [20]. 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 [21].

To demonstrate the advantages of AMF, we use an open FPGA-based flash platform, BlueDBM [22], [23], as our testbed. We implement a new lightweight FTL called an Application-managed FTL (AFTL) to support our block I/O interface. We carry out case studies on two popular applications, a file system and a key-value store. For a file system, we chose F2FS [15] because it is the most popular log-structured file system, and modified it to satisfy the new restriction of AMF IO. For our case study of a key-value store, we chose RocksDB [19], which is based on the LSM-tree. We provided RocksDB with a new storage engine that is compatible with AMF IO instead of using the POSIX interface of RocksDB. Our experiments with various benchmarks show that the modified F2FS and RockDB improve I/O performance by up to 80 and 54 percent, respectively, enhancing device lifetime by up to 68 and 62 percent, respectively. The DRAM requirement for the FTL is reduced by a factor of 128 while the additional host-side resources required by AMF are minor.

This paper is organized as follows: After describing related work in Section 2, we explain our new block I/O interface in Section 3. Section 4 describes an FTL design for AMF. In Sections 5 and 6, we present our case studies with F2FS and RocksDB in detail. After evaluating AMF with various benchmarks in Section 7, we conclude with a summary and future directions in Section 8.

2 Related Work

We review various approaches that were proposed before to address the problems of the conventional SSD design.

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 [6], [7], [8]. For example, file access patterns of applications [6] and multi-streaming information [7] are useful in separating data to reduce cleaning costs. Some techniques go one step further by offloading a part or all of the file-system functions onto the device (e.g., file creations or the file-system itself) [24], [25]. 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 made easily without additional interfaces or offloading host functions to the device.

Direct Flash Management without FTL. Flash file systems (FFS) [26], [27] and NoFTL [28] are designed to directly handle raw NAND chips through NAND-specific interfaces [29], [30]. 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 [31], [32]. 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 [33], [34] are different from our approach 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 collectors, which results in the amplification of GC overheads [35]. As a result, host-based FTLs still have the same problems that the conventional FTL-based storage has.

A software defined flash (SDF) [36] exposes each flash channel to upper layers as individual devices with NAND I/O primitives (e.g., block erasure). 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 datacenters where aggregate I/O throughput is important and applications can easily access

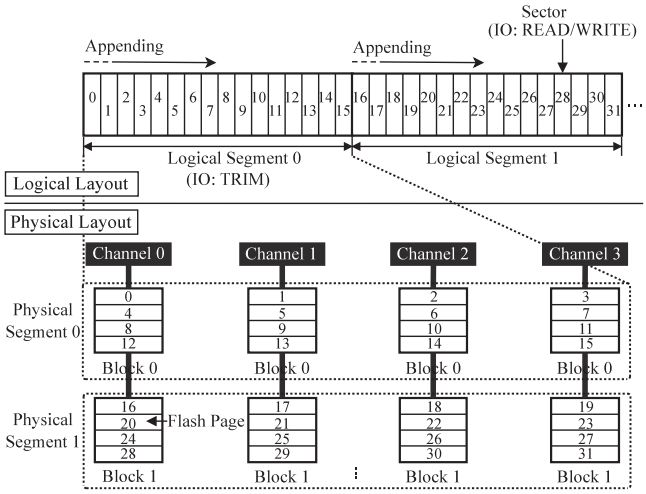


Fig. 1. An AMF block I/O abstraction: It shows two logical segments (logical segments 0 and 1) and two corresponding physical ones on the device side (physical segments 0 and 1). A logical segment is composed of 16 sectors which are *statically* mapped to flash pages. A physical segment is organized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

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 them, any application can run on AMF.

Open-Channel SSD. An open-channel SSD (OCSSD) is another type of a host-managed SSD design which is officially supported by the Linux kernel. Just like other host-managed approaches, it runs on top of a specialized SSD that exposes its internals, including NAND-specific interfaces as well as geometry. To support compatibility with existing applications, it provides the in-kernel FTL module that manages NAND chip arrays. For direct media management, OCSSD offers a user-space I/O library which enables user applications to directly access NAND flash with full understanding of media geometry and operations. OCSSD, however, has the same limitations as DFS and FFS. Like DFS, it is affected from duplicate storage management when the in-kernel FTL is used. Similar to FFS, the host-side management modules have to be revised according to the changes of NAND physics.

3 AMF Block I/O Interface (AMF IO)

Fig. 1 depicts the block I/O abstraction of AMF, showing both logical and physical layouts. AMF IO 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 when an application issues a`TRIM` command on it, which is always issued in the granularity 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. This property does not allow typical applications to run directly on top of AMF IO. To avoid this overwrite problem, the host software should be refactored or designed to write the sectors of a segment in an append-only manner, starting from the lowest sector address. Those sectors can be reused after the segment they belong to has been deallocated.

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. With log-structured systems, a logical segment is used as the unit of free-space allocation and reclamation, where data is *sequentially* appended and is reclaimed later. A physical segment on the storage device side is optimized for such a sequential access by software. In Fig. 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. As will be discussed later, this design greatly simplifies a flash controller design, without sacrificing raw device performance.

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. This design decision was made based on the fact that the lifetime of NAND devices is managed more effectively at lower levels. Besides P/E cycles, the lifetime of NAND devices is affected by factors (such as recovery effects [31] and bit error rates [32]), and those keep changing. Only SSD vendors take these factors into consideration in wear-leveling and bad-block management since such information is proprietary and confidential. 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` on 4 KB sectors). 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. In our Linux implementation, for example, the existing block I/O layer is not changed at all. This makes it easy to convert existing software to run on AMF.

From the perspective of SSD vendors, the design of an FTL for AMF is similar to block FTLs [37] which require minimal functions for flash management. Thus, SSD vendors can easily build AMF devices by removing unnecessary functions from their devices. Alternatively, SSD vendors could make a dual-mode SSD supporting two different modes, device-managed and application-managed modes, which can be chosen by users' preference. For example, by selecting the device-managed mode, any kinds of applications can be run on top of the SSD, and thus good compatibility is offered. A user can choose the application-managed mode for better performance and more stable I/O response times, but in this case, only AMF-compatible applications with 100 percent append-only writes can be mounted to the SSD. Simultaneously running two different types of applications on the same SSD may be technically possible, but we do not consider it in this study.

4 AMF Flash Translation Layer (AFTL)

An architecture of the AMF FTL (AFTL) is similar to a simplified version of the block-level FTL [37], except that it

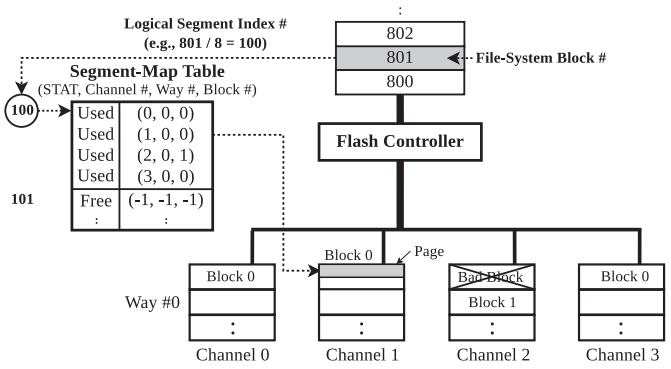


Fig. 2. An example of how AFTL handles writes: There are four channels and one way in AFTL, and each block is composed of two pages. A physical segment has eight pages. When a write request comes, AFTL gets a logical segment index (i.e., 100 = 801/8) using the logical sector number. It then looks up 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. The logical block number '801' is written the OOB area of the page, and this information is used for reconstructing the segment-map table when the table is lost.

does not need to run address remapping to avoid in-place updates, nor does it need to copy valid pages for garbage collection. While we implemented AFTL in a host device driver for this work, the full hardware implementation of AFTL and its benefit have been studied in [38]. For the sake of simplicity, we focus on describing three functions that AFTL must support: wear-leveling, bad-block management, and I/O scheduling.

