Mitigating Synchronous I/O Overhead in File Systems on Open-Channel SSDs

YOUYOU LU, JIWU SHU, and JIACHENG ZHANG, Tsinghua University

Synchronous I/O has long been a design challenge in file systems. Although open-channel solid state drives (SSDs) provide better performance and endurance to file systems, they still suffer from synchronous I/Os due to the amplified writes and worse hot/cold data grouping. The reason lies in the controversy design choices between flash write and read/erase operations. While fine-grained logging improves performance and endurance in writes, it hurts indexing and data grouping efficiency in read and erase operations. In this article, we propose a flash-friendly data layout by introducing a built-in persistent staging layer to provide balanced read, write, and garbage collection performance. Based on this, we design a new flash file system (FS) named *StageFS*, which decouples the content and structure updates. Content updates are logically logged to the staging layer in a persistence-efficient way, which achieves better write performance and lower write amplification. The updated contents are reorganized into the normal data area for structure updates, with improved hot/cold grouping and in a page-level indexing way, which is more friendly to read and garbage collection operations. Evaluation results show that, compared to recent flash-friendly file system (F2FS), StageFS effectively improves performance by up to 211.4% and achieves low garbage collection overhead for workloads with frequent synchronization.

CCS Concepts: • **Information systems → Flash memory**; • **Software and its engineering → File systems management**;

Additional Key Words and Phrases: Synchronous I/O, file system, flash memory, open-channel SSD

ACM Reference format:
Youyou Lu, Jiwu Shu, and Jiacheng Zhang. 2019. Mitigating Synchronous I/O Overhead in File Systems on Open-Channel SSDs. *ACM Trans. Storage* 15, 3, Article 17 (May 2019), 25 pages.
https://doi.org/10.1145/3319369

1 INTRODUCTION

Synchronous I/O has been a design challenge in file systems (FS) for a long time. For slow magnetic disks, synchronous I/O causes high latency and incurs high performance overhead. File system updates are required to be synchronized to persistent storage for either system consistency or data durability. To reduce consistency-induced synchronization (i.e., I/O synchronization due to

This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61772300, 61832011), Research and Development Plan in Key Field of Guangdong Province (Grant No. 2018B010109002), and Huawei Technologies Co. Ltd. Youyou Lu is also supported by the Young Elite Scientists Sponsorship Program of China Association for Science and Technology.

Authors' addresses: Y. Lu, J. Shu (corresponding author), and J. Zhang, Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China; emails: {luyouyou, shujw}@tsinghua.edu.cn, zhang-jc13@mails.tsinghua.edu.cn.

1553-3077/2019/05-ART17 $15.00
https://doi.org/10.1145/3319369

system consistency maintenance), a lot of efforts have been tried to keep the write ordering for updates with dependencies, instead of forcing them to be written back to persistent storage [14, 17, 37, 39, 43]. While consistency-induced synchronization has been intensively studied, durability-induced synchronization (i.e., I/O synchronization due to data durability) remains a challenge. Unfortunately, from either explicit synchronization calls (e.g., *fsync*) in applications or background daemons (e.g., *bdi-flush*) in operating systems, I/O synchronization[1] is frequently called in both desktop [14, 17, 20, 39, 43] and mobile applications [23, 25].

Solid state drives (SSDs) are getting widely used in enterprise storage systems recently, mostly due to the low latency and high bandwidth features. However, even though the access latency of a synchronous I/O is significantly reduced in flash-based SSDs, I/O synchronization still blocks other I/Os. This not only degrades system performance, but also makes SSDs worn-out more quickly. In this article, we have two observations that motivate us to revisit the design of file system writes.

The first observation is that write traffic is dramatically amplified for partial-page writes or writes that have good temporal write locality, which, however, is unnecessary. The amplified write traffic not only slows down the performance, but also accelerates the wear-out process of flash memory, thus reducing the lifetime of SSDs. In both server and mobile workloads, there are a large number of partial page writes, which write only part of a page (less than 4KB) [29, 32]. As writes in flash memory are performed in page units, a page with only a small part updated has to be written back in a full page. This leads to high write amplification. In addition, with frequent synchronization, data pages have not been well coalesced when being written back. One page may be written back multiple times, even when it will be updated soon with good temporal locality. This also reduces the chances of cache coalescing.

The second observation is that frequent I/O synchronization makes data layout more challengeable. Inferior data layout may degrade garbage collection performance. Different from magnetic disks, flash memory has to erase a flash block before overwriting a flash page. A flash device performs updates in a no-overwrite way and uses the garbage collection process to reclaim flash blocks. Data layout, especially hot/cold data grouping, significantly affects the performance of garbage collection. A log-structured file system, like file-friendly file system (F2FS) [26] or ParaFS [52], divides file system data and metadata into different numbers (e.g., 6) of groups. However, inside an SSD, there is only one log head when flash pages are sequentially allocated, or a limited number of log heads (also called open blocks, which are the flash blocks that are currently used for allocation) due to limited embedded memory inside SSDs. When data groups are written back to flash memory, they share the log heads if the number of data groups is smaller than the number of log heads. In this case, effects of data grouping are affected by not only the goodness of an algorithm but also the input data amount. Even sophisticated algorithms can hardly exploit the benefit from small datasets. Unfortunately, with frequent I/O synchronization, fewer data blocks are written back at one time, making the datasets small. This can not be mitigated even with large memory capacities. I/O synchronization weakens the efforts from data placement and grouping, and makes system performance poorer.

