OrcFS: Orchestrated File System for Flash Storage

JINSOO YOO and JOONTAEK OH, Hanyang University
SEONGJIN LEE, Gyeongsang National University
YOUJIP WON, Hanyang University
JIN-YONG HA, JONGSUNG LEE, and JUNSEOK SHIM, Samsung Electronics

In this work, we develop the *Orchestrated File System (OrcFS)* for Flash storage. OrcFS vertically integrates the log-structured file system and the Flash-based storage device to eliminate the redundancies across the layers. A few modern file systems adopt sophisticated append-only data structures in an effort to optimize the behavior of the file system with respect to the append-only nature of the Flash memory. While the benefit of adopting an append-only data structure seems fairly promising, it makes the stack of software layers full of unnecessary redundancies, leaving substantial room for improvement. The redundancies include (i) redundant levels of indirection (address translation), (ii) duplicate efforts to reclaim the invalid blocks (i.e., segment cleaning in the file system and garbage collection in the storage device), and (iii) excessive over-provisioning (i.e., separate over-provisioning areas in each layer). OrcFS eliminates these redundancies via distributing the address translation, segment cleaning (or garbage collection), bad block management, and wear-leveling across the layers. Existing solutions suffer from high segment cleaning overhead and cause significant write amplification due to mismatch between the file system block size and the Flash page size. To optimize the I/O stack while avoiding these problems, OrcFS adopts three key technical elements.

First, OrcFS uses *disaggregate mapping*, whereby it partitions the Flash storage into two areas, managed by a file system and Flash storage, respectively, with different granularity. In OrcFS, the metadata area and data area are maintained by 4Kbyte page granularity and 256Mbyte superblock granularity. The *superblock-based storage management* aligns the file system section size, which is a unit of segment cleaning, with the superblock size of the underlying Flash storage. It can fully exploit the internal parallelism of the underlying Flash storage, exploiting the sequential workload characteristics of the log-structured file system. Second, OrcFS adopts *quasi-preemptive segment cleaning* to prohibit the foreground I/O operation from being interfered with by segment cleaning. The latency to reclaim the free space can be prohibitive in OrcFS due to its large file system section size, 256Mbyte. OrcFS effectively addresses this issue via adopting a polling-based segment cleaning scheme. Third, the OrcFS introduces *block patching* to avoid unnecessary write amplification in the partial page program. OrcFS is the enhancement of the F2FS file system. We develop a prototype OrcFS based on F2FS and server class SSD with modified firmware (Samsung 843TN). OrcFS reduces the device mapping table requirement to 1/465 and 1/4 compared with the page mapping and the smallest mapping scheme known to the public, respectively. Via eliminating the redundancy in the segment cleaning and

This research was supported by Basic Research Lab Program through the NRF funded by the Ministry of Science ICT&Future Planning (No. 2017R1A4A1015498), the BK21 plus program through the NRF funded by the Ministry of Education of Korea, the ICT R&D program of MSIP/IITP (R7117-16-0232, Development of extreme I/O storage technology for 32Gbps data services), and the Ministry of Science ICT&Future Planning under the ITRC support program (IITP-2016-H8501-16-1006) supervised by the IITP.
Authors' addresses: J. Yoo, J. Oh, and Y. Won (corresponding author), Hanyang University; emails: {jedisty, na94jun, yjwon}@hanyang.ac.kr; S. Lee (corresponding author), Gyeongsang National University; email: insight@gnu.ac.kr; J.-Y. Ha, J. Lee, and J. Shim, Samsung Electronics; emails: {jy200.ha, js0007.lee, junseok.shim}@samsung.com.

https://doi.org/10.1145/3162614

garbage collection, the OrcFS reduces 1/3 of the write volume under heavy random write workload. OrcFS achieves 56% performance gain against EXT4 in *varmail* workload.

Categories and Subject Descriptors: D.4.2 [**OPERATING SYSTEMS**]: Storage Management

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Log-structured File System, Flash memories, Garbage Collection

ACM Reference format:
Jinsoo Yoo, Joontaek Oh, Seongjin Lee, Youjip Won, Jin-Yong Ha, Jongsung Lee, and Junseok Shim. 2018. OrcFS: Orchestrated File System for Flash Storage. *ACM Trans. Storage* 14, 2, Article 17 (April 2018), 26 pages.
https://doi.org/10.1145/3162614

1 INTRODUCTION

Flash-based storage devices have emerged as a mainstream storage medium not only in the mobile device but also in the server and cloud computing areas (Berry 2015). The proportion of Flash storage sales in the entire storage market increased from 6.5% in 2010 to 31.4% in 2014, and the market share grew beyond 50% by the end of 2016 (ChosunBiz 2016), with most of the growth coming the server sector. The $ per byte of SSD against HDD has also dropped from 6× in 2012 to 2.8× in 2016 (Mearian 2016). The rapid proliferation of Flash storage is greatly indebted to the introduction of new technologies such as 3D-NAND (Samsung 2015) and finer process technology (Davis et al. 2013).

SSDs are getting larger and faster as SSD vendors aggressively adopt internal parallelism, multi-core controller, and massive-size DRAM. We examined the DRAM sizes of 184 SSD models manufactured between 2011 and 2015. Figure 1 illustrates the result. The size of DRAM has increased considerably over the years. About 60% of SSDs manufactured in that period had 512Mbyte or 1Gbyte of DRAM. Mapping table size is roughly equivalent to 0.1% of the Flash storage capacity. Our survey reflects this trend. Given this trend, it is not difficult to foresee that 4Tbyte SSDs will be loaded with 4Gbyte of DRAM to harbor various information, such as mapping table, and so on.

The application, as well as the file system, optimizes itself to better exploit the physical nature of the underlying Flash storage. The application adopts write-optimized and append-only search structure, such as LSM-tree (O' Neil et al. 1996), to improve the insert/update performance (Chang et al. 2008; Lakshman and Malik 2009; Banker 2011; RocksDB 2014). The file systems adopt append-only log structure in managing file system partition (Lee et al. 2015; Konishi et al. 2006) to align their behavior with the physical nature of the Flash storage. The efficacy of these application level and file system level efforts is limited, because it does not control the behavior of the storage firmware, the Flash Translation Layer (FTL) (Lee et al. 2008; Gupta et al. 2009; Kang et al. 2006). The application, the file system, and the FTL each maintains its own mapping table, performs wear-leveling and garbage collection, and reserves a fraction of space for the over-provisioning purpose. The redundancies across the layers bring significant waste of storage space and duplicate efforts associated with consolidating the valid blocks. The redundancies negatively affect the storage performance as well as the lifespan of the storage device. Despite the sophisticated efforts to harness the SSD with more horse-power, the host is able to see only less than 50% of the raw bandwidth (Ouyang et al. 2014).

Recently, several studies have proposed methods to remove the redundancies between the host I/O stack and the storage firmware (Lee et al. 2016; Zhang et al. 2016). These studies utilize a log-structured file system to reduce the size of the mapping table in the storage and to eliminate the duplicated effort to reclaim invalid blocks. However, they have limitations such as excessive

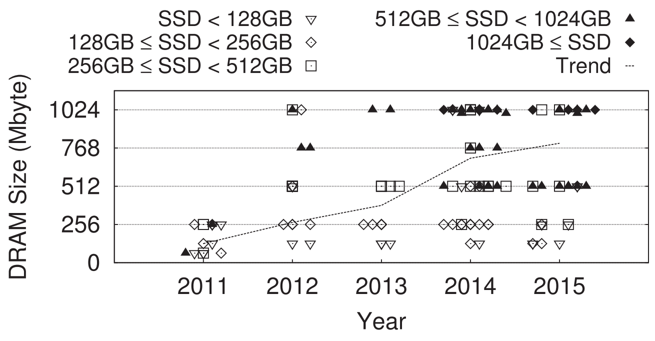


Fig. 1. DRAM size trend in SSDs (from 2011 to 2015).

segment cleaning overhead due to a large segment cleaning unit and the additional write amplification caused by the discrepancy between the file system block size and the Flash page size in a block mapping FTL (Lee et al. 2007). These limitations make it difficult to apply the techniques to a real desktop or server system (Section 4).

To address this limitation, in this work, we develop the *Orchestrated File System (OrcFS)*. OrcFS vertically integrates the file system and the storage firmware, eliminating the redundancies between the two. OrcFS shares much of its philosophy with the preceding works that integrate and optimize storage stack (Marmol et al. 2015; Ouyang et al. 2014; Weiss et al. 2015; Zhang et al. 2015, 2016; Lee et al. 2016). However, its actual design uniquely distinguishes itself from the preceding works in three aspects. First, OrcFS eliminates the redundancy in address mapping by managing the Flash storage directly. The mapping scheme fully exploits the internal parallelism of the underlying Flash storage. Second, it avoids the excessive segment cleaning latency due to a superblock-based cleaning unit via employing *quasi-preemptive segment cleaning (QPSC)*. Third, by employing *block patching*, it eliminates the partial Flash page writes, which increase the write amplification of the storage. The contributions of OrcFS are as follows.

- Disaggregate Mapping Scheme: OrcFS uses a newly developed *disaggregate mapping* scheme that manages page-level mapping and superblock-level mapping in the storage and file system, respectively. Using superblock-level mapping, OrcFS can fully exploit the internal parallelism of the Flash storage. As the storage manages page-level mapping only for the file system metadata, it reduces the mapping table size by 1/465 compared to the storage using page-level mapping (Section 3.2).
- Quasi-Preemptive Segment Cleaning: By applying QPSC, OrcFS resolves the problem that the processing of the foreground I/O operation takes 6–9s due to segment cleaning operation of the file system. In a random write workload, the maximum latency of the `write()` system call in F2FS is 9s, while OrcFS with QPSC reduces the maximum latency to 1s (Section 5.6).
- Block Patching: Recent works (Lee et al. 2016; Zhang et al. 2016) adopt block mapping to reduce the size of mapping table. However, the block mapping can increase write amplification due to partial Flash page write (Lee et al. 2007) when the file system block size and the Flash page size do not coincide. OrcFS employs *block patching* to address this issue (Section 3.5).

The rest of the article is organized as follows. Section 2 describes the stacked log system and compound segment cleaning problem. Section 3 describes the design and implementation of OrcFS.