Wear-Leveling and Bad-Block Management. Sectors in a logical segment are statically mapped to flash pages in a physical segment, and their locations do not change. However, to support wear-leveling and bad-block management, flash blocks of a physical segment must be mapped dynamically depending upon the wearing level (e.g., P/E cycles) as well as the health status (e.g., bad or not) of a flash block. To this end, AFTL maintains a small segment-map table that maps a logical segment to a physical one, and each entry of the table contains the physical locations of flash blocks mapped to a logical segment. Those blocks are striped across channels and ways, and each has a flag (`STAT`) that indicates its status, including Free, Used or Invalid.

Fig. 2 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. It picks up the least worn-out free blocks in the corresponding channel/way, excluding bad ones. 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 [39]. If there are previously allocated flash blocks only with invalid pages (i.e., `STAT` is Invalid), they are erased. In case where a logical segment is already mapped (i.e., `STAT` is Used), the data is written to the fixed location in the physical segment. AFTL is informed through TRIM commands which physical segment has only obsolete data. Upon receiving the TRIM command, AFTL invalidates that segment by changing its `STAT` to Invalid.

Like mapping tables in typical FTLs, the segment-map table is protected by built-in capacitors. This provides us sufficient time to flush out the segment-map table to the flash persistently in the event of sudden power or system failure. In the worst case where the segment-map table is lost, AFTL is able to reconstruct the table by scanning flash blocks' OOB area which keeps logical block numbers.

I/O Queueing and Scheduling. 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. Multi-threaded applications may allocate several segments and write multiple data streams at the same time. In AFTL, data arriving from multiple streams are put into designated queues according to their LBA numbers, and the FIFO schedulers fetch and send them to NAND chips. Write skews do not occur for any channel or way in AFTL, thanks to AMF-aware applications that allocate and write data in the unit of a segment, distributing all the write requests to channels and ways uniformly. Consequently, simple multiple I/O queueing is efficient enough to offer full performance of flash hardware.

AFTL schedules GC I/Os and user I/Os with different priorities. AFTL does not involve any live page copies for GC. But, it has to internally erase a group of flash blocks which are invalidated by TRIM commands. The most straightforward way is erasing associated blocks immediately upon receiving TRIMs commands. This *foreground cleaning*, however, often results in high I/O fluctuation. The elapsed time of erasing blocks is much longer than that of reading or writing pages. Thus, erasing flash blocks on demand seriously interfere with user I/Os. To avoid this, AFTL employs a *background cleaning* policy. It delays performing block erasures and processes user I/Os with a higher priority. When there are not user I/Os and sufficiently long idle time is detected, it triggers actual block erasures. Only when free space is under 5 percent of the total flash space, it invokes foreground GC to reclaim free space.

5 Case Study: Log-Structured File System

In this section, we explain our experience with the design and implementation of an application-managed log-structured file system (ALFS). We implement ALFS in the Linux 3.13 kernel based on an F2FS file system [15]. Most of the data structures and modules of F2FS are left unchanged. Thus, 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. For the sake of generality, we explain the high-level design of ALFS using general terms found in Sprite LFS.

5.1 File Layout and Operations

Fig. 3 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*.

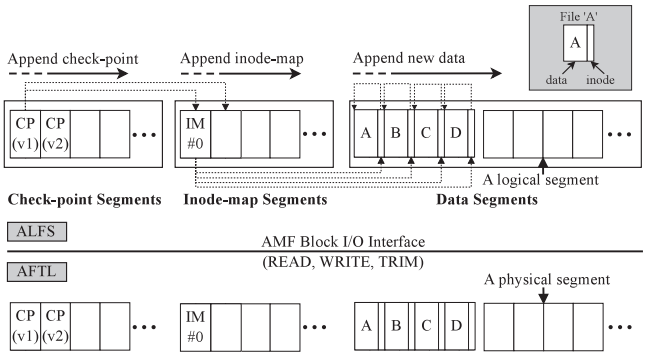


Fig. 3. The upper figure illustrates the logical layout of ALFS. There is an initial check-point CP(v1). Four files are appended to data segments along with their inodes in the following order: A, B, C, and D. Then, an inode map IM#0 is written which points to the locations of the inodes of the files. 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.

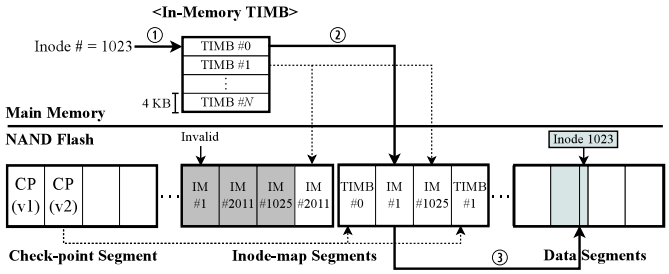


Fig. 4. To find an inode, ALFS first looks up in-memory TIMB to find the location of inode-map blocks that points to the inode in flash. Each 4 KB TIMB block indicates 1,024 inode-map blocks in inode-map segments (e.g., TIMB#0 points to IM#0∼IM#1023 in flash). Since each inode-map block points to 512 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.

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 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 search of inodes.

5.2 Check-Point Segment

The management of check-point segments is straightforward. ALFS reserves *two fixed* logical segments whose indices are #1 and #2 for check-points. A logical segment 0 is reserved for a superblock. ALFS appends new check-points with incremental version numbers to the segments. If free space is exhausted, the segment containing only old check-point versions is selected as a victim for erasure. The latest check-point is still kept in the other segment. ALFS sends TRIM commands to invalidate and free the victim, and then it switches to the freed segment and keeps writing new check-points. Even if ALFS uses the same logical segments repeatedly, it will not unevenly wear out flash owing to wear-leveling in AFTL. When ALFS is remounted, it scans all the check-point segments and looks for the latest check-point by referring to version numbers. This brute force search is efficient because only two segments are maintained 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. Be advised that, regardless of how large an SSD is, only two logical segments are enough for bookkeeping check-points.

5.3 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 ($= 8\ \text{B} \times 2^{28}$). 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. For example, IM#0 in Fig. 3 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. Fig. 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 ($= 4\ \text{B} \times 524,288$), 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 during 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 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 in-memory TIMB size is 2 MB, the number of TIMB blocks is 512 ($= 2\ \text{MB}/4\ \text{KB}$). If 4 B is large enough to point to locations of TIMB blocks, the TIMB-blocks list is $2\ \text{KB}$ ($= 4\ \text{B} \times 512$). A check-point requires hundred bytes (e.g., 193 bytes in F2FS), so a TIMB-block list can be written together with a check-point 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 [13].

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 slightly 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. 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). This decision is made empirically through experiments with various applications. The space wasted by extra segments is small (e.g., $0.78\% = 8\ \text{GB}\ /\ 1\ \text{TB}$). 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 percent of the total I/Os.

5.4 Data Segment

The rest of logical segments are used as data segments, and this decides the maximum capacity that ALFS can actually use for storing user files and directories. 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. 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 (e.g., hot-cold separation) from F2FS [15] without any effort.

6 Case Study: Key-Value Store

We present another case study: the RocksDB [19] key-value store. RocksDB uses a log-structured merge tree (LSM-tree) [18], and was chosen because it is the most popular LSM-tree-based KVS implementation. We begin by explaining the overall organization and operations of RocksDB and then show how we modified its design to work with AMF.