Open-channel SSDs remove the flash translation layer (FTL) in the firmware to export direct access interface to the software, and thus remove redundancy functions in the I/O path and provide better performance and endurance [10, 28, 36, 52]. Synchronous I/Os remain a challenge in open-channel SSDs. Fortunately, open-channel SSDs provide the accesses to physical addresses to system software, which allows us to flexibly organize and place the data in flash memory.[2]

[1]In this article, we use *synchronous I/O* and *I/O synchronization* interchangeably, to refer to the write operations that are required to be written to persistent storage media.
[2]While commercial SSDs can also benefit from the staging phase, the patching phase benefit can not been ensured.

In this article, we propose a file system, StageFS, to introduce a *staging* phase to absorb synchronized writes before updating the file system image (i.e., the *patching* phase). StageFS decouples the content (e.g., page, dirent, inode) and structure (e.g., bitmap) updates. In the staging phase, the content updates are logically logged to the persistent staging layer. Small writes from synchronous I/Os are updated in the record-level logging way, i.e., they are appended using byte-addressable record units rather than page units. In this way, writes are performed sequentially and efficiently, with reduced write size compared to page-aligned writes. Record-level logging is not read-friendly, but the staging data are recently accessed and have a high probability to be in the memory cache. Therefore, their cache copies are pinned in main memory to provide fast reads.

The file system image is protected from being written by synchronous writes. It is only patched when the staging area runs out of space, with a reorganized data layout. In the *patching* phase, more data blocks can be buffered in the staging area. With accumulated data blocks, a file system makes better decisions on data placement and grouping. In this way, StageFS improves the garbage collection performance by updating the file system using an optimized data layout.

Our contributions are summarized as follows:

— We propose a new flash file system architecture, StageFS, which introduces a built-in persistent layer to efficiently absorb synchronized writes and protect the file system image from small random writes.
— We use logical logging to provide data durability efficiently in the staging phase, improving both performance and endurance when persisting data, without sacrificing read performance.
— We lazily perform data allocation and grouping until the patching phase, so as to update the file system image with an optimized data layout, aiming at improving read and garbage collection performance.
— We implement and evaluate StageFS in the Linux kernel. Evaluation results show that, for workloads with synchronous I/Os, StageFS effectively improves performance and reduces garbage collection overhead compared to legacy file systems.

The rest of the article is organized as follows. Section 2 gives the background of flash memory and persistent memory/storage extensions. Sections 3 and 4 describe the design and implementation of StageFS. Evaluation results are shown in Section 5. Section 6 discusses the related work, and Section 7 concludes the article.

2 BACKGROUND

2.1 Flash Memory and Open-Channel SSDs

Flash memory has two major properties that are different from magnetic disks. The first is the *no-overwrite* way of data updates. Updates are redirected to free pages, while leaving the old pages for recycling in the garbage collection (GC). Since flash memory is erased in the units of flash blocks, the GC process moves valid pages out before erasing the selected flash blocks. Thus, data grouping, which groups data pages with similar hotness into different flash blocks, is essential to keep the garbage collection overhead low [38].

The second is the *endurance* problem. A flash memory cell can endure a limited number of program/erase (P/E) cycles. The number of P/E cycles decreases dramatically when the memory density increases, e.g., 100,000 in single-level cell (SLC) and 1,000 in triple-level cell (TLC) [18]. A memory page wears out as the P/E cycle approaches the limit. Wear leveling [9] and write reduction [12, 35, 36, 50] need to be carefully designed to alleviate this problem.

Open-Channel SSD generally refers to the SSD that has no FTL inside the device and opens up the physical layout to the host. It is also called raw-flash device [36]. Raw-flash device first appears in host-FTL based SSD architecture, which moves the FTL from the device to the host [4]. The raw-flash device is first opened up to the system software for direct management in Ref. [36], which directly performs read, write, and erase operations from the operating system. This software-managed flash way is later called software-defined flash [42] and application-managed flash [28]. In this article, open-channel SSD specifically refers to the latter one [28, 36, 42], in which an SSD supports direct software management without device-based or host-based FTL. By removing the FTL, open-channel SSDs provide flexible control of flash memory operations to the software. Meanwhile, it brings new challenges in programming, including data indexing [10, 36], space allocation [52], I/O scheduling [49, 52], and data layout. Open-channel SSDs are able to lay out data to physical addresses, but choosing a layout remains a challenge to balance different operations.