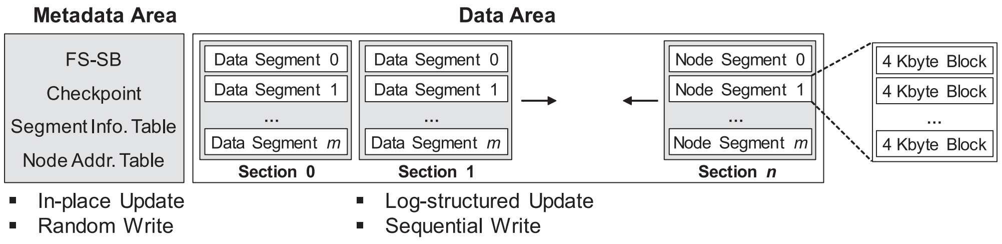


Fig. 2. F2FS partition layout (FS-SB: File system Superblock).

Section 4 compares *OrcFS* with other systems. Section 5 shows the performance of OrcFS through various experiments and workloads. Section 6 describes the related work and Section 7 concludes the article.

2 BACKGROUND

2.1 Segment Cleaning in Log-Structured File System

The log-structured file system is a write-optimized file system (Rosenblum and Ousterhout 1992). The file system minimizes the seek overhead by clustering and placing the data blocks and the updated metadata blocks in proximity. The file system writes the blocks in an append-only manner, and all obsolete blocks are marked as invalid. The file system maintains an in-memory buffer to locate the up-to-date version of individual file system blocks. To minimize the disk traffic, the file system buffers the updates and then flushes them to the disk as a single unit, a *segment* (e.g., 2Mbyte), either when the buffer is full or when `fsync()` is called. The invalid file system blocks need to be reclaimed to accommodate the newly incoming writes. The process is called *segment cleaning*. The segment cleaning overhead is one of the main obstacles to the wider adoption of its technology, since it makes the file system suffer from unexpected long delays (Seltzer et al. 1995).

Since the log-structured file system was first released to the public (Rosenblum and Ousterhout 1992), a fair amount of log-structured file systems have been proposed (Engel and Mertens 2005; Konishi et al. 2006; Lee et al. 2015; Czezatke and Ertl 2000; StarWind 2014). While the append-only nature of the log-structured file system ideally fits with the physical characteristics of the Flash storage, most log-structured file systems fail to address the critical issues inherent in log-structured file systems: segment cleaning overhead, `fsync()` overhead, and inability to handle direct IO. Due to these reasons, few works have gone beyond academic exercises and are used in very specialized environments (Konishi et al. 2006). Furthermore, Flash-optimized file systems still operate on top of block device abstraction provided by the FTL, and thus, it cannot fully exploit the raw performance of the SSD (Ouyang et al. 2014).

The recently proposed F2FS (Lee et al. 2015) successfully addresses most of the existing issues in log-structured file systems; it properly exploits the append-only nature of the device and shows good random write performance of Flash storage. Instead of clustering the data and the associated metadata together, F2FS maintains the file system metadata and the data in separate regions (Figure 2). F2FS updates the metadata region and the data region in an in-place manner and append-only manner, respectively. The metadata area consists of the file system super block (FS-SB), checkpoint, segment information table (SIT), and node address table (NAT). The SIT manages the block bitmap of each segment in the data area. The NAT manages the block address corresponding to each node ID. To distinguish the file system superblock with the Flash storage superblock, which is a set of the Flash blocks (Park et al. 2009), we refer to the file system superblock as FS-SB. In log-structured file systems, it is important to cluster the data blocks with similar update frequency.

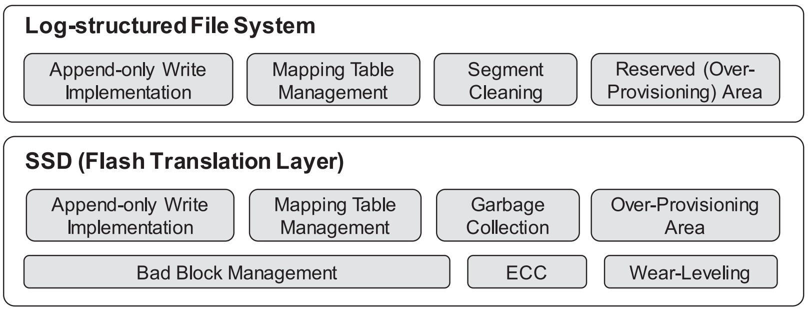

Log-structured File System
Append-only Write Implementation
Mapping Table Management
Segment Cleaning
Reserved (Over-Provisioning) Area
SSD (Flash Translation Layer)
Append-only Write Implementation
Mapping Table Management
Garbage Collection
Over-Provisioning Area
Bad Block Management
ECC
Wear-Leveling

Fig. 3. Log-structured file system over Flash storage.

This is to minimize the segment cleaning overhead. F2FS defines the two types of segment: data segment and node segment. It further partitions each type of segment into three subject to the update frequency of the blocks in it: hot, warm, and cold. F2FS maintains six active segments in-memory. F2FS flushes an active segment when it is full.

F2FS provides a notion of *segment* and *section*. F2FS partitions a file system partition into fixed-size *segments*. A *segment* is an array of file system blocks (4Kbyte). The default *segment* size is 2Mbyte. Usually, a *segment* size is configured to match the Flash block size. A *section* is an array of the fixed number of *segments*. In F2FS, the *section* is the unit of segment cleaning.

2.2 Garbage Collection in Flash Storage

Garbage collection is a process of reclaiming invalid pages in the Flash storage (Agrawal et al. 2008). Garbage collection not only interferes with the I/O requests from the host but also shortens the lifespan of the Flash storage. A fair amount of garbage collection algorithms have been proposed: greedy (Kawaguchi et al. 1995), EF-greedy (Kwon et al. 2007), cost benefit (Rosenblum and Ousterhout 1992), and so on. Various techniques have been proposed to hide the garbage collection overhead from the host. They include background garbage collection (Smith 2011), preemptive garbage collection (Lee et al. 2011), and so on. The SSD controller allocates a separate hardware thread for garbage collection so that garbage collection does not interfere with I/O requests from the host.

The host helps FTL to minimize the garbage collection overhead via the TRIM command (Shu and Obr 2007) and Multi-stream ID (Kang et al. 2014). Although numerous algorithms have been proposed, since the inception of Flash storage, garbage collection is still problematic (Zheng et al. 2015; Yang et al. 2015).

2.3 Redundancies across the Layers

Log-structured file systems effectively transform the random writes to the sequential writes. Sequential writes benefit not only the HDD but also the Flash storage. Sequential writes greatly simplify the address mapping and the garbage collection overhead of the underlying storage. Recently, a number of key-value stores exploit the append-only update mechanism to optimize its behavior towards write operations (Lim et al. 2011; Ghemawat and Dean 2014). The log-structured file system for SSD entails a number of redundancies when it is used for Flash storage. There are three major redundancies: (i) mapping information, (ii) segment cleaning, and (iii) over-provisioning area.

Figure 3 illustrates a log-structured file system over an SSD, which has the three redundancies. First, both the file system and the Flash storage introduce the level of indirections. The file system and the Flash storage each has to manage its own metadata for address mapping. As the capacity of SSD increases, the memory overhead of legacy page mapping, which is the most widely used mapping scheme in commercial Flash products, becomes more significant. Second, each layer performs segment cleaning on its own. We define compound segment cleaning as the phenomenon where the storage-level log system performs the garbage collection on the data blocks that are already segment-cleaned by a log-structured file system. The compound segment cleaning not only degrades the IO performance but also shortens the lifespan of the Flash storage (Yang et al. 2014; Lee et al. 2016; Zhang et al. 2016). Third, each of the log layers needs to reserve a certain fraction of its space as the over-provisioning area. The log-structured file system (or SSD) sets aside a certain fraction of its file system space (or storage space) to host the extra write operations caused by the segment cleaning. Over-provisioning is an integral part of Flash storage, but at the same time, it adds significant overhead to the expensive Flash storage. The total amount of the over-provisioning areas for individual log layers can be very significant, e.g., 20% (Ouyang et al. 2014).

Recently, several studies (Lee et al. 2016; Zhang et al. 2016) have proposed mitigating the mapping table overhead of the Flash storage and eliminating the various redundancies in the I/O stack by using an application or a file system that uses a log-structured update. By using a log-structured file system, these studies enable the storage to use block mapping FTL. Thus, the size of the mapping table is significantly reduced compared to the page mapping. They also eliminate the duplicated overhead for valid block consolidation by integrating the garbage collection of the storage and the segment cleaning of the file system layer. However, these works fail to address the increased tail-latency of the I/O request. These works use the larger segment cleaning unit to fully exploit the internal parallelism (e.g., multiple channels and ways of the Flash storage). The larger segment cleaning unit inevitably entails increase in the segment cleaning latency and may severely interfere with the I/O request from the host. In addition, they do not cover the additional write amplification caused by partial Flash page writes in a block mapping scheme (Lee et al. 2007). The problem may occur when the file system block size and Flash page size are different. Considering the trend of using Flash pages larger than 8 Kbyte in the latest Flash storage (Micron 2016; Samsung 2015), the problem makes it difficult to apply the above techniques to real storage systems. The limitations of these studies and the comparison with our work are discussed in more detail in Section 4.

3 DESIGN: ORCHESTRATED FILE SYSTEM

3.1 Concept

The *OrcFS* aims at reducing the overhead of the segment cleaning in a large segment unit and at resolving the write amplification caused by partial Flash page write while eliminating all data structural and operational redundancies across the file system and the Flash storage. In OrcFS, SSD exposes its physical blocks to the host and leaves most of its FTL functions (address translation and garbage collection) to the host file system. Since the file system is responsible for consolidating the valid storage blocks, the underlying Flash storage does not have to set aside a space for over-provisioning purposes.

Via eliminating all redundancies, OrcFS reduces the write amplification, saves space for over-provisioning, and reduces the memory requirement of the Flash storage device. As a result, it improves the performance as well as the lifespan of the storage device. Furthermore, via reducing the memory requirement and eliminating the need for over-provisioning, we can build less expensive Flash storage.