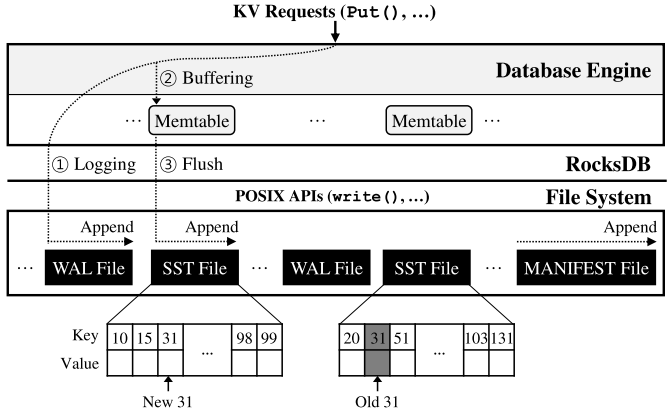


Fig. 5. The write operation of RocksDB, along with three major files (WAL, SST, MANIFEST files). When a KV request comes, it is first logged into an WAL file and buffered in a memtable. Once the memtable becomes full, key-value pairs are flushed to an SST file.

6.1 RocksDB Files and Operations

RocksDB runs on top of a file system, such as EXT4, maintaining three major type of files: *WAL*, *MANIFEST*, and *SST* as shown in Fig. 5. When new key-value requests arrive, RocksDB first buffers them in a special data structure, called a *memtable*, which resides in the main memory. To provide data consistency and durability in the event of sudden power/hardware failure, RocksDB uses the write-ahead logging (WAL) technique which writes incoming key-value pairs to a WAL file. As in other logging techniques, all the logging data is written to a WAL file sequentially. Once a memtable becomes full, it is persistently flushed out to an SSD as an SST file. This flushing operation is done by sequentially writing the contents of a memtable to an SST file. Then, a WAL file can be removed. To improve I/O throughput, RocksDB often maintains multiple memtables in the memory, and, in that case, each memtable has its own WAL file.

An SST file is like a huge table containing key-value pairs which are sorted by keys. When a duplicate key with a new value arrives, the pair is written to a new SST file rather than an in-place update on the old one (e.g., the new value of the key 31 in Fig. 5). This is due to the LSM-tree's immutable property which does not allow in-place updates of existing values. Those obsolete values are discarded later when RocksDB performs *compaction* which merges and sorts key-value pairs from more than two SST files, creating new sorted ones containing only unique key-value pairs [18]. The behavior of compaction is similar to FTL garbage collection, which implies that RocksDB would be able to replace the major functionality of the FTL.

While RocksDB is performing its operations (e.g., flushing and compaction), several WAL and SST files are created and removed in a file system. If system failure suddenly happens, it may be impossible to return to the last consistent state. For this reason, RocksDB maintains another transactional log, a MANIFEST file, which keeps track of all of the changes in files. Whenever new WAL or SST files are added and/or old files are removed, all of this information is recorded in a MANIFEST file in an append-only manner. A MANIFEST file also keeps the latest consistent state, so it

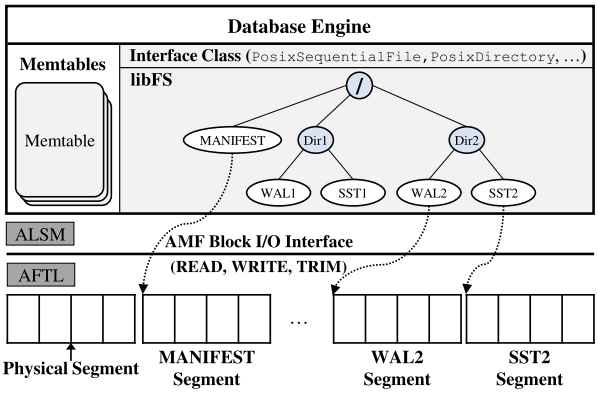


Fig. 6. Directory and file management of libFS in ALSM.

can be used to quickly restore RocksDB to the latest known consistent state on a restart.

The important files in RocksDB are all written in an append-only manner, which is well suited to AMF. However, RocksDB accesses files through POSIX APIs as well as a kernel file system (e.g., EXT4) which manages the underlying media in an in-place update manner. One might think that, by using ALFS instead of EXT4, in-place updates can be eliminated. Unfortunately, this is not so because the direct use of ALFS causes another problem, i.e., double logging [35]. The LSM-tree appends key-value pairs to SST files like a log, and, at the same time, ALFS also manages all the files as a huge log. It is reported that such double logging incurs amplified cleaning overheads [35].

To overcome the above problem, we designed and implemented a new LSM-tree based key-value store, which we call Application-Managed LSM-tree (ALSM). Similar to our study with F2FS, we haven't changed any fundamental parts of RocksDB, but improve its storage engine module so that it satisfies the design principle of AMF.

6.2 libFS: User-Space File Interface

To make ALSM directly access AMF devices, we develop a user-space file-system library, *libFS*, which implements the essential file-system features that are needed by RocksDB's core module. libFS receives file and directory I/Os from the core module and then delivers them to AMF devices directly, bypassing a kernel file system. RocksDB creates large files (e.g., 32 MB∼256 MB) which are read and written sequentially and are deleted at once. Thanks to such simple I/O access patterns on files, the implementation of libFS does not require much effort.

Fig. 6 shows the overall architecture of ALSM with libFS. The libFS library is designed to provide a virtualized file/directory view, maintaining all the related metadata in the main memory. It exposes the same interfaces that RocksDB uses to create/delete/traverse directories and to read/write files (e.g., `PosixSequentialFile`, and `PosixDirectory` classes). Hence, no modification of the internal DB engine is required.

libFS internally builds a tree representing a directory structure. Also, a list of free logical segments is maintained by libFS. Whenever a new file is created, this file is attached to a designated directory as a new node. A free logical segment, which is selected from the free list, is assigned to the newly created file. Using standard `read()` and `write()` system calls, data can be read from and written to the corresponding physical segment in an AMF block device. Some important properties like a current file offset and a file size are maintained in the tree. When an existing file is to be removed, it is deleted from the tree and the corresponding segment is trimmed. In this way, libFS is able to support all of the operations relevant to directories and files required by the DB engine.

All of the file/directory information of libFS is kept in DRAM, and would be lost in case of a system failure. Fortunately, all the changes made to files are logged in a MANIFEST file. Thus, by scanning a MANIFEST file when RocksDB restarts, we are able to build the latest and consistent directory/file structure. A MANIFEST file also has snapshots and thus the rebuilding process can be done quickly. In the following subsection, we explain how ALSM manages a MANIFEST file.

6.3 MANIFEST File

A MANIFEST file starts with an initial state (i.e., a snapshot) that captures the file/directory state at which the MANIFEST file is created. The information of newly created and removed files are then appended. When RocksDB creates a new MANIFEST file, ALSM allocates an empty segment from the free segment list of libFS. Currently, a MANIFEST file size is set the same as a logical segment size (i.e., 32 MB). When RocksDB requests writing a new event log, ALSM appends it to the currently allocated segment. Once a MANIFEST file reaches its maximum size, RocksDB scans it and creates the latest snapshot which is written to the beginning of a new MANIFEST file. After that the old MANIFEST file can be removed from the file system and its corresponding segment can be freed.

The original RocksDB leaves the changed state of a file along with an event type in a MANIFEST file (e.g., `Add Dir2/SST2` and `Remove Dir2/SST2`). This is enough to create directory and file nodes for libFS when ALSM restarts. But, it does not contain any mapping information between SST/WAL files and logical segments, that is, the locations of logical segments where SST/WAL file's data is stored. To resolve this, ALSM leaves logical segment indices associated with the new WAL or SST file in a MANIFEST file (e.g., `Add Dir2/SST2 Seg3`).