2.2 Controversy Layout Choices between Flash Operations

Flash write operations favor fine-grained updates [36] and the log-structured update way [26], which in combination is called *persistence-efficient way*. The fine-grained updates reduce the write traffic, and thereby improve performance and endurance of flash storage. The log-structured update way matches well with the no-overwrite feature of flash memory. However, the two ways of data layout that are preferred by the write operation hurt the read and garbage collection operations. First, the fine-grained update way incurs high overhead in data indexing. The indexing metadata consumes higher space when data are indexed in a fine-grained way. The fine-grained update way also leads to irregular data pieces, which results in high search cost. Second, the log-structured way hurts hot/cold data grouping, especially for synchronous I/Os. For synchronous I/Os, there is little chance for data grouping before they are persisted to flash memory. While in-place updates keep access locality for data blocks in each file, the log-structured way loses both access locality and hot/cold data grouping. As such, it is difficult to choose a single layout that is suitable for different operations.

However, *there is a window between the write and the read operations on flash memory due to page cache in the memory*. The recently written data have a higher probability to be in the page cache. This time period can serve read requests in the memory cache, and does not hurt read performance, even when data are persisted in a persistence-efficient way. Therefore, we leverage this time period as a stage phase, and introduce a stage area in flash memory to make the co-existed data layout possible. For flash writes, data are first persisted to the staging area in a log-structured fine-grained logging way in the stage phase, and then are grouped and reorganized to the normal data layout.

2.3 Persistent Memory/Storage Extensions

Keeping different layouts in a single storage system is not unique to our design. Efforts have been made in both main memory and secondary storage levels to extend the storage functions (e.g., performance, capacity, consistency).

In the main memory level, non-volatile RAM (NVRAM) has been used as the persistent buffer to improve performance (i.e., *performance extension* [13, 21]) or the journaling to ensure consistency (i.e., *consistency extension* [27]). For instance, NetApp Write Anywhere File Layout (WAFL) [21] employs a non-volatile RAM to buffer Network File System (NFS) requests. Since WAFL is designed for snapshots, its consistency is easily switched using the root nodes (i.e., the consistency point). The used NVRAM keeps a log of requests after the last consistency point, to replay after system crashes. The memory-level NVRAM also reduces the response time of write requests. However, these memory-level storage extensions have drawbacks. First, it is difficult to maintain

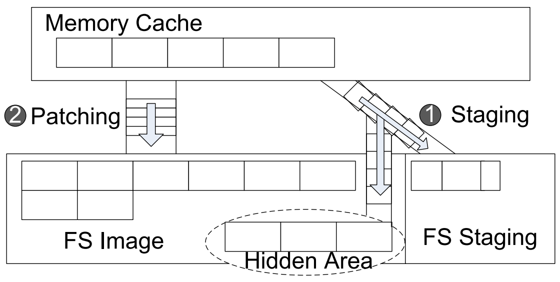


Fig. 1. Overview of StageFS.

the data management in both memory and storage levels. Second, the memory-level data may get inconsistency when the volatile data in the CPU cache gets lost after unexpected failures [15, 33, 34, 44, 48].

In the secondary storage level, part of the storage space has also been allocated to enhance I/O functions, such as swap in virtual memory and file system journaling. Virtual memory allocates the swap area in secondary storage to extend the capacity of main memory (i.e., as the *capacity extension*). When the memory runs short of space, some memory pages are swapped to the swap area. Programs see a virtual memory whose size is larger than that of the physical memory. The majority of modern file systems allocate a journaling area in secondary storage and write file system updates to this area [47]. Only when writes in the journaling area are completely updated, they can be checkpointed to their home locations in file system image. The journaling area is the *consistency extension* to ensure the consistency of file systems.

Our proposed StageFS is a storage extension in the secondary storage level and is a built-in component of a file system. It is designed for flash memory as the *performance extension*, with the goal to improve both the persistence performance and the garbage collection performance for workloads with frequent synchronization.

3 STAGEFS DESIGN

We describe the design of StageFS in this section. We start by introducing the overview of StageFS, and then present the two phases of StageFS:

(1) *Staging with Logical Logging* to efficiently provide durability to file system writes in the staging area without sacrificing the read performance,
(2) *Patching with Data Reorganization* to patch the staging data to the file system image with reorganized data layout, so as to improve garbage collection performance.

And finally, we discuss the consistency of StageFS.

3.1 Overview of StageFS

StageFS introduces a new persistent layer named *staging* inside the file system. As shown in Figure 1, StageFS has two phases when updating data pages: the *staging* phase and the *patching* phase. The staging phase has dedicated storage space that is independent of the file system image storage. StageFS consists of two parts in the persistent storage space: the *file system staging* and the *file system image*. File system updates are first performed in the FS staging in *a persistence-efficient way*, and then are written back to the file system image with *optimized data layout*, aiming at better garbage collection performance.