We developed the prototype of OrcFS using F2FS. The file system layout and the metadata structure of OrcFS are the same as those of F2FS (Figure 2). As in F2FS, OrcFS maintains the file system

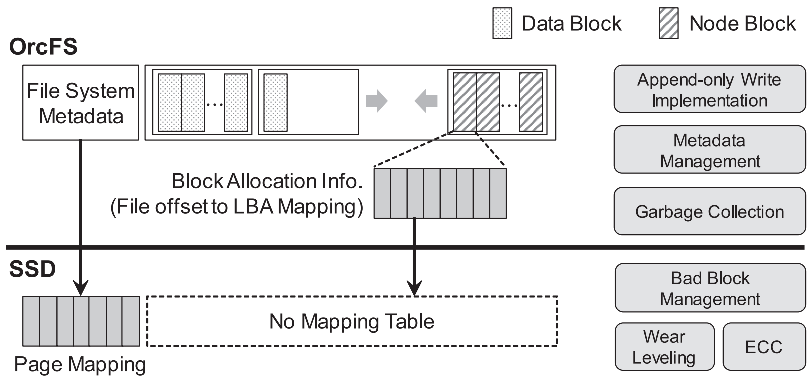


Fig. 4. OrcFS architecture.

metadata and the data in separate regions. The metadata region and the data region are updated in an in-place manner and append-only manner, respectively. The size of the metadata region is determined according to the file system partition size. The metadata region consists of the FS-SB, checkpoint, SIT, and NAT. The size of SIT and NAT are determined by the total number of segments and the total number of blocks in the file system partition, respectively. Note that the per-file metadata (node) is written in the data area. In OrcFS, we set a section size the same as the superblock size in the Flash storage. A superblock is a set of Flash blocks located at the same offset in each Flash memory bank (Park et al. 2009). For example, if the storage's superblock size is 256Mbyte, then OrcFS has 256Mbyte section, and each section has 128 segments. OrcFS uses a section as the unit of segment cleaning.

Figure 4 illustrates the architecture of the OrcFS. In OrcFS, the Flash storage is partitioned into two regions: page granularity partition and superblock granularity partition. The size of these partitions is aligned with the multiples of a superblock. These two partitions are statically bound to the metadata region and the data region of the file system, respectively. For the metadata region, OrcFS delegates the address translation and its management (e.g., garbage collection) to the Flash storage. The Flash storage maintains its own mapping table for the metadata region. In our implementation, SSD uses page mapping to manage the metadata region.

For the data region, OrcFS is responsible for managing the physical Flash blocks. OrcFS stores the block allocation information for a file in the associated node blocks. A node block translates a file offset to the logical block address (LBA) in the file system partition. Like F2FS, a node block contains 923 direct pointers, two single-indirect pointers, two double-indirect pointers, and one triple-indirect pointer. As each direct node block and indirect node block has 1,018 block addresses, an inode can point to as many as 1,057,053,429 block addresses.[1] Thus, the maximum file size supported by OrcFS is 3.94Tbyte. In other words, OrcFS requires about 4Gbyte to store the block allocation information[2] to represent a 4Tbyte file, which is the same as F2FS. Note that OrcFS does not need to load all the node blocks in the memory at once. This is because OrcFS manages node blocks in the same way as the data blocks. When the file system reads or updates a node block, the block is cached to the host page cache. The node blocks are periodically flushed to the storage by kworker thread or flush commands such as `fsync()`. When the LBA for an incoming I/O request

[1] $923 + 2 \times 1,018 + 2 \times 1,018^2 + 1 \times 1,018^3 = 1,057,053,429$.
[2] The maximum number of node blocks for a file: $1 + 2 + 2 \times (1 + 1018) + (1 + 1018 + 1018^2) = 1,039,384$ blocks. The total size of the node blocks: 1,039,384 node blocks × 4Kbyte = 4Gbyte.

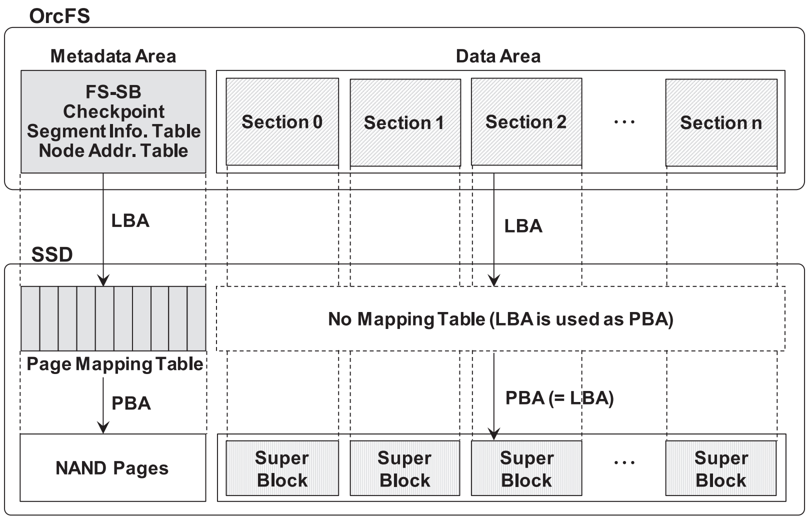


Fig. 5. Disaggregate mapping (FS-SB: File system Superblock).

is for the data region, the Flash storage directs the requests to the storage firmware without any interpretation.

In OrcFS, the storage has responsibility for bad block management, wear-leveling, and ECC. We leave these features at the storage level, since the storage can utilize Flash-specific information such as bit error rates and P/E cycles. The bad block management module in the storage seamlessly redirects I/O requests for a bad Flash block to a spare Flash block. The file system assumes that all Flash blocks exposed to the host are not bad blocks. Section 3.4 addresses the bad block management in detail.

3.2 Disaggregate Mapping

The crux of OrcFS is *disaggregate mapping*. OrcFS manages the two file system regions (metadata region and data region) differently, both vertically and horizontally. The metadata region and data region are managed by the storage device and the host, respectively. OrcFS applies different address mapping granularities to each file system partition. It applies page mapping to the metadata region. The data region is managed in the superblock granularity by the host file system, OrcFS. The unit of allocation is a superblock, and the individual file system sections are mapped to each superblock in the Flash storage. The host file system performs segment cleaning in the unit of a section. Figure 5 schematically illustrates the disaggregate mapping. OrcFS writes file system metadata to the metadata region in an in-place manner and delegates the address mapping for this region to Flash storage. OrcFS applies append-only updates to the data region and is responsible for managing it directly.

The key advantage of OrcFS is superblock-unit-based storage management. The file blocks in a section are consolidated, after which the entire section becomes free. This superblock-unit-based allocation enables OrcFS to fully exploit the internal parallelism in the underlying Flash storage, leveraging the append-only sequential write characteristics of the log-structured file system.

To mitigate the segment cleaning overhead in superblock-based management, it is important to cluster Flash pages with similar hotness in the same superblock. F2FS categorizes a block in the data region into one of the six groups subject to whether it is a data block or node block and subject to its hotness group (hot, warm, and cold) (Lee et al. 2015). OrcFS uses the same mechanism in categorizing the file blocks in F2FS. The blocks in the same group are clustered together and are placed at the same section. The overhead of section cleaning will be discussed in Section 3.3.

As the OrcFS generates small random write requests for the metadata region, we decided to manage the region with page mapping FTL in the Flash storage. If OrcFS manages the page mapping in the file system layer, then the host must know the internal structure of the Flash storage such as the number of channels/ways to exploit I/O parallelism of the storage. This makes the file system implementation complicated. Thus, we leave the page mapping in the storage.

The SSD firmware treats the incoming request differently subject to where the request is designated. If the LBA for incoming request is for the metadata area, then the firmware directs them to page mapping FTL; if LBA is for the section granularity region, then the firmware recognizes the LBA as a physical block address (PBA). Mapping information for the metadata area and for the data area are harbored by the storage device and the host, respectively. For a 4Tbyte SSD, the legacy page mapping requires 4Gbyte of DRAM for the page mapping table. If we format a 4Tbyte SSD with OrcFS, then the metadata area of the file system occupies 9Gbyte. Since OrcFS manages the page mapping table only for the metadata area, it requires 9Mbyte at the storage device for the mapping table. Eliminating the redundancy in address translation, OrcFS removes the compound segment cleaning (Yang et al. 2014). The Flash storage is relieved of the burden of allocating surplus storage space (invisible to the host) for consolidating the valid blocks.

The size of the over-provisioning area is set to 7% of an SSD's capacity; however, more than 20% of an SSD is recommended for a data center or heavy workload applications (Kingston Technology 2013; Samsung Electronics Co. 2014). Disaggregate mapping enables Flash storage to greatly reduce the size of over-provisioning area. This is because the Flash storage in OrcFS performs garbage collection only for the Flash blocks mapped to the file system metadata region. For example, if we format a 256Gbyte SSD with OrcFS, the size of the metadata region is 768Mbyte, which occupies 3 % of the SSD capacity. Thus, the size of the over-provisioning area is 154Mbyte if we set the over-provisioning region to 20% of the metadata region. It occupies only 0.06% of the 256Gbyte SSD.

Disaggregate mapping differs from hybrid FTLs (Lee et al. 2008, 2007; Kwon et al. 2010). The gist of the hybrid FTLs is log blocks managed by page-level mapping, while most of the Flash blocks (or data blocks) are managed at the block-level. Whereas OrcFS redirects write requests to each region according to the LBAs, hybrid FTLs write the host data temporarily in log blocks. When the log is full, the data in the log blocks are migrated to the corresponding data blocks, which is referred to as a merge operation. As each merge operation involves valid page copies and Flash block erases, it results in the same overhead as garbage collection (Kang et al. 2006). Additionally, merge operations with a segment cleaning of the log-structured file systems create the "compound segment cleaning" problem. While the size of each region is fixed in the disaggregate mapping, the hybrid FTLs dynamically change the portion of log blocks according to the workload. The performance of these dynamic approaches is very sensitive to the appropriate categorization of the incoming write request, and misclassification can degrade the SSD performance by as much as 22% (Yoo et al. 2013).

3.3 Quasi-Preemptive Segment Cleaning

In segment cleaning, the segment cleaning module selects a victim section and copies the valid file system blocks in the victim section to the destination section. It cleans each segment in a section one at a time. After it cleans all segments in the section, the segment cleaning module marks the victim section as free. Then, the segment cleaning module sends the TRIM command (Shu and Obr 2007) with the list of cleaned block numbers as the parameter. The Flash storage invalidates and erases the associated Flash blocks later in the time line.

The latency of segment cleaning can be very large. Consider an eight-channel four-way SSD with a 2Mbyte block size. The section size corresponds to 64Mbyte. Assume that half of the file blocks in a section are valid. Then, we need to clean two sections to create a new free section. Copying 64Mbyte of the file system blocks entails significant latency. This can block applications for more than a few seconds, and the consequence can be disastrous. The existing log-structured file systems are not designed for a segment cleaning unit this large. While details may vary, the segment cleaning unit establishes an exclusive lock on the log until it creates a new segment. The subsequent operation on the respective partition is temporarily blocked. The file system stops the host write request processing until the segment cleaning operation is completed and free space is procured.