A MANIFEST file itself can be stored in any logical segment. To quickly rebuild libFS without full scanning, the locations of MANIFEST files should be kept in a fixed area in the flash. Similar to the idea of the check-point segment in ALFS, two logical segments whose indices are #1 and #2 are dedicated to keeping track of segment indices of latest MANIFEST files. Therefore, just by scanning them, ALSM is able to locate the segments holding latest MANIFEST files.

6.4 WAL File

A WAL file is managed in a similar manner as a MANIFEST file. New key-value pairs are appended to a WAL file, and, when a recovery is necessary, all of the recorded key-value pairs are scanned sequentially from the beginning of a WAL file. Recall that RocksDB maintains multiple memtables, each of which is paired with its own WAL file. Whenever a new memtable and a corresponding WAL file are

created, ALSM allocates and assigns a new segment to the WAL file. The allocated segment is freed when the WAL file is removed after the memtable is materialized to an SST file.

A key technical issue associated with the management of a WAL file is a proper selection of a memtable size. A WAL file is a persistent copy of a memtable, so it contains the same key-value pairs that a memtable has. Some might think that, by setting a memtable size equal to a logical segment size, we are able to fully utilize the available space of a segment for logging. However, the actual size of a WAL file is slightly larger than that of a memtable. This is because additional metadata (e.g., CRC codes) should be embedded into a WAL file, which is required to reconstruct a memtable later. For example, suppose that a memtable size is set the same as a logical segment size, say 32 MB. In that case, the resulting WAL file is slightly larger than 32 MB owing to the metadata, thus requiring two logical segments. Eventually, the one segment is fully filled with data, but the other one is seriously underutilized. When the WAL file is deleted, the second segment has to be reclaimed even though only few pages (in the physical segment) are used. This could result in frequent block erasures belonging to the second segment, lowering overall storage lifetime.

We worked around the above problem by setting the memtable size to be slightly smaller than that of a segment. Currently, a memtable size is set to 93 percent of a segment size. This simple remedy gets rid of almost all of the underutilized segments for WAL files.

6.5 SST File

The majority of write requests (e.g., about 79 percent) generated by RocksDB are destined for SST files, which keep key-value pairs persistently. Similar to data segments in LFS, RocksDB always writes large amounts of data buffered in a memtable to an SSD at once, which creates an optimal I/O pattern for an AMF device to deal with. Even more, during compaction, RocksDB reads a bunch of data from SST files, sorts them, and writes them to new SST files. This implies that compaction only generates sequential reads and writes which can be efficiently handled by an AMF device.

Note that an SST file is also a persistent copy of a memtable, so it could suffer from the segment space underutilization problem. This is due to that fact that an SST file keeps metadata, such as a header and bloom filters. Fortunately, with a reduced memtable size (i.e., 93 percent of a logical segment), ALSM is able to avoid the segment underutilization problem.

7 Experimental Results

We begin our analysis by comparing the memory requirements of AFTL, ALFS, and ALSM, with commonly used FTL schemes. We then evaluate the performance of ALFS and ALSM using micro and realistic benchmarks to understand their behavior under various I/O access patterns.

7.1 Comparison of Memory Requirements

We compared the mapping table sizes of AFTL with two representative FTL schemes: block-level 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 used in low-end devices like SD cards [40]. Page-level FTL performs mapping on flash pages (4-16KB), showing higher performance. AFTL maintains the segment-map table pointing to flash blocks for wear-leveling and bad-block management.

TABLE 1
A Summary of Memory Requirements

Capacity	Block-level FTL	Page-level FTL	DFTL	AMF		
				AFTL	ALFS	ALSM
512 GB	4 MB	512 MB	102 MB	4 MB	5.3 MB	5.6 MB
1 TB	8 MB	1 GB	204 MB	8 MB	10.8 MB	11.2 MB

Table 1 lists the mapping table sizes of 512 GB and 1 TB SSDs. For the 512 GB SSD, the mapping table sizes are 4 MB, 512 MB and 4 MB for block-level, page-level FTLs and AFTL, respectively. The mapping table sizes increase in proportion to the storage capacity, i.e., for 1 TB storage, block-level, page-level FTLs and AFTL require 8 MB, 1 GB and 8 MB memory, respectively. AFTL requires less than 1 percent of memory for mapping compared to the page-level FTL, which is widely used in SSDs, enabling us to keep all mapping entries in small DRAM even for the 1 TB SSD.

Table 1 also shows the host DRAM requirement for ALFS and ALSM. ALFS maintains various data structures, including tables for inode-map blocks (TIMB), but it consumes a tiny amount of host DRAM. Similarly, ALSM only keeps the file-system metadata (i.e., directories and files) in the host, so it consumes 11.2 MB of DRAM when 1 TB key-value pairs are written to it.

7.2 Experimental Results: ALFS

In this subsection, we present our experimental results with ALFS, comparing them with various FTLs and file systems.

7.2.1 Experimental Setup

We used a Xeon server with 24 1.6 GHz cores and 24 GB DRAM as the host machine. The SSD prototype had 8 channels and 4 ways with 512 GB of NAND flash, comprising 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 is mostly decided by I/O characteristics (e.g., data locality and I/O patterns), not by storage capacity. The host DRAM was limited to 1.5 GB to ensure that requests were not entirely served from the page cache.

7.2.2 Benchmarks

To understand the effectiveness of AFTL, we compared it with two file systems, EXT4 and F2FS [15], running on top of two different FTL schemes, page-level FTL (PFTL) and DFTL [9], which are denoted by `EXT4+PFTL`, `EXT4+DFTL`, `F2FS+PFTL`, and `F2FS+DFTL`, respectively. PFTL is based on page-level mapping that maintains all the mapping entries in DRAM. In practice, the mapping table is too large to be kept in DRAM. To address this, DFTL stores all the mapping entries in flash, keeping only popular ones in DRAM. While DFTL reduces the DRAM requirement, it incurs extra I/Os to

TABLE 2
A Summary of Benchmarks

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

read/write mapping entries from/to NAND flash. We set the DRAM size so that the mapping table size of DFTL was 20 percent of PFTL. For both PFTL and DFTL, greedy garbage collection was used, and an over-provisioning area was set to 15 percent of the total capacity. Note that the over-provisioning area was not necessary for AFTL because it did not perform garbage collection.

For EXT4, a default journaling mode was used and the discard option was enabled to use TRIM. For F2FS, the segment size was always set to 2 MB which was the default size. The segment size for ALFS 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 percent of file-system space was used as an over-provisioning area which was the default value.

We evaluated ALFS using five workloads (see Table 2), spanning two categories: file-system and DBMS. To understand the behaviors of AMF under various file-system operations, we conducted a series of experiments using well known file system benchmarks, FIO [41] and Postmark [42]. We also evaluated AMF using response time sensitive database workloads: Non-Trans, OLTP and TPC-C.

7.2.3 Performance Analysis

FIO. We evaluated sequential and random read/write performance using the FIO benchmark. FIO first wrote a 10 GB file and then performed sequential-reads (SR), random-reads (RR), sequential-writes (SW) and random-writes (RW), separately. We used a libaio I/O engine, 128 io-depth, and a 4 KB block, and 8 jobs ran simultaneously. Except for them, default parameters were used.

Fig. 7 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 3). 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. This because FIO uses the rest of the main memory to buffer file data, avoiding actual I/Os. 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 percent 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 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 determined when they were being evicted.

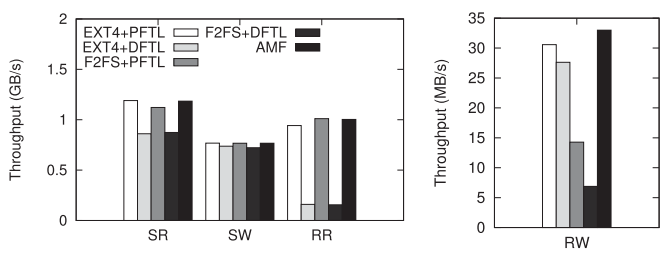


Fig. 7. Experimental results with FIO.