To improve efficiency of data persistence, StageFS decouples content and structure updates and writes back the content updates using record-level logging in the staging phase. In this article, we

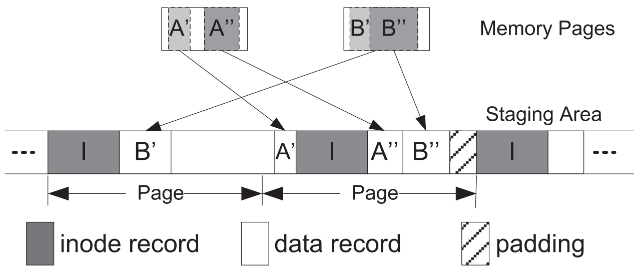


Fig. 2. Illustration of record-level logging.

refer to the file or directory updates as the content updates, and the allocation metadata updates as the structure updates. Updates to files or directories have their dirty bytes written to the staging area or are logged to the hidden area in the file system image. The hidden area has space allocated from the file system image area. The structure metadata is also updated. But data pages in this hidden area are not logically indexed in the file system image, so that they are invisible. This record-level logging way compacts the writes on I/O synchronization, and mitigates the performance and endurance penalty from synchronous I/Os (Section 3.2).

Since the durability of dirty pages is provided in the staging phase, these dirty pages can be buffered in main memory. With accumulated data pages in memory, they can be separated into groups with similar hotness more efficiently. Also, storage space in a file system image is lazily allocated for the buffered pages until the patching phase. Lazy grouping and allocation in the patching phase improve data layout and thereby lead to better garbage collection performance (Section 3.3).

Due to the decoupled content and structure updates, consistency of the staging and patching phases can be maintained independently. Updates to files or directories are logically persisted in the staging phase. Only when they are written back to the file system image in the patching phase, space is allocated followed by updated allocation metadata. Content consistency and structure consistency are separately maintained in each phase (Section 3.4).

3.2 Staging: Logical Logging

The staging phase is designed to provide efficient persistence to file systems, i.e., to efficiently write file system updates durably to flash memory. The goal includes both high write performance and low write amplification.

Logical Logging. The staging phase provides data durability to the content updates, including updates to file pages, directory entry pages, and index nodes. Instead of marking a page dirty, StageFS tracks the dirty bytes of each page. It records the write location of each write request in main memory, including the offset and length in each page, which the request updates. When an I/O synchronization is required, StageFS iterates all dirty files in the file system. For each dirty file, its dirty pages are performed using either *full-page steal* or *record-level logging*, according to their dirty granularity, hotness, and so on.

For a *full-page steal* write, StageFS steals a page from the hidden area in the file system image and writes dirty pages in full pages. The hidden area is invisible to the file system image. Pages in this area are not indexed in the file system image. The mappings from the logical addresses to the physical addresses are recorded in the staging area using a *mapping* record. In the patching phase, these mappings are updated to the file system image to make these updates visible.

For a *record-level logging* write, StageFS compacts these dirty parts and the mappings of full-page steal writes to the staging area. Figure 2 illustrates the records in the record-level logging.

For each synchronization, a file has its inode and data records logged continuously, as well as the mapping records that store the mapping of full-page writes in the hidden area. When the log is written back to flash memory physically, the log is written in page units, and the free space in the tail page is a padding record. The inode record keeps the inode of the file, while the data record keeps a dirty data extent. Each inode or data record has a logical ID for identification. A logical ID is a tuple of $<type, ino, off, len>$. The first field indicates the type of the record: data, inode, mapping, or padding. The other three fields respectively represent the inode number, and the offset and length of the data in the file. File updates (including inode, dirent, and data pages) are tracked using the logical ID tuples in the page cache.

Read performance in the staging phase is not sacrificed, even though data records in the record-level logging have variable lengths, which are not good for data lookup operations. Since the staging data are mostly recently accessed, StageFS pins their memory copies in the page cache. When a page has its dirty data recorded in the log, the reference of the page is increased. Only after the page is written to the file system image, the reference is decreased to release the memory page. Therefore, read operations do not fall into the staging area, as they are satisfied in the memory copies.

In the staging phase, memory cache keeps the latest version for each page. As shown in Figure 2, both page A and B are respectively updated in two synchronization intervals. In page A, the updated two parts are independent, and thus are both updated in the memory page of page A. In page B, the updated two parts have overlaps. The later one (B'') overwrites the previous one (B'). Since the memory copies of the staging data are pinned in memory and are brought to the latest version, reads can be directly performed in these memory pages without scanning and merging those variable-length records.

Staging Bypassing. The staging phase in StageFS efficiently provides durability to small random writes, but may cause inefficiency in sequential or asynchronous write workloads. StageFS calculates the efficiency of logical logging:

$$E = (Nr_{writes})/(Nr_{logging} + Nr_{patching}),$$

where Nr_{writes} is the accumulated number of page updates, $Nr_{logging}$ is the number of record-level logging pages, and $Nr_{patching}$ is the number of patching pages (i.e., the number of all pages that have dirty bytes in the staging area). When E is less than 1, i.e., directly writing is more efficient than logical logging, StageFS chooses to bypass the staging phase and falls back to the normal write flow.

3.3 Patching: Data Reorganization

Another problem of frequent I/O synchronization is that the input datasets of data grouping are too small to exploit the benefits of data grouping and allocation algorithms. StageFS uses the staging area as a persistent cache, which provides data durability to synchronous I/Os. Data allocation and grouping are lazily performed until the patching phase, so as to accumulate data in the input datasets and improve the effect of data layout optimization.