OrcFS triggers a segment cleaning if the free space in the file system becomes less than 5% of the entire file system partition size (foreground segment cleaning) or if the file system becomes idle (background segment cleaning). At the beginning of a segment cleaning, the segment cleaning module acquires `gc_mutex` lock. It blocks the `kworker` threads from performing write operations till the lock is cleared. Then, it selects a victim. Foreground segment cleaning uses the greedy (Kawaguchi et al. 1995) scheme and background segment cleaning uses cost-benefit (Rosenblum and Ousterhout 1992) scheme in victim selection. After selecting a victim section, the valid blocks in the victim section are read into the page cache. It is worth noting that valid blocks read from the victim section are categorized as *cold* blocks (Lee et al. 2015). They are written to a *cold* segment. In foreground segment cleaning, the valid blocks are immediately written to the storage, creating the new clean segment. For background segment cleaning, the file system continues cleaning the next victim segment. The segment cleaning module continues to clean the sections until the number of free sections increases by one. Once it acquires one new free section, the segment cleaning module flushes the dirty node blocks to the storage and flushes the file system metadata. Finally, it unlocks the `gc_mutex`.

The buffered `write()` system call can be interfered with by the file system segment cleaning. In servicing `write()`, the F2FS checks if there is sufficient free space (5% free space by default in the storage). If it is sufficient, then it allocates a page cache entry for a given `write()`. If free space is not sufficient, then the file system triggers segment cleaning and the application that has issued a `write()` is blocked until the segment cleaning ends. Due to this mechanism, the buffered `write()` can be delayed. The delay can be prohibitive, especially when the segment cleaning unit is large and when several segments (or sections in OrcFS) need to be cleaned due to high utilization in the victim segment. The `read()` request is not blocked by the segment cleaning.

There exist a fair number of preemptive garbage collection algorithms (Yan et al. 2017; Lee et al. 2011; Wu and He 2012). All these works are on the device-level garbage collection. A pre-emptive approach in the file system level segment cleaning poses a new challenge in that the kernel needs to save the state of the segment cleaning when the higher priority jobs arrive. Due to the complexity of storing the kernel state in realizing the fully preemptive segment cleaning, we propose QPSC, which enables OrcFS to implement segment cleaning with a large segment cleaning unit. Non-blocking writes (Campello et al. 2015) and non-blocking garbage collections (Yan et al. 2017; Lee et al. 2011; Wu and He 2012) are similar to QPSC in that they are intended to reduce the latency of host write requests. However, they aim at different objectives from QPSC. Non-block write (Campello et al. 2015) eliminates the page cache miss penalty that occurs in the write system call. Tiny-tail Flash (Yan et al. 2017) limits the GC domain within a channel or a die so that the controller services the I/O requests in other channels.

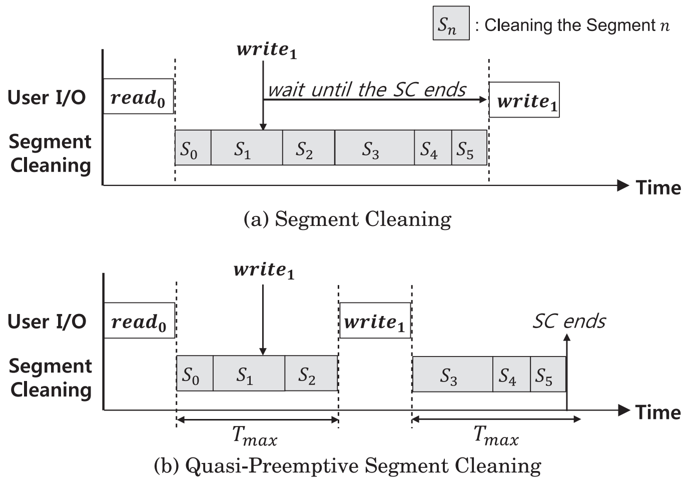
The pseudocode of QPSC is shown in Pseudocode 1. We define minimum polling interval, T_{max}. When the segment cleaning starts in OrcFS, it sets the timer to T_{max} (Default 100ms). After cleaning each segment, the segment cleaning module examines if the timer has expired. It continues

PSEUDOCODE 1: Segment Cleaning in OrcFS

```
function OrcFS_SegmentCleaning($T_{max}$)
    lock ($gc\_mutex$)
    $T_{start} \leftarrow current\_time()$
    if not $SC\_context.has\_victim$ then
        // Get the First Segment Number of the Victim Section
        $sec\_no \leftarrow select\_victim\_section()$
        $SC\_context.sec\_no \leftarrow sec\_no$
        $SC\_context.current\_seg \leftarrow 0$
    else
        $sec\_no \leftarrow$ SC_context.sec_no
    end if
    // Do Segment Cleaning during $T_{max}$
    // and check b_io for Preemption
    for offset←SC_context.current_seg to $N_{SegsPerSec}$ do
        $segment\_cleaning(sec\_no, offset)$
        if $T_{max} \leq current\_time() - T_{start}$ then
            if b_io is not empty then
                $SC\_context.has\_victim \leftarrow true$
                $SC\_context.current\_seg \leftarrow offset + 1$
                unlock ($gc\_mutex$)
                return // Preemption
            else
                $T_{start} \leftarrow current\_time()$
            end if
        end if
    end for
    if this is foreground segment cleaning then
        $checkpoint()$
    end if
    $SC\_context.has\_vicim \leftarrow false$
    unlock ($gc\_mutex$)
    return
end function
```

cleaning the segment if the timer has not expired. If the timer expires, then the segment cleaning module searches for any outstanding `write()` requests. If there are pending requests, then the segment cleaning module temporarily releases the lock, `gc_mutex`, and services the pending requests. After servicing all the pending requests, the segment cleaning module resumes with the timer reset to T_{max}. If there is no pending request, then it resets the timer to T_{max} and keeps cleaning the segments. The VFS layer of Linux OS maintains a list of inodes for which there exist outstanding write requests: `b_io` field in the `bdi_writeback` structure. The segment cleaning module examines this list to determine if there are any pending requests.

Figure 6 illustrates how the `write()` request is delayed when it is interfered with by the segment cleaning. In Figure 6(a), the `write()` is delayed until all the segments, $S_0, S_1, \ldots, S_5$, in a section are reclaimed. It should be noted that the `write()` request may have to wait until multiple sections are reclaimed if the utilization of each section is high. In Figure 6(b), the `write()` request is postponed until the polling interval expires. The `write()` request is serviced after reclaiming S_2.



S_n : Cleaning the Segment n
User I/O
Segment Cleaning
$read_0$
$write_1$
wait until the SC ends
S_0 S_1 S_2 S_3 S_4 S_5
Time
(a) Segment Cleaning
SC ends
T_{max}
(b) Quasi-Preemptive Segment Cleaning

Fig. 6. Segment cleaning.

To preserve the context of the segment cleaning, we introduce a new kernel object, `SC_context`. `SC_context` consists of the victim section number, the most recently cleaned segment number in the victim section, and `has_victim` flag for determining whether these values are legitimate. The segment cleaning thread starts the cleaning either from the first segment in the section or from the next segment to the one stored in the `SC_context` subject to the `has_victim` flag.

3.4 Bad Block Management and Wear-Leveling

In OrcFS, the storage device is responsible for bad block management. An SSD sets aside a set of Flash blocks as a spare area (Chow et al. 2007). The NAND blocks in this area are used to replace the bad blocks. This area should not be confused with the over-provisioning area. The NAND blocks are not visible to the host. An SSD provides a bad block management module with a bad block indirection table. A bad block indirection table consists of a pair of <*physical block number*, *spare block number*>. The *physical block number* denotes the physical block number of the bad block. The *spare block number* is the physical block number of the spare block to which the I/O request for the bad block is redirected. In segment cleaning, all the valid blocks are consolidated to the newly allocated segment. The replacement block allocated for the bad block is also migrated to the newly allocated segment. After the replacement block is copied to the newly allocated segment, the respective bad block mapping table is reset. Figure 7 shows how the bad block management layer in SSD interacts with segment cleaning. In accordance with the data region, bad block management for the metadata region also utilizes the bad block indirection table to redirect bad blocks that have file system metadata to spare blocks.

Typically, Flash storage has responsibility for wear-leveling. In static wear-leveling (Chang et al. 2007), FTL periodically migrates the cold data from the relatively young Flash block to the more worn Flash block so that the young Flash block can be used by the other data. Note that the wear-leveling generates additional Flash page writes that are unrelated to the Host I/O requests. OrcFS can perform wear-leveling without data migration in the storage layer by utilizing the file system-level hotness separation (Section 3.2). For each incoming write request, the data classified as cold can be allocated to the worn out superblocks for the static wear-leveling while the hot data are allocated to the fresh superblocks for the dynamic wear-leveling. By integrating the block allocation and wear-leveling in the file system, the storage can eliminate the additional

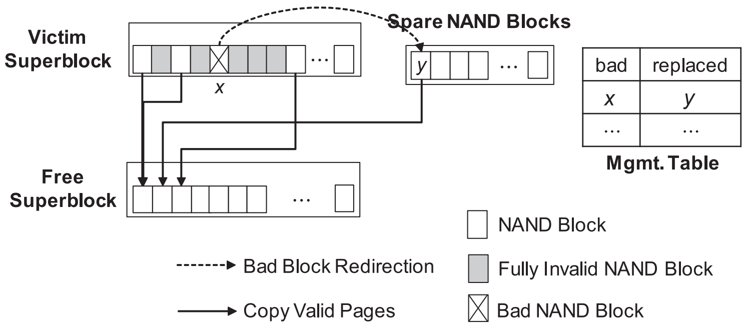


Fig. 7. Segment cleaning with bad block management.

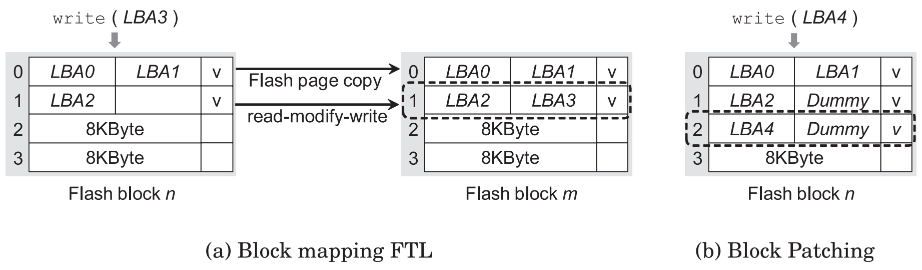


Fig. 8. Example of block patching.

overhead caused by the wear-leveling. The file system-level wear-leveling requires that the file system knows the erase count information of each Flash block of the Flash storage. The file system can get the information through the S.M.A.R.T (smartmontools 2010) interface modification. We leave the wear-leveling implementation for the OrcFS as a future work.