TABLE 3
Write Amplification Factors (WAF)

	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

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.

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 percent 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. AMF outperforms all the other schemes, exhibiting the highest I/O throughput and the lowest write amplification factor (WAF) (see Table 3). EXT4+PFTL shows slightly lower performance than AMF, but its performance is similar to that of AMF. 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. 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 evaluated AMF with Postmark, which is a small I/O and metadata intensive workload. To understand how garbage collection affects overall performance, we performed our evaluations with two different scenarios, light and heavy, denoted by Postmark(L) and Postmark(H). They simulated situations where a storage space utilization was low (40 percent) and high (80 percent). While Postmark(L)

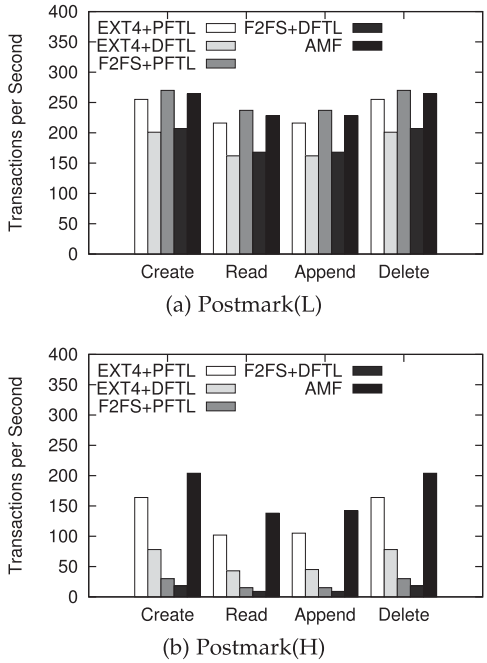


Fig. 8. Experimental results with postmark.

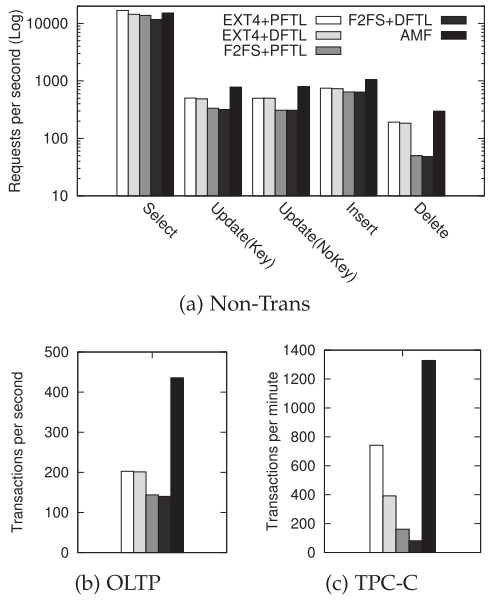


Fig. 9. Experimental results with database apps.

created 15K files, `Postmark(H)` created 30K files. Each file size was 5K-512 KB and 60K transactions were issued in both cases.

Fig. 8 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 require 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 percent of the total I/Os. Therefore, its effect on performance is negligible.

Database Application. We compared the performance of `AMF` using DBMS benchmarks. MySQL 5.5 using Innodb was selected for benchmarking and default parameters were used. `Non-Trans` compared the performance of different types of queries: Select, Update (Key), Update (NoKey), Insert and Delete. The non-transactional mode of a SysBench benchmark was used to generate individual queries [43]. For transactions, `OLTP`, an I/O intensive online transaction processing workload generated by the SysBench tool, was used. For both `Non-Trans` and `OLTP`, 40 million table entries were created and 6 threads ran simultaneously. `TPC-C` is a well-known OLTP workload. We ran `TPC-C` on 14 warehouses with 16 clients each for 1,200 seconds.

Fig. 9 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 3, `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 may work better with more advanced features, but it comes at the price of more resources and complexity. In that sense, this result shows how efficiently and cost-effectively flash can be managed by the application.

7.2.4 Lifetime Analysis

We analyzed the lifetime of the flash storage using 10 different write workloads. We estimated the 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 Fig. 10, `AMF` incurs 38 percent fewer erase operations overall compared to `F2FS+DFTL`.

We have also carried out in-depth analysis with ALFS, evaluating its inode-map management overheads, CPU utilizations and I/O latency. Owing to the limited space, the analysis is omitted in this paper. For details, instead, please refer to [44].

7.3 Experimental Results: ALSM

In this subsection, we present our experimental results with ALSM, a LSM-tree based key-value store with AMF. We focus on analyzing the impact of intermediate layers, including a kernel file system and an FTL, on performance.

7.3.1 Experimental Setup

For our experiments, we used a host system equipped with Intel's 8-core CPUs running at 3.4 GHz and 20 GB DRAM. We used a DRAM-emulated SSD that modelled I/O latency and I/O throughput of our SSD prototype. Similar to our

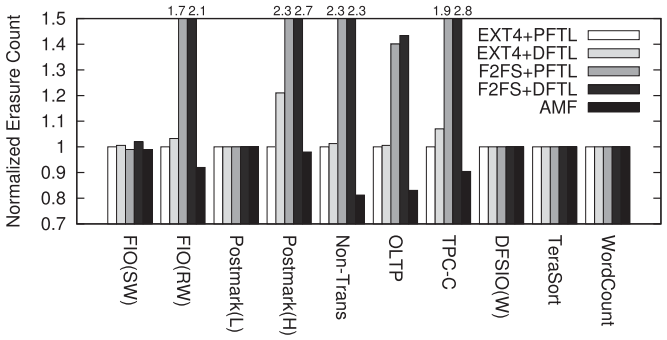


Fig. 10. Erasure operations normalized to EXT4+PFTL.

study on ALFS, for fast evaluation under aged SSDs, we intentionally used a small SSD whose capacity was 16 GB. The rest of DRAM, which was about 4 GB, was used to execute KVS systems as well as benchmark programs.

We used two benchmark programs, db_bench and YCSB . db_bench is a RocksDB-provided benchmark program that is used to evaluate the performance of key-value stores under various access patterns, such as random read/write and sequential read/write. It is useful to assess the basic performance characteristics of key-value stores. The initial database size was set to 4 GB, and the number of key-value requests generated was 1,000K. Default key and value sizes were 16 bytes and 4 KB, respectively.

YCSB is a realistic key-value benchmark that mimics a set of workloads that are commonly observed in cloud service systems. We used five workloads (YCSB-A, B, C, D, and F) which have different I/O access patterns, in terms of a read-write ratio and an operation type (e.g., READ, UPDATE, INSERT, and Read-Modify-Write (RMW)). We used ARDB [45] to run the YCSB workloads on top of RocksDB. Table 4 summarizes the key parameters of the five YCSB workloads. The default value size was set to 4 KB, but we also evaluated KVS performance with various value sizes in Section 7.3.2.

To generate sufficient I/O traffic that was able to fully saturate I/O bandwidth, for both db_bench and YCSB, 10 worker threads were created and ran simultaneously. Note that, after the execution of db_bench and YCSB, the database size grew up to 15.2 GB, which was almost the same as the SSD capacity.

7.3.2 Performance Analysis

To understand the impact of ALMS on performance, we analyzed four different KVS configurations that are summarized in Table 5. The original RocksDB (denoted by RocksDB) ran

TABLE 4
Key Parameters of the five YCSB Workloads

	YCSB				
	A	B	C	D	F
# of READ ops	2M	3.8M	4M	3.8M	2M
# of INSERT ops	-	-	-	0.2M	-
# of UPDATE ops	2M	0.2M	-	-	-
# of RMW ops	-	-	-	-	2M
R:W ratio	50:50	95:5	100:0	95:5	50:50

TABLE 5
KVS Configurations

	RocksDB	FS+ALSM+PFTL	ALSM+PFTL	ALSM
File System	EXT4	libFS + EXT4	libFS	libFS
FTL	PFTL	PFTL	PFTL	AFTL

on top of the EXT4 file system, and the conventional page-level FTL (PFTL) was used. Since RocksDB accesses data in the SSD through the file system and the page-level FTL, it is useful to understand how much the two intermediate layers affect the performance.