Allocation on Patching. Space allocation in the FS image for file system updates is lazily performed until the patching phase. In the staging phase, each update is appended to the staging log with only the logical ID, which indicates its offset and length in the file. In the patching phase, space allocation is performed to reorganize the data into a better layout: (1) page-level indexing that is transformed from the non-indexed record-level logging, and (2) more sequential accesses by merging and reordering random writes.

File system updates are written to the FS image in page units, and are written back using the memory copies. In the FS image, the indices are built to indexing data of page units (i.e., page-level indexing). As memory pages of the staging data are pinned in main memory, it does not need to scan and merge the variable-length records in the staging area. Therefore, file system updates are written twice: one to FS staging in record level for data durability, and the other to FS image in page level for data indexing.

To provide this functionality, StageFS differs the dirty statuses of each memory page using two dirty flags: the staging dirty flag `FLG_SG_DT` and the patching dirty flag `FLG_IG_DT`. `FLG_SG_DT` (FLaG StaGing DirTy bit) is used to indicate whether this page needs to be persisted to the stage phase, while `FLG_IG_DT` (FLaG file system ImaGe patching DirTy bit) is used to indicate whether the page needs to be updated to the file system image. When a page is updated, its staging dirty flag `FLG_SG_DT` is set. Meanwhile, the reference count of this page is increased to prevent this page from being released. When the patching phase starts, `FLG_SG_DT` is reset, and the `FLG_IG_DT` is set. The reference count is also decreased to perform the patching write. When the patching phase finishes, `FLG_IG_DT` is reset. The two flags facilitate the two-phase updates.

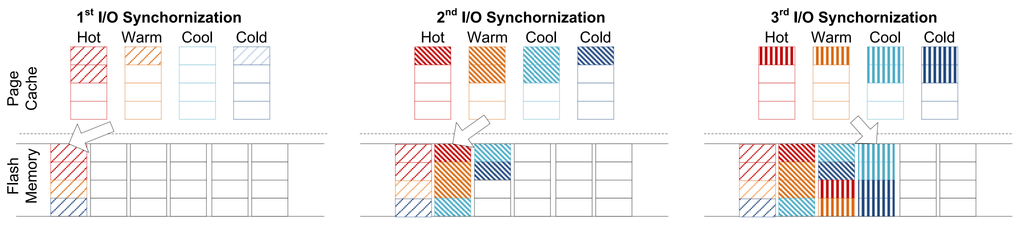
File system updates are made more sequential when written back to the FS image. StageFS allocates space sequentially for each file or data segment. During the space allocation, it first iterates all dirty files by checking the `FLG_IG_DT` flag. It then calculates the size and allocates space for all dirty data in each file. Afterward, dirty pages of each file are written consecutively in the allocated space. Since the input dataset for each data allocation is increased, these data can be written to different parallel units of flash storage, which gains benefit from the internal parallelism of flash storage. After the patching operation, StageFS has all files indexed in page units with optimized data placement.

Note that allocation on patching is different from delayed allocation in recent file systems like Ext4. StageFS allocates space during the patching phase, in which the logs have already been persisted and are written back to file system image. The delayed allocation in Ext4 delays the space allocation for writes that are buffered in the page cache, but has to allocate space before writing logs to the journal area.

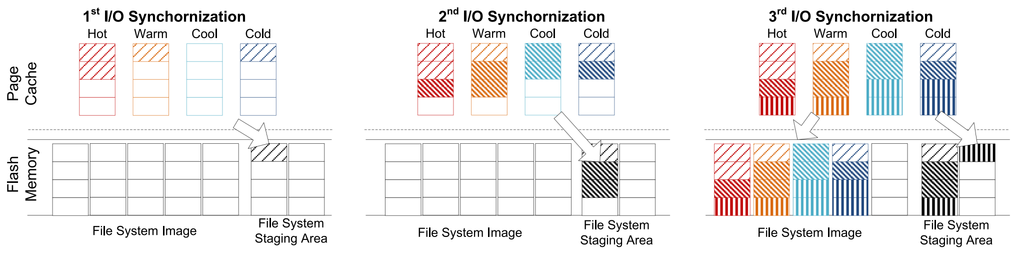
Lazy Grouping. Data grouping is also lazily performed until the patching phase. The effect of data grouping is critical for erase performance in flash storage because an inferior data grouping increases the number of valid pages that need to be moved during garbage collection. The effect of data grouping is affected by not only the goodness of algorithm but also the input dataset. Lazy grouping aims at enlarging the input dataset to improve the data grouping effect.