3.5 Block Patching

The file system block size is 4Kbyte in OrcFS. The page size of an SSD varies from 4 Kbyte to 16Kbyte, depending on manufacturer. In the legacy I/O stack, SSD firmware is responsible for handling this discrepancy through request merge, sub-page mapping, read-modify-write (Agrawal et al. 2008), and so on. The misalignment between the file system block size and the Flash page size leads to severe performance degradation of block mapping FTL (Lee et al. 2007). This is because block mapping FTL has the constraint that each logical page must be written to a fixed offset in the Flash block. Figure 8 shows an example, where Flash pages are 8Kbyte and each Flash block has 4 Flash pages. The Flash page 1 at block *n* is partially written with *LBA*2 (Figure 8(a)). When the file system needs to write *LBA*3 next to *LBA*2, all valid pages in block *n* need to be migrated into the new Flash block *m*, preserving the offsets within a block. The page 1 at block *m* becomes the host for *LBA*2 and *LBA*3. After copying all the valid pages, the FTL erases the Flash block *n*. Although the *replacement block* scheme (Kim et al. 2002) was proposed to mitigate the burden, it does not entirely eliminate the overhead caused by partial Flash page write in a block mapping FTL. This is because the *replacement block* scheme requires a merge operation between the original Flash block and the *replacement blocks* whenever the Flash block needs more than two *replacement blocks*. The *replacement block* scheme also has overhead for traversing the *replacement blocks* in performing the read operation.

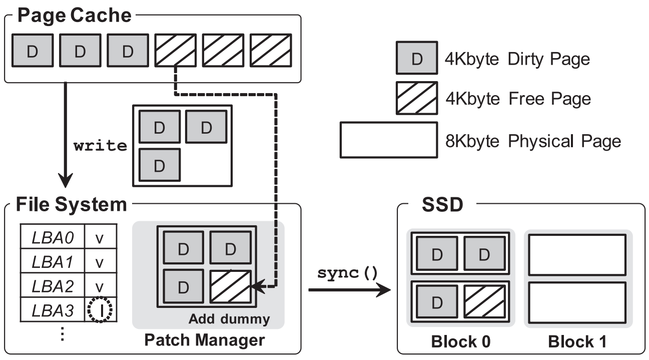


Fig. 9. Block patching.

We developed *block patching* to resolve the discrepancy between file system block size and Flash page size at the file system layer. Block patching aligns the size of write requests to the Flash page size and ensures that partial Flash page writes do not occur in the Flash storage. When the write request size is not aligned with the Flash page size, OrcFS pads the free page cache entry (4Kbyte) to the write request to make its size aligned with the Flash page size. The file system needs to allocate an additional file system block to accommodate the padded page cache entry. While the padded page cache entry is not reflected in the file size, it consumes an additional file system block. The block patching allocates an empty page from the page cache and concatenates it with the original write request (Figure 9). We mark the bit corresponding to the dummy page as invalid in the valid bitmap to let the segment cleaning module reclaim the page.

Figure 8(b) shows an example of block patching. When writing the *LBA*2 on Flash page 1, block patching adds a dummy page and performs a full Flash page write. Since the dummy page occupies *LBA*3, the next write request is to *LBA*4 and is written to Flash page 2 in the Flash block *n*. As shown in the example, block patching eliminates the unnecessary Flash page copies and additional write amplification.

4 COMPARISON

We compared the OrcFS with the six preceding works (Table 1) where the host directly manages the Flash storage to avoid the redundancies across the layers. In Table 1, "Device Mapping Table Size" denotes the size of *LBA* to *PBA* translation information. We exclude the size of the *file offset* to *LBA* translation information (*host mapping table size*) from the table. This is because the *host mapping table size* is dynamically changed according to the number of files and sizes. "Standard Interface" is marked as "○" if the technique does not require new I/O interfaces. Since the TRIM command (Kim et al. 2010) has been standardized as a part of the ATA standard interface (Shu and Obr 2007), we considered it as one of the standard interfaces. "Mapping Table (Device)" is marked as "◎" if the technique uses page-level mapping (Zhang et al. 2015; Marmol et al. 2015; Weiss et al. 2015).

ParaFS (Zhang et al. 2016) and application-managed flash (AMF) (Lee et al. 2016) adopt F2FS as the baseline file system and propose using block mapping FTL to exploit the sequential nature of the log-structured file system. ParaFS sets the file system segment size to the same as the Flash block size (2 Mbyte) and manages the Flash storage using a block granularity mapping table. On the other hand, AMF matches the file system segment size with the superblock size in the Flash storage (16 Mbyte) and maintains block-level mapping information in the storage. Since the host

Table 1. Comparison of OrcFS and Other Systems: ParaFS (Zhang et al. 2016), AMF (Lee et al. 2016), NVMKV (Marmol et al. 2015), ANViL (Weiss et al. 2015), FSDV (Zhang et al. 2015), and SDF (Ouyang et al. 2014) ("○": Medium, "◎": Large, "△": Small, "×": None, Mapping Table Size Is Based on 1TB SSD with 512KB Flash Block, 4KB Page)

	OrcFS	ParaFS 2016	AMF 2016	NVMKV 2015	ANViL 2015	FSDV 2015	SDF 2014
Mapping Table (Host)	◗	○	○	×	×	×	×
Mapping Table (Device)	◖	○	○	◎	◎	◎	○
Device Mapping Table Size	2.2MB	10MB	8MB	1GB	1GB	≤1GB	8MB
Garbage Collection	Host	Host	Host	Host	Device	Device	Host
Standard Interface	○	○	×	○	×	×	×
Overprovisioning	△	△	×	○	○	○	×
Application	General	General	General	KV Store	General	General	KV Store

file system and the storage device maintain a level of indirection, both of the approaches still have the mapping redundancy.

ParaFS and AMF do not properly address the issues caused by the discrepancy between the file system block size and the Flash page size. On the other hand, OrcFS successfully removes partial Flash page write by using the *block patching* scheme (Section 3.5) when the size of file system block and Flash page are different.

In AMF, the application may suffer from excessive tail latency in I/O operation if the unit of segment cleaning is superblock of the underlying Flash storage. When the superblock size is 256Mbyte, an application may observe as long as 9s latency in performing a 4Kbyte buffered write if the write request arrives when the file system performs segment cleaning (see Section 5.6). OrcFS solves the problem through the QPSC scheme described in Section 3.3.

NVMKV (Marmol et al. 2015), ANViL (Weiss et al. 2015), and FSDV (Zhang et al. 2015) use page mapping at the device-level. ParaFS (Zhang et al. 2016), AMF (Lee et al. 2016), and SDF (Ouyang et al. 2014) use block mapping. For 1Tbyte SSD, mapping table size for page mapping and block mapping corresponds to 1Gbyte and 8Mbyte, respectively. OrcFS requires only 2.2Mbyte for the mapping table at the storage device to manage the metadata region. It requires only 1/4 of the smallest mapping table in the existing works, that of AMF.

There is a critical issue in AMF. To use only block mapping, AMF modifies the F2FS so that the metadata region is updated in an append-only manner. While this eliminates the need to manage the metadata region with page mapping, it can cause cascade updates in the metadata due to the change in the physical location of metadata (Rosenblum and Ousterhout 1992). We suspect that this brings non-negligible overhead. ParaFS proposes block mapping and page mapping for the data region and metadata region, respectively.

Another important issue is the need to use a special interface. For compatibility, it is important that the new storage layer is fully functioning without any changes in the existing applications and does not need a new interface. Among the seven works in Table 1, only OrcFS, ParaFS, and NVMKV are compatible with standard API.

The size of the mapping table dynamically changes in FSDV, and in the worst case, it becomes the same as that of a page mapping table. NVMKV (Marmol et al. 2015) utilizes the mapping table in the Flash storage as its key-value store metadata and removes the host-side metadata. NVMKV still requires a page granularity mapping table in the storage. In addition, NVMKV (Marmol et al.

Table 2. Host System and Storage (Samsung SSD 843TN (SSD843Tn 2014))

	Desktop	Server
CPU	Intel i7-3770	Xeon E5-2630
Mem	8GB	256GB
OS	Ubuntu 14.04 (kernel 3.18.1)	
Storage	Capacity: 256GB (include 23.4GB OVP) 4MB NAND Block size, 8KB Page size Sequential Write: 488MB/sec 4KB Random Write: 102 KIOPS	

Table 3. Summary of Filebench Workload

	Files	File size	Threads	R/W	fsync
fileserver	80,000	128KB	50	33/67	N
varmail	8,000	16KB	16	50/50	Y

2015) and SDF (Ouyang et al. 2014) each have limited usage for a specific workload such as a key-value store.

ANViL (Weiss et al. 2015) lets the host modify logical-to-physical mapping information in the device through new I/O interfaces. However, to support the remap operations, ANViL requires find-grained mapping granularity, such as a page-level mapping table in the storage.

5 EXPERIMENT

5.1 Experiment Setup

We compared the performance of OrcFS against F2FS and EXT4 on Linux Kernel version 3.18.1. OrcFS uses disaggregate mapping, and F2FS and EXT4 use page mapping SSD. We used Samsung SSD 843TN (SSD843Tn 2014) for experiments and modified its firmware to implement OrcFS. The firmware manages the metadata area with page mapping. It performs garbage collection only on the metadata area. The firmware supports one OrcFS file system partition. We leave the work of developing a firmware that supports multiple file system partitions as future work.

To format a partition with OrcFS, the file system and the storage need to know each other's information. The host needs the specification of the Flash storage, such as the capacity, the Flash page size, and the superblock size, while the storage requires the size of the metadata region of the file system. As a prototype, each layer's information is hard-coded into both the file system format utility, `f2fs_tools` (f2fs-tools 2012), and the storage firmware.

Table 2 shows the specification of the host system and SSD 843TN used in the experiment. The superblock size is 256Mbyte. Thus, we set the file system section size in OrcFS to 256Mbyte. The storage performance of Table 2 shows the raw performance of the SSD 843tn (SSD843Tn 2014). We measured the performance with a 10-Gbyte file size and 512Kbyte (4Kbyte) record size for the sequential (random) write. The experiment in Section 5.5 was performed in the server environment given in Table 2, and the other experiments were conducted in the desktop environment.