FS+ALSM+PFTL used libFS to manage RocksDB files (e.g., SST files) but still operated on the EXT4 file system. It created a single file (rocksdb.dat) on EXT4 and accessed it as if it was a block device. On the SSD side, the page-level FTL was employed to manage the flash. By using libFS, FS +ALSM+PFTL generated more flash-friendly I/Os which were identical to ALSM, but these I/Os were redirected by the kernel file system as well as the page-level FTL, which created inefficiency.

ALSM+PFTL operated without the file system; it opened the SSD as a block device, split it into multiple segments, and ran ALSM on them, accessing data via libFS. However, the SSD still used the page-level FTL. Therefore, it enabled us to measure the impact of the typical FTL. Conversely, compared to FS+ALSM+PFTL, ALSM+PFTL didn't rely on EXT4, which made it possible for us to assess the benefit of bypassing the kernel file system.

ALSM was the final KVS configuration, where ALSM executed directly on the AMF device, bypassing both the file system and the page-level FTL.

db_bench. Fig. 11 displays the throughputs (IOPS) of the four KVS configurations, RocksDB, FS+ALSM+PFTL, ALSM +PFTL, and ALSM, under four different I/O patterns, Fill-Seq, FillRand, ReadSeq, and ReadRand. While Fill-Seq and FillRand wrote 1,000K values in sequential or random key order, respectively, ReadSeq and ReadRand read all the values (previously written) sequentially or randomly, respectively.

In comparison to RocksDB, three KVS configurations, FS +ALSM+PFTL, ALSM+PFTL and ALSM, exhibit better performance. FS+ALSM+PFTL shows slightly higher throughputs than RocksDB for FillSeq and FillRand where many writes are observed. libFS only maintains a single file in EXT4 and internally manages RocksDB data structures on it. This causes less write traffic to the SSD. However, since it

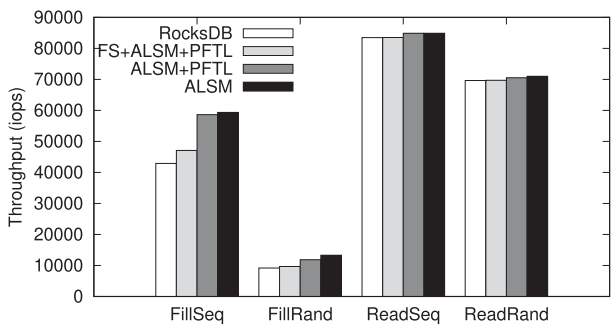


Fig. 11. I/O throughput of db_bench.

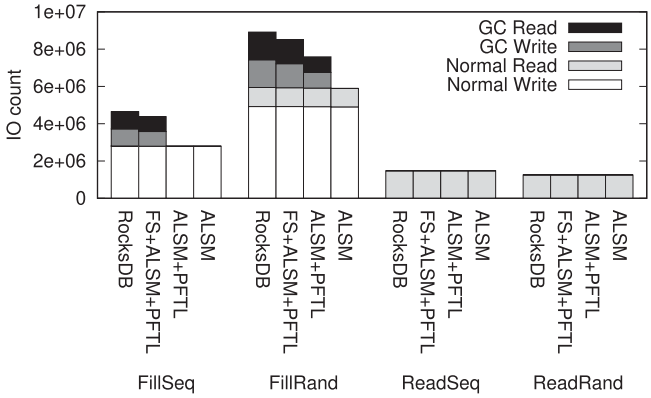


Fig. 12. Analysis of I/O counts of db_bench by type.

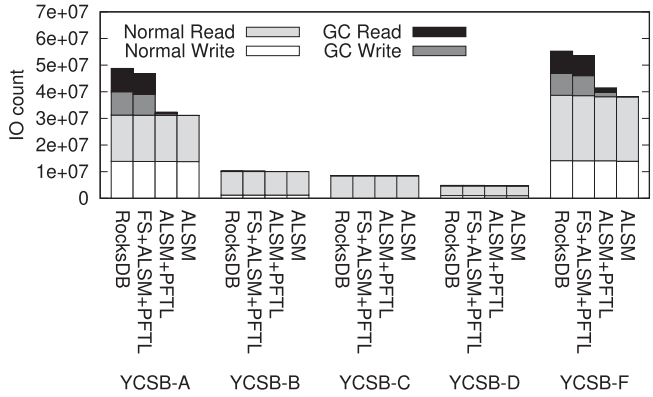


Fig. 14. Analysis of I/O counts of YCSB by type.

still relies on EXT4, extra I/Os created by the kernel file system, such as metadata and journaling I/Os, cannot be completely eliminated.

ALSM+PFTL performs much better than RocksDB and FS+ALSM+PFTL for FillSeq and FillRand, respectively. This benefit is obtained by removing the kernel file system, which results in the reduction of GC overheads on the SSD side.

RocksDB with EXT4 generates write requests not only to MANIFEST, WAL, and SST files, but also to metadata (e.g., inodes) and a journal area. We observe that metadata and journaling data are often mixed up with other RocksDB files in the same flash block. Metadata and journaling data have short lifetime times, so they become invalid sooner than other RocksDB files that stay longer in the flash block. The mixture of valid and invalid pages in the same block causes many valid page copies during SSD GC. ALSM+PFTL uses libFS, bypassing the kernel file system, so it is never affected from such metadata and journaling writes.

Fig. 12 summarizes the number of I/O operations according to their types. Normal reads and writes are I/Os issued by RocksDB to deal with key-value requests, while GC reads and writes are the ones that are invoked to perform GC inside the SSD. Fig. 12 does not display the number of block erasure operations performed by the SSD. We analyze this in Section 7.3.3 in detail.

ALSM+PFTL exhibits a smaller number of GC reads and writes than RocksDB and FS+ALSM+PFTL. ALSM further improves I/O performance by completely removing FTL-level GC operations as shown in Figs. 11 and 12. ALSM+PFTL still suffers from FTL GC because data from different RocksDB files are often written to same flash blocks. ALSM+PFTL relies on libFS which isolates data from different files into separate logical segments. However, the page-level FTL running inside the SSD is unaware of such file-to-segment mapping, and writes incoming data to flash blocks according to their arrival times. This causes different files to be written to the same flash block. For example, given two RocksDB files, WAL-1 and SST-1, ALSM+PFTL assigns them to two different logical segments which have different LBA ranges (e.g., $0 \sim (N_s - 1)$ for WAL-1 and $N_s \sim (2N_s - 1)$ for SST-1, where N_s is the number of LBAs belonging to a logical segment). If WAL-1 and SST-1 are being written by RocksDB threads simultaneously, data from the WAL and SST files can be mixed up in the same flash block. The lifetime of WAL files is shorter than that of SST files, and thus, they become invalid sooner than SST files. ALSM can eliminate such inefficiency by strictly separating them across different flash blocks at the FTL level. As depicted in Fig. 12, there are no I/Os for FTL-level GC.

For ReadSeq and ReadRand, there are no benefits of using ALSM because both are read-only workloads with almost zero writes.

YCSB. Fig. 13 shows the KV performance using realistic cloud-serving workloads generated by YCSB. Similar to what we have observed in db_bench, for write-intensive workloads, YCSB-A and YCSB-F, ALSM outperforms RocksDB by 54 and 28 percent, respectively. ALSM exhibits 7.6 and 6.7 percent better throughputs than ALSM+PFTL by eliminating FTL-level GC I/Os. Fig. 14 which analyzes the number of I/Os according to their types shows consistent results with those previously observed in db_bench.