StageFS uses a simple data grouping algorithm that groups data with two preferences: (1) data pages that belong to a same file or file segment and/or (2) data pages that have similar access frequencies (i.e., hotness). The hotness of each page is tracked in the staging phase. StageFS records the number of accesses to each page in the staging phase and calculates the hotness of each file/segment by accumulating the hotness of its pages. And then, all files/segments are sorted in the order of hotness. Finally, they are separated into different groups and written to flash blocks in different parallel units. With more pages accumulated in the staging phase, the hotness tracking is more accurate, and the grouping effect is improved (i.e., pages in each group are closer). As such, StageFS enlarges the input datasets to improve the grouping performance.

Example. Figure 3 gives an example to show the benefits of data reorganization on patching in StageFS. In traditional file systems as shown in Figure 3(a), hot/cold data groups are forced to be written to a file system image on each I/O synchronization. Since there are not enough pages in each group, these pages are probably written to flash memory consecutively. The hot or cold pages are mixed, resulting in high overhead in garbage collection. Comparatively, StageFS provides data durability in a persistence-efficient way in the staging phase, and dirty pages stay in memory. As shown in Figure 3(b), the number of dirty pages is increased, and this enables better data grouping and improves garbage collection performance.



(a) Data Layout in Traditional File Systems



(b) Data Layout in StageFS

Fig. 3. An example of data reorganization on patching.

3.4 Consistency

In StageFS, *content consistency* and *structure consistency* are ensured separately in the staging and patching phases.

Content Consistency. In the staging area, file system updates are written in a log-structured way, in which the new version does not overwrite the old version. StageFS makes uses of this feature and makes each synchronized write as a transaction. This is similar to the journaling mechanism except that the writes are performed in record level. Different from journaling, StageFS uses the padding record as the commit record. The commit record is used to indicate the end of a transaction. In most cases, the tail page has a padding record. For the cases in which a file system writes exactly the size of the page units, a new page is allocated to be the padding record. Therefore, every synchronized write has a padding record to indicate its completeness. Though the padding record is used as the commit record, there is no ordering between data/inode record writes and the padding record write. This is because an unwritten page or a partially written page can be detected in flash storage. An unwritten page has all “1” s,[3] and the partially written page is detected by checking the Error Correction Code (ECC). If any page in one transaction is not written, the transaction is not committed.

During recovery after unexpected failures, content updates in the staging area need to be merged with corresponding pages in the file system image. StageFS reads the updates of files or directories in the staging area, and marks their inode pages in `icache` (i.e., inode cache) as obsolete by setting their obsolete bits. Instead of performing merging operations immediately, StageFS delays the merge operation to the succeeding I/O accesses. Therefore, I/O operations during recovery need to check the obsolete bit in `icache` before performing read or write operations. If the obsolete bit is set, data pages in the file system image are read to the `page cache` followed by the updates from the staging area.

Structure Consistency. In the patching phase, bitmaps in persistent storage need to correctly indicate the allocation statuses of each page. To achieve this, StageFS pre-allocates all space that

[3]A page that contains all valid “1” s is remapped to a virtual all-one page, without being written physically.

is needed in the current patching phase when starting the patching operation. It then writes the bitmap changes to the tail of the staging logging. Only after the bitmap changes are persistently written, the patching writes are performed. If the system fails during patching, bitmap changes are read to check the write statuses of the staging data. If the patching fails, StageFS marks all corresponding pages of the bitmap changes as invalid, and then restarts the patching phase by allocating space and writing the staging data to the FS image.

In conclusion, both content and structure consistency can be easily and efficiently ensured leveraging the log-structured staging design in StageFS.

3.5 Discussion

StageFS is designed from open-channel SSD, but is also partially applicable to commercial SSDs. Also, the built-in staging phase can significantly benefit from byte-addressable non-volatile memories.

Commercial SSD Cases. In commercial SSDs, the FTL design is a black box to file systems. This makes it difficult for file systems to efficiently place data to flash memory. Data groups may be broken, and distributed to different channels of flash memory [52]. Thus, for commercial SSDs, the staging phase in StageFS is still effective, but the effectiveness of the patching phase depends.

Non-Volatile Memory Cases. Although the staging layer is designed to be a built-in part in StageFS, it can also be deployed in byte-addressable non-volatile memory (NVM) to boost performance. There are two cases for the NVM usage. One is to use NVM as disk cache. Since most disk cache is high-utilized and is transparent to software, it is not feasible to configure the staging layer to it. The other is use NVM as persistent memory, in which NVM is directly attached to the memory bus, either as the persistent buffer to improve performance [13, 21] or the journaling to ensure consistency [27]. However, these memory-level storage extensions have drawbacks. First, it is difficult to maintain the data management in both memory and storage levels. Since most storage devices support a hot-plugging feature and memory devices do not, hardware changes lead to incorrect file system state. Second, the memory-level data may get inconsistency when the volatile data in the CPU cache gets lost after unexpected failures [15, 33, 34, 44, 48]. As such, we leave the usage of non-volatile memory to StageFS as future research.

4 IMPLEMENTATION

We implement StageFS based on the F2FS file system in the Linux kernel, with a line of code (LOC) of 1125. The file system image of StageFS borrows the on-disk data layout of F2FS [26]. We modify the persistence flow of write operations by introducing the staging phase, and thereby change the read and write flows in the file system.