Filebench (Tarasov et al. 2016) is a benchmark framework for file systems and storages, providing over 40 pre-defined workloads. We use the *fileserver* workload and the *varmail* workload, characteristics of which are given in Table 3. The *varmail* workload describes the workload that

Table 4. Size of Mapping Table (256 Gbyte SSD)

	Mapping Size
Page mapping	256Mbyte
FSDV (Zhang et al. 2015)	≤256Mbyte
Hybrid mapping (Lee et al. 2008)	4Mbyte
Disaggregate Mapping	1Mbyte

occurs in the `/var/mail` directory of the traditional UNIX system. It performs file create, write, and `fsync` operations that occur while a user receives and reads the email. The *fileserver* workload, on the other hand, depicts a fileserver operation in which 50 user threads create files to perform writes, or open existing files to append or delete.

We set the section size of F2FS to 256Mbyte in all experiment. This F2FS configuration is based on our experiment result. We checked the performance of F2FS while varying the size of a section. Compared to the IOPS and WAF of F2FS with section size of 2Mbyte, F2FS with section size 256Mbyte shows 24% higher IOPS and 20% lower WAF.

5.2 Mapping Table Size

Table 4 compares the size of mapping tables in page mapping, FSDV (Zhang et al. 2015), hybrid mapping (Lee et al. 2008), and disaggregate mapping. In the page granularity mapping scheme, a 4-byte sized mapping table entry represents one physical address of a Flash page. Thus, a 1Kbyte mapping table is needed for a 1Mbyte Flash block (4 byte × 256 Flash pages).

Page mapping uses 256Mbyte of memory, while disaggregate mapping of OrcFS uses only 1Mbyte. In EXT4, it is necessary to use page mapping to efficiently process small random writes. Even in the case of F2FS, since the in-place update is performed on the file system metadata region, it is necessary to adopt the page mapping in spite of the fact that the sequential write is dominant in the data area. By using the disaggregate mapping scheme, OrcFS has a mapping table size 1/256 that of EXT4 and F2FS with page mapping.

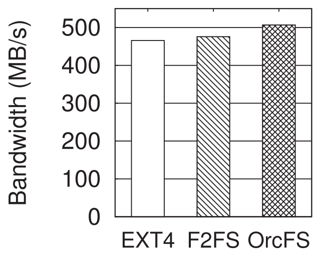
FSDV (Zhang et al. 2015) makes the file system point to a physical address in an SSD, and the pointed entry in the SSD is removed from the mapping table. The size of the mapping table dynamically changes. In the worst case, it has to maintain 256Mbyte of the mapping table, just like the page mapping table.

The memory footprint of OrcFS is 1Mbyte, which corresponds to 1/256 of the size of page mapping. Even though we added several other metadata used by OrcFS, such as segment bitmap and buffered segment number, the size of total metadata is only 4.73 Mbyte, which is 1/54 that of page mapping. Note that LAST FTL consumes 4Mbyte for the mapping table.

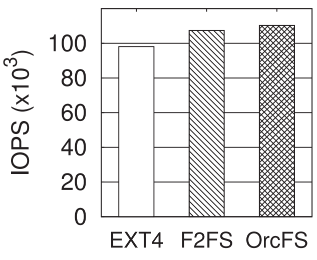
5.3 Primitive IO

We measured the performance of sequential write and random write (Figure 10). We formatted the file system and created a file with a size of 188Gbyte, which is about 80% of the file system partition size. One iteration of an experiment issues 512Kbyte buffered sequential writes until all LBAs are overwritten. We repeated the iteration for fifteen times. Figure 10(a) shows the average performance. The performance of EXT4 and F2FS are 466Mbyte/s and 476Mbyte/s, respectively. The performance of OrcFS shows about 507Mbyte/s, 6% higher than that of F2FS.

The performance gap between EXT4 and OrcFS stands out more in random write workload (Figure 10(b)). To measure the random performance of the device, we format the device and create a 50Gbyte file in the partition. We set the file size to be smaller than the sequential write test



(a) Sequential Write



(b) Random Write

Fig. 10. Sequential/random write performance. Sequential write: 188Gbyte file size, 512Kbyte record size, 2.75Tbyte total write volume/random write: 50Gbyte file size, 4Kbyte record size, 750Gbyte total write volume, section size of F2FS: 256Mbyte.

Table 5. Fio Benchmark (Sequential Write: 188Gbyte File size, 512Kbyte Record Size, Random Write: 50Gbyte File Size, 4Kbyte Record Size, Section Size of F2FS: 256Mbyte)

Workload	EXT4	F2FS	OrcFS
Sequential Write (MB/s)	472	485	506
Random Write (K IOPS)	83	107	112

Table 6. SATA Write Request Size (Fileserver)

	25%	Median	Mean	75%
EXT4	4KB	12KB	120KB	212KB
F2FS	304KB	424KB	360KB	480KB
OrcFS	312KB	432KB	365KB	488KB

in order to prevent the valid block copy operation due to segment cleaning when measuring the performance. The performance comparison of each file system when segment cleaning overhead occurs is shown in Section 5.8. An iteration of the experiment touches all the LBAs with 4Kbyte buffered random writes, and the graph shows the average of fifteen iterations. OrcFS is 12% faster than EXT4; IOPS of OrcFS is 110.3 KIOPS and Ext4 is 98.1 KIOPS.

Using the same workload, we measured the performance of each file system with FIO benchmark (Axboe 2005), and Table 5 shows the results. The benchmark results are similar to those shown in Figure 10(a) and Figure 10(b).

5.4 Macro Benchmark

We used two workloads: *fileserver* and *varmail* from Filebench (Tarasov et al. 2016). Table 3 shows the summary of the each *fileserver* workload. *fileserver* workload creates 80,000 files with a size of 128Kbyte and creates 50 threads to issue reads or buffered appends with a ratio of 1:2. In the *fileserver* workload, on average, the size of append requests is 16Kbyte. *varmail* creates eight thousand 16Kbyte files using 16 threads to create and delete the files with a ratio of 50:50, and each operation is followed by `fsync()`. We ran each workload for 60s.

Figure 11 shows the result of filebench using *fileserver* and *varmail* workload. The performance is normalized with respect to the performance of EXT4. In the case of *fileserver* workload, OrcFS shows 60% better performance than EXT4. Table 6 shows the quartile statistics of the size of

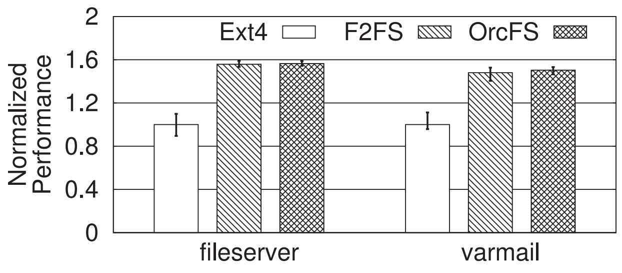


Fig. 11. Filebench.

Table 7. Block Patching Overhead

Workload	Dummy Write	Total Write	Ratio
varmail (Section 5.4)	514MB	1.8GB	28%
YCSB (Section 5.5)	196MB	74GB	0.3%
Rand W (Section 5.8)	2GB	113GB	2%

writes in the *fileserver* workload. The median write request size of OrcFS (432Kbyte) is about 36 times larger than that of EXT4 (12Kbyte). The log-structured file system merges the received write requests into larger one before dispatching them to the storage. The number of write requests in OrcFS is about the half that in EXT4 (119,482).

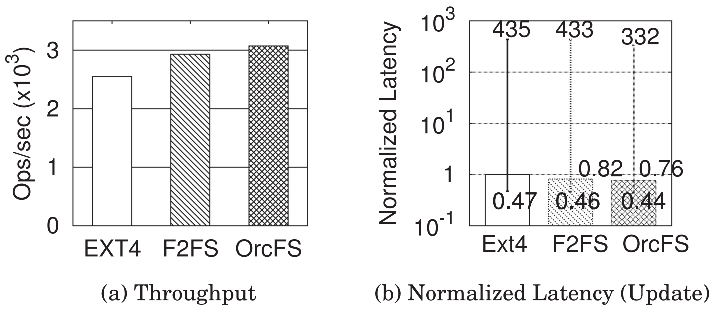
The result is similar in *varmail* workload. The performance of OrcFS is 50% better than that of EXT4 (20Mbyte/s). EXT4 shows slower performance than the other two file systems, because EXT4 sends a lot of `discard` commands to notify the state of the deleted file blocks to the storage device. While the way to handle a discard command varies by SSD vendor, it requires invalidating the mapping table entry for the deleted file blocks and can accompany non-negligible overhead (Saxena and Swift 2010). F2FS and OrcFS also issue the `discard` command. However, the overhead is not as significant as in the case of EXT4, because a much smaller number of discard commands are being issued.

In *fileserver* workload, the throughput of OrcFS (341 Mb/s) is similar to that of F2FS (337Mb/s). The macrobench workloads do not invoke cleaning operations in either the file system or the storage. Thus, F2FS does not suffer from the compound segment cleaning overhead and shows performance close to that of OrcFS. The SATA write request size of OrcFS is 5 Kbyte larger than that of F2FS (Table 6) on average. This is because block patching adds dummy pages to write requests and aligns them with the Flash page size (8Kbyte).

5.5 Key-Value Store Workload

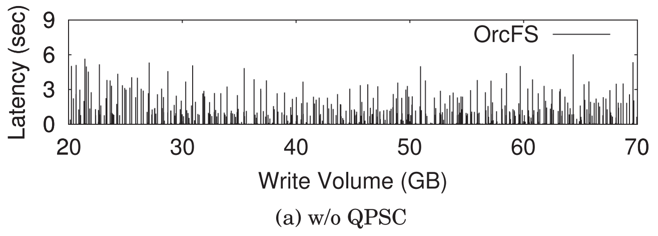
We use Cassandra DB (Lakshman and Malik 2009) with YCSB to measure the performance of EXT4, F2FS, and OrcFS. Cassandra DB adopts the log-structured merge tree (O' Neil et al. 1996) as its essential data structure. For the Cassandra DB storage engine, we use Apache Cassandra 2.2.5. The size of the DB used in the experiment is 20Gbytes. We used YCSB Workload-A to perform read and update operation with a ratio of 50:50.