To understand how much value sizes affect performance, we carried out additional experiments with various value sizes, ranging from 512 B to 4 KB. YCSB-A was used for this evaluation. As presented in Fig. 15, ALSM exhibits the best performance among all the configurations. However, it is evident that the performance benefit decreases as the value size gets smaller. We observe that, with smaller values, RocksDB receives a larger number of KV requests, and thus has to handle more requests per second. This requires nontrivial CPU cycles, moving the bottleneck from I/O to CPU. For example, when the value size is 4 KB, the CPU utilization is about 51 percent, but it increases to 84 percent with 512 B values. Consequently, the benefit of improving system performance with ALSM becomes relatively small.

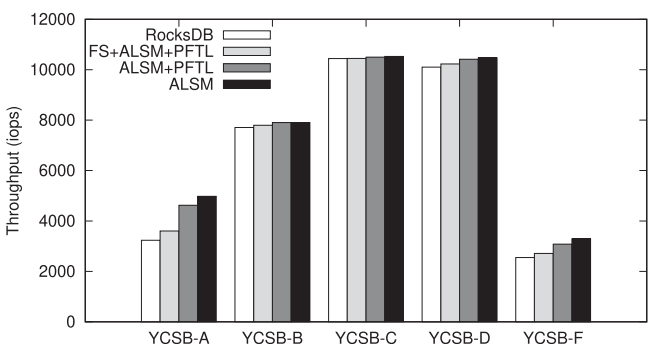


Fig. 13. I/O throughput of YCSB.

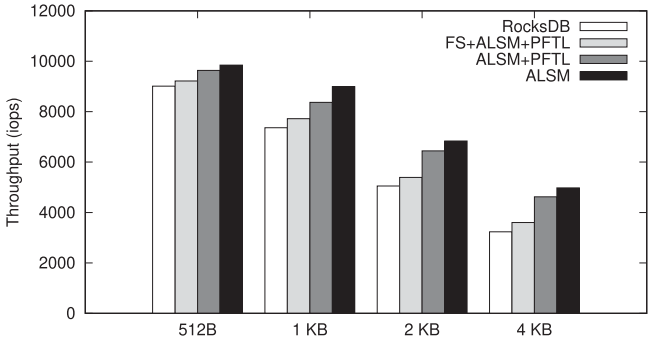


Fig. 15. Impact of value sizes on performance.

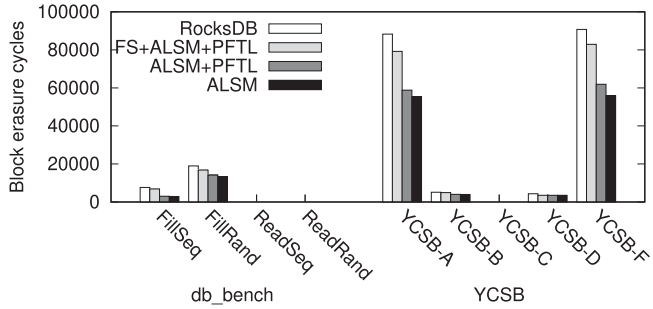


Fig. 17. Comparison of block erasure cycles of RocksDB, FS+ALSM +PFTL, ALSM+PFTL, and ALSM.

I/O Fluctuation. Using YCSB-A with 50 percent writes, we carried out a detailed analysis to understand the impact of individual KVSes on I/O fluctuation. Fig. 16 shows our experimental results. RocksDB, FS+ALSM+PFTL, and ALSM +PFTL are based on the page-level FTL with background cleaning. ALSM-FG is the same as the proposed ALSM, but uses the foreground cleaning policy that immediately erases obsolete flash blocks upon receiving TRIM commands. On the other hand, ALSM uses the background cleaning policy.

Among all the KVSes, ALSM exhibits the most stable I/O throughput, providing the shortest benchmark execution time, 830 seconds. Owing to its foreground cleaning policy, ALSM-FG seriously suffers from high throughput fluctuation. As pointed out earlier, the erasures of multiple flash blocks take long time, so it interferes with incoming user I/Os frequently.

Another noticeable observation found in Fig. 16 is that I/O stability and throughput get improved as the intermediate layers are removed. Compared to FS+ALSM+PFTL, ALSM +PFTL shows more stable and higher throughput by removing the redirection at the file system level. ALSM eliminates both the file system and the FTL, which makes it possible that RocksDB data structures created by libFS are perfectly aligned with NAND flash blocks in the SSD. Again, this perfect data alignment removes I/Os for GC in the SSD.

7.3.3 Lifetime Analysis

Finally, we analyzed the impact of ALSM on SSD lifetime. The number of block erasures was used as a metric to estimate its lifetime benefit. Fig. 17 shows experimental results for both db_bench and YCSB. Thanks to the reduction of GC I/Os, ALSM exhibits the smallest erasure counts across all the benchmarks, except for read-intensive ones, extending the SSD lifetime by 35 percent, on average.

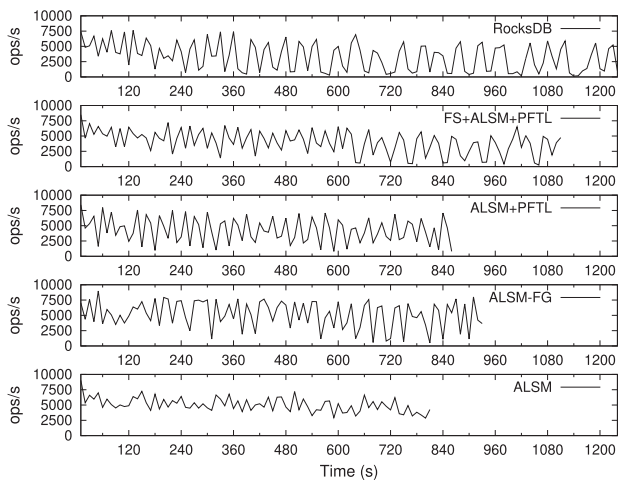


Fig. 16. Throughput of YCSB-A by type.

8 Conclusion

In this paper, we have proposed AMF, an Application-Managed Flash architecture. AMF is based on the new block I/O interface (AMF IO) exposing flash storage as append-only segments, while hiding unreliable NAND devices because their management requires vendor-specific details. Based on our new block I/O interface, we developed ALFS, a new file system, and ALSM, a new key-value store, along with AFTL, a new FTL for the storage device. Our evaluation shows that AMF outperformed EXT4 and RocksDB 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. Many DBMS engines manage storage devices in an LFS-like manner [16], [17], [20], so we expect that AMF can be easily adopted to them. Another future research topic is to enhance the AMF device such that it efficiently supports multiple types of applications (e.g., AMF-compatible and traditional applications) that share the same SSD.

Acknowledgments

An earlier version of this article was presented at the USENIX Conference on File and Storage Technologies, February 22-25, 2016 [44]. This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korea government (MSIT) (NRF-2017R1E1A1A01077410) and in part by the DGIST RD Program of the Ministry of Science and ICT (18-EE-01). This work at CSAIL, MIT was supported by Samsung GRO 2017–2018.

References

[1] S.-M. Jung, *et al.*, "Three dimensionally stacked NAND flash memory technology using stacking single crystal Si layers on ILD and TANOS structure for beyond 30nm node," in *Proc. Int. Electron Devices Meeting*, 2006, pp. 1–4.

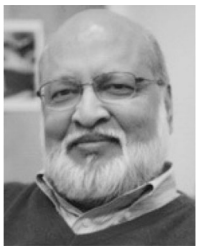
[2] W. Cheong, *et al.*, "A flash memory controller for 15s ultra-low-latency SSD using high-speed 3D NAND flash with 3s read time," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2018, pp. 338–340.