Fine-grained Dirty Tracking. StageFS decouples the content and the structure of a file system. It provides durability to the content in the staging phase, while delaying the structure updates to the patching phase. In the staging phase, StageFS tracks the updated (dirty) parts of each file in main memory, including file data pages, directory entry pages, and inode pages in the page cache. StageFS tracks these writes for each file indexed in a hash table, and uses an ordered linked list to keep the tuples for each page. The tuple <*page_no*, *off*, *len*> describes the location of each dirty part of a page. For a page write, a new tuple is generated and inserted. If the write has overlaps with previous writes, the tuple is merged with corresponding tuples. The number of list nodes in the linked list is limited to eight in the current implementation. When there are more than eight nodes, the whole page is set to dirty as a full-page write.

When a synchronization is called, StageFS scans the hash table to locate the dirty pages. Partial pages are written to the staging area in the record-level logging way, and full pages are written to the hidden area of the file system image, without updating the block pointers or space allocation metadata in the file system image.

Fine-grained dirty tracking is not supported in mmap pages currently in StageFS. This is because, when mmap pages are mapped to user space, the kernel file system is not able to know the dirty parts inside each page. However, when mmap is needed, StageFS can fall back to normal file systems by taking every page as a full-page read/write.

Double Writeback Control. Updates to file system image are delayed to the patching phase, while they are first written to the staging area for durability. Memory pages are written twice, respectively in the staging and patching phase. To distinguish the two writeback operations, StageFS uses two dirty flags as discussed in Section 3.3. In the staging phase, the staging dirty flag *FLG_SG_DT* is set without setting the patching dirty flag. Meanwhile, the reference count of this page is increased to prevent this page from being released. Until the patching phase, the patching dirty flag *FLG_IG_DT* is set and the reference count is decreased to perform the patching write. When these dirty pages are written to the file system image, the structure metadata (e.g., block pointers, space allocation metadata) are updated.

Crash Recovery. After unexpected crashes, content updates in the staging phase need to be merged to corresponding pages in the file system image. StageFS reads the updates of files or directories in the staging area, and marks their inode page in *icache* as obsolete. Instead of performing merging operations immediately, StageFS delays the merge operation to the following I/O accesses. During recovery, I/O operations need to first check the obsolete bit in inode cache before performing read or write operations. If the obsolete bit is set, data pages in the file system image are read to the *page cache* followed by the updates from the staging area.

5 EVALUATION

In this section, we first compare StageFS with Ext4, BtrFS, and F2FS file systems to evaluate its overall performance, using both open-channel and commercial SSDs, and single-operation performance. We then evaluate the garbage collection performance under each file system, and measure the recovery overhead. Finally, we show StageFS's sensitivity to the staging size and the internal parallelism of an SSD.

5.1 Experimental Setup

We customized the Peripheral Component Interconnect express (PCIe) open-channel SSDs from Huawei in 2013, following the requirements in Object Flash Storage System (OFSS) [36] to directly manage flash memory with physical addresses. These PCIe open-channel SSDs bypass the FTL inside the device, and export the physical addresses to the host server with a proprietary device driver in the host. Because this device driver only supports Linux Kernel 2.6.32, our implementations and evaluations have to be performed in this kernel version.[4]

5.1.1 Open-Channel SSD. In the evaluation, we ported F2FS to the open-channel SSD, and ran F2FS and StageFS directly on the open-channel SSD. To run legacy Ext4 and BtrFS, we implemented a host-based page-level FTL (PFTL) in the generic block device layer, and ran Ext4 and BtrFS on PFTL. The evaluation framework is shown in Figure 4.

The Open-Channel SSD Device. We customize the open-channel SSD device, which exports the physical addresses to the software and support read/write/erase operations directly to physical addresses. The device has neither FTL nor embedded RAM inside. It is connected to the host computer using a PCIe interface, and exposes the physical layout to the operating system. The

[4]Unfortunately, we have not been able to continue this collaboration to update the open-channel SSD and drivers. But we believe open-channel SSDs will be available in the market soon, as open-channel SSDs are now being adopted in industry.

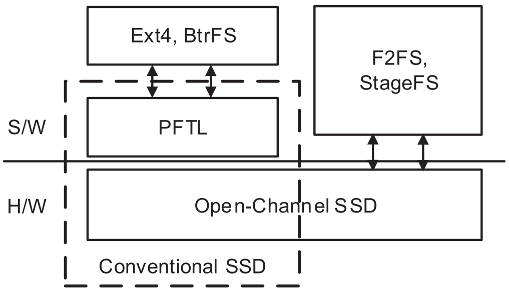


Fig. 4. Evaluation framework.

Table 1. Single-Unit Performance of Open-Channel SSD

Performance	Latency (8KB)	Bandwidth
Read	52.3us	49.84MB/s
Write	544.9us	6.55MB/s

OS directly issues read, write, and erase commands to the device. In the device, the size of a flash page is 8KB, and the size of a flash block is 2MB. We use 16 parallel units (or channels) in the flash device, with a total size of 128GB. Measured performance parameters of a single flash unit in the device (bypassing cache) are listed in Table 1. The OS can exploit high device bandwidth by distributing requests to the 16 parallel units.