Figure 12(a) shows the overall throughput of read/update operation. The throughput of OrcFS (3,145Kops/s) is about 21% and 15% higher than those of EXT4 and F2FS, respectively. Figure 12(b) shows the latency of update operation. The results are normalized against EXT4's latency. The latency in OrcFS is about 24% lower than that of the EXT4. EXT4 generated 20% more write commands than OrcFS. While the maximum latency of OrcFS is 25% lower than that

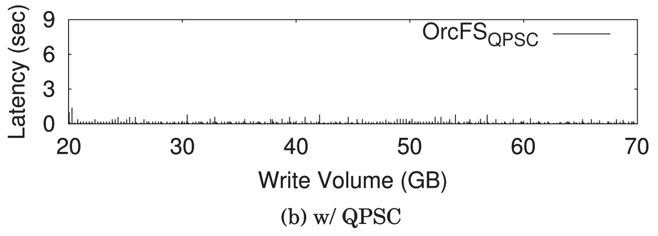


(a) Throughput (b) Normalized Latency (Update)

Fig. 12. Cassandra (YCSB).



(a) w/o QPSC



(b) w/ QPSC

Fig. 13. Write latency (110Gbyte cold file, 80Gbyte hot file, 4Kbyte buffered random write to hot file).

of F2FS, the throughput of OrcFS shows marginal improvement for the same reason as the Macro Benchmark. The performance difference between OrcFS and F2FS under a heavy segment cleaning scenario is discussed in Section 5.8.

5.6 Quasi-Preemptive Segment Cleaning

We examined the effect of segment cleaning over the latency of foreground I/O. We perform 4Kbyte buffered `write()` and create a situation where there run out of the free sections. When the free sections run out, the segment cleaning module of both F2FS and OrcFS cleans the sections until it creates at least one additional section. If the section utilization is ρ, then it needs to clean $\frac{1}{1-\rho}$ sections to create one additional free section. If ρ is 0.6, then the segment cleaning module consolidates the blocks from at least three sections. With a 256Mbyte section, it has to scan 800Mbyte of file system blocks and migrates nearly 600Mbyte of data blocks to a destination section. If this operation interferes with the ongoing I/O, then the application may experience an intolerable amount of delay. Thus, we examined the latency of the 4Kbyte buffered `write()` operation.

We set the polling interval to 100ms. Figure 13 illustrates the result. We only showed the interval where the segment cleaning is active. As shown in Figure 13(a), without proper management, the latency of a 4Kbyte `write()` can be as high as 9s, which is prohibitive. The stock F2FS also shows

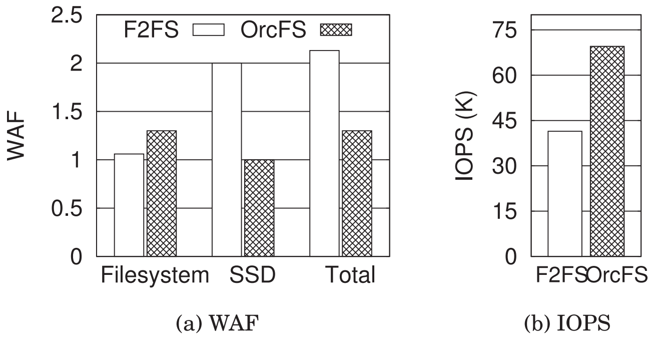


Fig. 14. WAF and IOPS of each system (85Gbyte cold file, 85Gbyte hot file, 4Kbyte buffered random write to the hot file, 170Gbyte total write volume, section size of F2FS: 256Mbyte).

the same write latency as shown in Figure 13(a). With a 100ms polling interval in QPSC, we could not observe any abnormal delay that has been observed in F2FS. In OrcFS, the latency of write operation is less than one sec. QPSC successfully resolves the excessive segment cleaning overhead that may appear in large-size section-based space management.

5.7 Block Patching Overhead

Depending on the nature of the workload, the space overhead of the block patching can vary widely. We examined the block patching overhead in three workloads which we have used in this study: random write, *varmail*, and YCSB workload, each of which has widely different access characteristics. YCSB benchmark uses a log-structured merge tree, which flushes the dirty page cache entries in the unit of tens of Mbytes. The overhead of block patching is negligible (0.3%). On the other hand, *varmail* workload frequently calls `fsync()` to synchronize the updated file metadata to the storage. In this workload, block patching entails 28% overhead. Despite the overhead, OrcFS shows similar performance to F2FS in the *varmail* workload (Figure 11). This is because block patching does not need to create additional Flash page writes to align the size of write requests with the Flash page size. For random write, the block patching overhead is 2%.

5.8 Effect of Eliminating the Redundancies

We measured the effectiveness of eliminating the redundancies from the perspective of write amplification as well as from the perspective of the performance. The workload is carefully crafted to trigger the garbage collection at the storage device as well as at the segment cleaning at the file system.

The garbage collection and the segment cleaning behavior are governed by the free space at the file system, and the size of the over-provisioning area at the Flash storage. For a 256Gbyte SSD, we created a 170Gbyte file. For the 170Gbyte file, we designated the half of the file as the hot region. We performed random write only to the hot region of the file. The random writes (buffered 4Kbyte writes) are created such that each of the file blocks in the hot region is written one and only once. A single iteration ends if all blocks are over-written. We ran the fifteen iterations of this workload. Write amplification is measured using `smartmontools` (smartmontools 2010). We measured performance and write amplification factor (WAF) for OrcFS and F2FS. We applied both QPSC and block patching to OrcFS.

First, we examined the performance. F2FS and OrcFS yield 53K IOPS and 75K IOPS, respectively (Figure 14(b)). OrcFS shows 45% performance gain against F2FS. Second, we examined the write

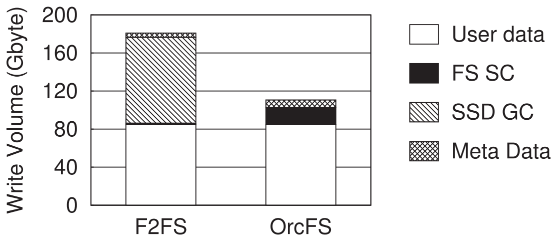


Fig. 15. Total write volume distribution (4Kbyte buffered random write, write volume generated by the application is 85Gbyte, section size of F2FS: 256Mbyte).

amplification created by each layer. Figure 14(a) illustrates the result. In the file system level, F2FS and OrcFS amplify the write by 1.06 and 1.3, respectively. Due to the larger segment cleaning unit size, it is inevitable that OrcFS inefficiently consolidates the file blocks and therefore has a larger write amplification. SSD in F2FS and in OrcFS exhibit a stark contrast in write amplification. In F2FS, the underlying SSD amplifies the write operation by 2. This is due to garbage collection in the storage device. On the other hand, in OrcFS, since the file system is in charge of consolidating the physical Flash storages, the storage device itself does not entail any amplification. F2FS and OrcFS yield 2.13 and 1.3 write amplification, seen from the storage device. OrcFS eliminates 1/3 of the write volume and it leads 68% performance improvement.

Third, we analyzed the cause for this amplification. Figure 15 illustrates the result. The application creates 85Gbyte of write to the file system data region in both F2FS and OrcFS. In F2FS and OrcFS, the file systems create 4Gbyte and 8Gbyte of metadata update, respectively. The reason that OrcFS updates more metadata than that of F2FS is because OrcFS writes only in append-only style, whereas F2FS exploits threaded logging (Lee et al. 2015) to reduce the number of segment cleaning when the file system utilization is high. Dirty data in node segments also has to be flushed to the storage at the time of checkpoint which is called at the end of a segment cleaning. In F2FS, we observed that Flash storage creates a lot of additional writes by itself due to garbage collection. Meanwhile, OrcFS is free from this behavior yielding much more efficient storage behavior.

The write volume governs the lifespan of the Flash storage. Roughly, eliminating the 1/3 of total write volume leads to the 30% extension of the lifespan.

6 RELATED WORK

Flash-based File System: There are a number of efforts, such as YAFFS (Manning 2010), JFFS2 (ZHANG and QIU 2006), LogFS (Engel and Mertens 2005), and UbiFS (Hunter 2008), to manage the raw Flash device on the host side. Since the device lacks an FTL, the host file system has to provide the mechanism for bad block management, wear-leveling, and garbage collection. Flash-based file systems do not use the Flash Translation Layer (FTL) to manage the flash device directly by the host. Therefore, they can not be applied to Flash storage having a multi-channel/multi-way structure with multiple Flash packages. And, they only support one Flash memory device.

FusionIO introduced Direct File System (DFS) (Josephson et al. 2010) which is a file system for a high-performing SSD. DFS keeps the Virtualized Flash Storage Layer (VFSL) to manage block allocation and inode management, which used to be the role of the file system. Because DFS directly manages the mapping of the SSD, it does not suffer from compound segment cleaning problem. However, as DFS simply brings the virtual to physical address mapping information at the device level to the host device driver, it does not have a benefit in size of mapping table.

Reducing the Metadata Overhead: To relieve the burden of the device memory requirement, FSDV (File System De-Virtualizer) (Zhang et al. 2015) provides a way for the host to manage the physical address mapping information. It reduces the mapping table management overhead by removing the logical to physical address mapping information on SSD. Unlike FSDV which removed the overhead of device mapping information, NVMKV (Marmol et al. 2015) removes the host side metadata management overhead by replacing FTL commands with key-value operations. NVMKV uses commands like atomic multiple-block `write`, `p-trim`, `exits`, and `iterate`, as the interface to the Flash device.

Host Managed Storage System: ANViL (Weiss et al. 2015) lets the host modify logical-to-physical mapping information on a device through new I/O interfaces. The host exploits the given interfaces to remove the redundant data copy overhead. It also provides useful features to the system, such as single journaling and file snapshot.

Compound Segment Cleaning: The compound segment cleaning problem on stacked log system was first elucidated by Yang et al. Yang et al. (2014). Since then, Baidu introduced SDF (Software-defined Flash) (Ouyang et al. 2014), Lee et al. introduced AMF (Application Managed-Flash) (Lee et al. 2016), and Zhang et al. proposed ParaFS (Zhang et al. 2016). AMF (Application Managed-Flash) (Lee et al. 2016), which is based on F2FS (Lee et al. 2015), exploits the fact that log-structured applications always append the data to reduce the size of metadata for Flash-based storage. AMF modifies the metadata area of F2FS so that it can also be managed in a log-structured manner; thus, it can exploit the block mapping in the storage layer. Since both layers of the file system and the Flash device can use block mapping, it gives a partial solution to the compound segment cleaning problem.