[3] Hitachi, "Hitachi accelerated flash," 2015. [Online]. available: https://community.hitachivantara.com/s/article/hitachi-accelerated-flash-an-innovative-approach-to-solid-state-storage

[4] Samsung, "Samsung SSD 840 EVO data sheet, rev. 1.1," 2013. [Online]. Available: https://static6.arrow.com/aropdfconversion/49828fa760cf76724e03f5161b5db14bc1a51b1a/samsung_ssd_840evo_datasheet_rev11.pdf
[5] Phison, "PS3110 controller," 2014. [Online]. available: https://www.phison.com/en/solutions/embedded/sata-pata/52-sata-pata/55-ps3110-s10
[6] K. Ha and J. Kim, "A program context-aware data separation technique for reducing garbage collection overhead in NAND flash memory," in *Proc. Int. Workshop Storage Netw. Archit. Parallel I/O*, 2011.
[7] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. USENIX Workshop Hot Topics Storage File Syst.*, 2014.
[8] S. S. Hahn, J. Jeong, and J. Kim, "To collect or not to collect: Just-in-time garbage collection for high-performance SSDs with long lifetimes," in *Proc. USENIX Symp. Operating Syst. Des. Implemenation*, 2014, pp. 1–6.
[9] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. Int. Conf. Architect. Support Program. Lang.Operating Syst.*, 2009, pp. 229–240.
[10] D. Park, B. Debnath, and D. Du, "CFTL: A convertible flash translation layer adaptive to data access patterns," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2010, pp. 365–366.
[11] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen, "S-FTL: An efficient address translation for flash memory by exploiting spatial locality," in *Proc. IEEE Symp. Mass Storage Syst. Technol.*, 2011, pp. 1–12.
[12] M. Bjørling, J. González, and P. Bonnet, "LightNVM: The linux open-channel SSD subsystem," in *Proc. USENIX Conf. File Storage Technol.*, 2017, pp. 359–374.
[13] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, pp. 1–15, 1991.
[14] D. Hitz, J. Lau, and M. A. Malcolm, "File system design for an NFS file server appliance," in *Proc. Winter USENIX Conf.*, 1994, pp. 235–246.
[15] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. USENIX Conf. File Storage Technol.*, 2015, pp. 273–286.
[16] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: A scalable log-structured database system in the cloud," in *Proc. VLDB Endowment*, vol. 5, 2012, pp. 1004–1015.
[17] "RethinkDB," 2015. [Online]. Available: http://rethinkdb.com
[18] P. O'Neil, E. Cheng, D. Gawlick, andE. O'Neil,"The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[19] "RocksDB: A persistent key-value store for fast storage environments," 2015. [Online]. Available: http://rocksdb.org
[20] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008, Art. no. 4.
[21] T. Feldman and G. Gibson, "Shingled magnetic recording areal density increase requires new data management," *USENIX Issue*, vol. 38, no. 3, pp. 22–30, 2013.
[22] S.-W. Jun, *et al.*, "BlueDBM: An appliance for big data analytics," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2015, pp. 1–13.
[23] M. Liu, S.-W. Jun, S. Lee, J. Hicks, and Arvind, "minFlash: A minimalistic clustered flash array," in *Proc. Des. Autom. Test Eur. Conf.*, 2016, pp. 1255–1260.
[24] Y. Kang, J. Yang, and E. L. Miller, "Efficient storage management for object-based flash memory," in *Proc. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2010, pp. 407–409.
[25] Y.-S. Lee, S.-H. Kim, J.-S. Kim, J. Lee, C. Park, and S. Maeng, "OSSD: A case for object-based solid state drives," in *Proc. Int. Symp. Mass Storage Syst. Technol.*, 2013, pp. 1–13.
[26] D. Woodhouse, "JFFS2: The journalling flash file system, version 2," 2001. [Online]. available: https://nnc3.com/mags/LM10/issue/17/DavidWoodhouse_JFFS2.pdf
[27] C. Manning, "YAFFS: Yet another flash file system," 2002. [Online]. available: https://yaffs.net/
[28] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann, "NoFTL: Database systems on FTL-less flash storage," *Proc. VLDB Endowment*, vol. 6, pp. 1278–1281, 2013.
[29] D. Woodhouse, "Memory technology device (MTD) subsystem for Linux," 2005. [Online]. available: http://www.linux-mtd.infradead.org/
[30] A. Hunter, "A brief introduction to the design of UBIFS," 2008. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf
[31] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of NAND flash memory," in *Proc. USENIX Conf. File Storage Technol.*, 2012, pp. 17–24.
[32] Y. Pan, G. Dong, and T. Zhang, "Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 7, pp. 1350–1354, Jul. 2013.
[33] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "DFS: A file system for virtualized flash storage," in *Proc. USENIX Conf. File Storage Technol.*, 2010, pp. 1–25.
[34] M. Jung *et al.*, "Exploring the future of out-of-core computing with compute-local non-volatile memory," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 75:1–75:11.
[35] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *Proc. Workshop Interact. NVM/Flash Operating Syst. Workloads*, 2014.
[36] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-defined flash for web-scale internet storage systems," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2014, pp. 471–484.
[37] A. Ban, "Flash file system," US Patent 5,404,485, 1995.
[38] C. Chung, J. Koo, J. Im, Arvind, and S. Lee, "Lightstore: Software-defined network-attached key-value drives," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 939–953.
[39] L.-P. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *Proc. Symp. Appl. Comput.*, 2007, pp. 1126–1130.
[40] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE Trans. Consumer Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.
[41] J. Axboe, "FIO benchmark," 2013. [Online]. Available: http://freecode.com/projects/fio
[42] J. Katcher, "PostMark: A new filesystem benchmark," NetApp, Sunnyvale, CA, Tech. Rep. TR3022, 1997.
[43] A. Kopytov, "SysBench: A system performance benchmark," 2004. [Online]. Available: http://sysbench.sourceforge.net
[44] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind, "Application-managed flash," in *Proc. USENIX Conf. File Storage Technol.*, 2016, pp. 339–353.
[45] "ARDB: A redis protocol compatible nosql," 2018. [Online]. Available: https://github.com/yinqiwen/ardb

Jinhyung Koo received the BS degree in computer science and engineering from Inha University, in 2017, and the MS degree in information and communication engineering from DGIST, in 2019. He is currently working toward the PhD degree in information and communication engineering at DGIST in South Korea. His current research interests include storage systems, operating systems, and system software.

Chanwoo Chung received the BS degree in electrical and computer engineering from Seoul National University, in 2014, and the MS degree in electrical engineering and computer science from Massachusetts Institute of Technology (MIT), in 2016. He is currently working toward the PhD degree in electrical engineering and computer science at MIT. His current research interests include embedded systems, hardware accelerators, and storage systems.



Arvind (Fellow, IEEE) received the BTechÂ degree in electrical engineering from IIT, Kanpur, and the PhD degree in computer science from the University of Minnesota. He is the Johnson professor and head of Computer ScienceÂ Faculty in the Schwarzman College of Computing at the Massachusetts Institute of Technology. His current research interests include rapid development of embedded systems and high-level hardware synthesis. He is a fellow of ACM, Chair of the Computer Science Section of the National Academy of Engineering, and a member of the American Academy of Arts and Sciences.

Sungjin Lee received the BE degree in electricalÂ engineering from Korea University, in 2005, and the MS and PhD degrees in computer science and engineering from Seoul National University, in 2007 and 2013, respectively. He is an assistant professor at DGIST in South Korea. Before joining DGIST, he was a postdoctoral associate with the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology, Cambridge, MA. His current research interests include storage systems, operating systems, and system software.