ParaFS (Zhang et al. 2016) acquires hardware information for the storage to identify the number of channels the device has. ParaFS exploits the information to distribute the data according to their hotness and to maximize the channel parallelism. It also implements a coordinated garbage collection scheme and parallelism-aware scheduling to solve the compound segment cleaning and load balancing.

7 CONCLUSION

Modern I/O stacks are full of unnecessary redundancies with duplicate efforts across the layers: the storage device, host file system, and even the application. To fully exploit the append-only and asymmetric read/write latency nature of the Flash storage device, these three layers endeavor to align their behavior with the physical characteristics of the Flash storage. Each of these layers introduces the level of indirection maintaining its address translation information using a log-structured file system or using append-only search structures such as log-structured merge tree. Subsequently, the file system and the storage device reserve a certain fraction of its space to consolidate the invalid blocks. This duplicate effort leads to the waste of storage space, increase in the write-amplification, and performance degradation. Existing studies to address these redundancies are also difficult to apply to real desktop or server environment due to segment cleaning overhead and the additional write amplification caused by the mismatch between the file system block size and Flash page size in a block mapping FTL. In this work, we propose OrcFS, Orchestrated File System, that orchestrates the behavior of the Flash-friendly file system and the Flash storage. OrcFS partitions the storage space into two regions and assigns the management role of individual partitions to different layers of the stack: the file system and the storage device. OrcFS fully exploits the internal parallelism of the underlying Flash storage (multi-channel and multi-way) via making the unit of segment cleaning equivalent to the superblock of the underlying Flash storage. Novel QPSC algorithm effectively resolves the excessive I/O delay which may entail due to large segment cleaning unit size. Block patching implementation enables OrcFS to be operated on various Flash storages by eliminating partial Flash page writes.

The benefit of the OrcFS is substantial. It reduces the device mapping table size to 1/465 and to 1/5 from the page mapping and the smallest mapping table size proposed in the literature, AMF, respectively. Despite its minimal hardware resource requirement, the OrcFS exhibits superior performance compared to EXT4 and F2FS over page mapping based SSD in a broad range of workloads: synchronous random write dominant workload including *varmail* workload. Via eliminating the compound segment cleaning efforts in the file system and the storage device, we eliminate 1/3 of the write volume to the storage device leading to the roughly 30% extension of its lifespan.

REFERENCES

Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 57–70.

Jens Axboe. 2005. Fio-flexible i/o tester synthetic benchmark. *URL https://github. com/axboe/fio (Accessed: 2015-06-13)* (2005).

Kyle Banker. 2011. *MongoDB in Action*. Manning Publications Co.

Frank Berry. 2015. Enterprise flash storage: Who's adopting them and why. *Proceedings of the Flash Memory Summit, Santa Clara, CA* (2015).

Daniel Campello, Hector Lopez, Ricardo Koller, Raju Rangaswami, and Luis Useche. 2015. Non-blocking writes to files. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 151–165.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4–4.

Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. 2007. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *Proceedings of the ACM Annual Design Automation Conference*. 212–217.

ChosunBiz. 2016. http://biz.chosun.com/site/data/html_dir/2016/08/12/2016081202016.html?main_box. (2016).

David Chow, Charles Lee, Abraham Ma, Frank Yu, Edward Lee, Ming-Shiang Shen, and others. 2007. Managing bad blocks in various flash memory cells for electronic data flash card. (2007). US Patent No. 11/864,684.

Christian Czezatke and M. Anton Ertl. 2000. LinLogFS-a log-structured file system for linux. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 77–88.

John D. Davis, Laura Caulfield, and Steve Swanson. 2013. Flash trends: Challenges and future. In *Proceedings of the IEEE Hot Chips 25 Symposium (HCS)*. IEEE, 1–42.

Jörn Engel and Robert Mertens. 2005. LogFS-finally a scalable flash file system. In *Proceedings of the 12th International Linux System Technology Conference*.

f2fs-tools. 2012. Formatting Tools for Flash-Friendly File System. http://git.kernel.org/cgit/linux/kernel/git/jaegeuk/f2fs-tools.git. (2012).

S. Ghemawat and J. Dean. 2014. LevelDB, A fast and lightweight key/value database library by Google. (2014).

Ayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. 229–240.

Adrian Hunter. 2008. A brief introduction to the design of UBIFS. In *Proceedings of the the Rapport Technique*.

William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. 2010. DFS: A file system for virtualized flash storage. *ACM Trans. Stor.* 6, 14 (2010), 14:1–14:25.

Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 13–13.

Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the the 6th ACM & IEEE International Conference on Embedded Software*. 161–170.

Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. 1995. A flash-memory based file system. In *Proceedings of the the USENIX Anual Technical Conference (ATC'95)*. 155–164.

Joohyun Kim, Haesung Kim, Seongjin Lee, and Youjip Won. 2010. FTL design for TRIM command. In *Proceedings of the the 15th International Workshop on Software Support for Portable Storage*. 7–12.

Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. 2002. A space-efficient flash translation layer for CompactFlash systems. *IEEE Consum. Electron.* 48, 2 (2002), 366–375.

Kingston Technology. 2013. Understanding over-provisioning. (2013).

Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux implementation of a log-structured file system. *ACM SIGOPS Operat. Syst. Rev.* 40, 3 (2006), 102–107.

Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2010. Janus-FTL: Finding the optimal point on the spectrum between page and block mapping schemes. In *Proceedings of the the ACM International Conference on Embedded Software (EMSOFT'10)*. 169–178.

Ohhoon Kwon, Jaewoo Lee, and Kern Koh. 2007. EF-greedy: A novel garbage collection policy for flash memory based embedded systems. In *Computational Science (ICCS'07)*. Springer, 913–920.

Avinash Lakshman and Prashant Malik. 2009. Cassandra: Structured storage system on a P2P network. In *Proceedings of the the 28th ACM Symposium on Principles of Distributed Computing (PODC'09)*. 5–5.

Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the the USENIX Conference on File and Storage Technologies (FAST'15)*. 273–286.

Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. 2011. A semi-preemptive garbage collector for solid state drives. In *Proceedings of the IEEE Performance Analysis of Systems and Software (ISPASS'11)*. 12–21.

Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and others. 2016. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 339–353.

Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. In *Proceedings of the ACM SIGOPS Operating Systems Review*, Vol. 42. 36–42.

Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.* 6, 3 (July 2007), Article 18.

Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. 1–13.

Charles Manning. 2010. How YAFFS works. Retrieved April 6, 2010 from https://yaffs.net/documents/how-yaffs-works.

Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*. 207–219.

Lucas Mearian. 2016. SSD prices plummet again, close in on HDDs: Prices dropped by 12 percent in just the last quarter alone. Retrieved from http://www.pcworld.com/article/3040591/storage/ssd-prices-plummet-again-close-in-on-hdds.html.

Micron. 2016. Technology Innovation Redefined. Retrieved from https://www.micron.com/~/media/documents/products/product-flyer/3d_nand_flyer.pdf.

Patrick O' Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O' Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (1996), 351–385.

Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage system. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 471–484.

JungWook Park, Gi-Ho Park, Charles Weems, and ShinDug Kim. 2009. Sub-grouped superblock management for high-performance flash storages. *IEICE Electron. Express* 6, 6 (2009), 297–303.

RocksDB. 2014. A persistent key-value store for fast storage environments. Retrieved from http://rocksdb.org/.

Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10 (1992), 26–52.

Samsung. 2015. Next generation Samsung 3bit V-NAND Techonology. Retrieved from http://www.samsung.com/semiconductor/global/file/insight/2015/08/3bit_V-NAND_technology_White_Paper-1.pdf.

Samsung Electronics Co. 2014. Over-provisioning: Maximize the lifetime and performance of your SSD with small effect to earn more. Application note. (2014).

Mohit Saxena and Michael M. Swift. 2010. FlashVM: Virtual memory management on flash. In *Proceedings of the USENIX Annual Technical Conference (ATC'10)*. 14–14.

Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. 1995. File system logging versus clustering: A performance comparison. In *Proceedings of the the USENIX Technical Conference Proceedings*. 21–21.

Frank Shu and Nathan Obr. 2007. Data set management commands proposal for ATA8-ACS2. *Management* 2 (2007), 1.

smartmontools. 2010. smartmontools package. Retrieved from http://sourceforge.net/apps/trac/smartmontools/wiki.

Kent Smith. 2011. Garbage collection. In *Proceedings of the Flash Memory Summit*. 1–9.

SSD843Tn. 2014. Samsung, SSD 843tn Specification. Retrieved from http://enterprise.m2m-direct.co.uk/downloads/resources/SAMSUNG%20Channel%20Info%20Memory%2010-14.pdf.

StarWind. 2014. Log-Structured File System^{TM}. Retrieved from https://www.starwindsoftware.com/vm-centric-storage-lsfs.

Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *USENIX Login Mag.* 41 (2016).

Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2015. ANViL: Advanced virtualization for modern non-volatile memory devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 111–118.

Guanying Wu and Xubin He. 2012. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'12)*. 10–10.

Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'17)*. 22:1–22:26.

Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Dont´stack your log on my log. In *Proceedings of the Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*.

Yudong Yang, Vishal Misra, and Dan Rubenstein. 2015. On the optimality of greedy garbage collection for SSDs. *ACM SIGMETRICS Perform. Eval. Rev.* 43, 2 (Sept. 2015), 63–65.

Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choil, Sungroh Yoon, and Jaehyuk Cha. 2013. Vssim: Virtual machine based ssd simulator. In *Proceedings of the the IEEE Mass Storage Systems and Technologies (MSST'13)*. 1–14.

Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: A log-structured file system to exploit the internal parallelism of flash devices. In *Proceedings of the USENIX Annual Technical Conference (ATC'16)*. 87–100.

Yiying Zhang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Removing the costs and retaining the benefits of flash-based SSD virtualization with FSDV. In *Proceedings of the Conference on Mass Storage Systems and Technologies (MSST'15)*. 1650–1665.

Yong Zhang and Xue-hong Qiu. 2006. Implementation of JFFS2 file system in embedded linux system. In *Proceedings of the Computer Technology and Development*, Vol. 4. 48–48.

Da Zheng, Randal C. Burns, and Alexander S. Szalay. 2015. Optimize unsynchronized garbage collection in an SSD array. *Computing Research Repository*, Vol. abs/1506.07566. 1–7.

Received November 2016; revised September 2017; accepted November 2017