Flash Translation Layer (FTL). To manage the raw flash device, we implement a PFTL in the generic block layer. PFTL borrows the mapping cache idea from DFTL [19], which leverages data locality to cache a small set of mapping entries. PFTL allocates 64MB memory for the mapping cache. PFTL also borrows the lazy indexing idea from OFSS [36], which delays the persistence of the mapping table by keeping the inverted index in page metadata of each flash page. The two ideas significantly reduce the memory usage and mapping persistence overhead of page-level FTL. Since all file systems are evaluated on the same FTL, the FTL overhead contributes the same to different file system evaluations.

In PFTL, the over-provision space is set to 10% of the total space. PFTL starts garbage collection when the free space drops below 10% of the total space. PFTL uses a dynamic wear leveling, which allocates free blocks when the erase count is below the predefined threshold. PFTL allocates 64MB space for read/write cache. It supports *trim* and *flush* commands. Experiments are all performed on the customized Open-Channel SSD, in order to (1) collect the FTL internal statistics for evaluation accuracy, and (2) eliminate the impact of complex and black-boxed mechanisms (e.g., read/write buffer, internal parallelism policies) in the FTL.

5.1.2 Workloads. In the evaluation, we use both enterprise server workloads and mobile workloads. Table 2 gives the read-write (R/W) ratios and the descriptions of all evaluated workloads. We evaluate enterprise server workloads using fileserver, varmail, webserver, and webproxy workloads in FileBench [3]. We also evaluate mobile workloads using the Mobibench benchmark. Due to the widely usage of SQLite in Android mobile phones, we also evaluate synthetic database workloads using DB_Bench [6] on SQLite 3.8.6 [8], in which we use workloads of asynchronous sequential write (*seq-async*), synchronous sequential write (*seq-sync*), asynchronous random write (*rand-async*), and synchronous random write (*rand-sync*). In the evaluation, the filebench workloads use default settings and run 10 minutes. The BD_Bench workloads perform 1 million write

Table 2. Evaluated Workloads

	Workloads	R/W Ratio	Description
server workloads	*fileserver*	1:2	file server workload: create, delete, append, read, write, and get attribute
	varmail	1:1	mail server workload: create-append-sync, read-append-sync, read, and delete
	webserver	10:1	web server workload: open-read-close, log append
	webproxy	5:1	web proxy workload: create-write-close, open-read-close, delete, log append
mobile workloads	*mobibench*	1:99	benchmark for mobile phones: random asynchronous writes to pre-allocated files
	seq-async	0:1	SQLite database workload: sequential write, asynchronous
	seq-sync	0:1	SQLite database workload: sequential write, synchronous
	rand-async	0:1	SQLite database workload: random write, asynchronous
	rand-sync	0:1	SQLite database workload: random write, synchronous

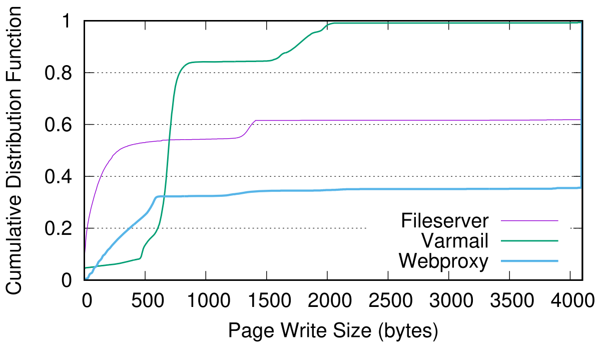


Fig. 5. Cumulative distribution function (CDF): the cumulative frequency of in-page write sizes [32].

operations for async workloads and 10,000 writes for sync workloads. Value size of each operation is 100B, which is the default size in the benchmark.

Figure 5 also shows the frequency distribution of write sizes inside each page for write-intensive server workloads. Over 80% of writes in varmail have write sizes smaller than 1KB. Fileserver and webproxy have bi-mode write size patterns, either with very small write sizes (smaller than 512B) or with full-page write size (4KB). Over 50% of writes in fileserver and 30% of writes in webproxy have write sizes smaller than 512B. For mobile workloads, all write sizes are very small, with the value size set to 100B in our experiments. But note that the buffered I/O mitigates this problem in the page cache if the synchronization is not frequent. Take the seq-async for example; these small writes can be buffered in the page cache. When they are written back to flash memory, most of them are full page writes. Therefore, these small partial-page writes provide optimization opportunities for StageFS when frequent synchronizations are required in the workload.

StageFS is evaluated against several widely-used file systems: Ext4 [2], Btrfs [1], and F2FS [26]. Ext4 is the successor of the well-known Ext2 file system with enhanced features like journaling and delayed allocation. Btrfs is a recently developed copy-on-write (COW) file system, and F2FS is a log-structured file system developed for flash storage. All file systems are mounted with default options. The staging size in StageFS is set to 128MB by default.

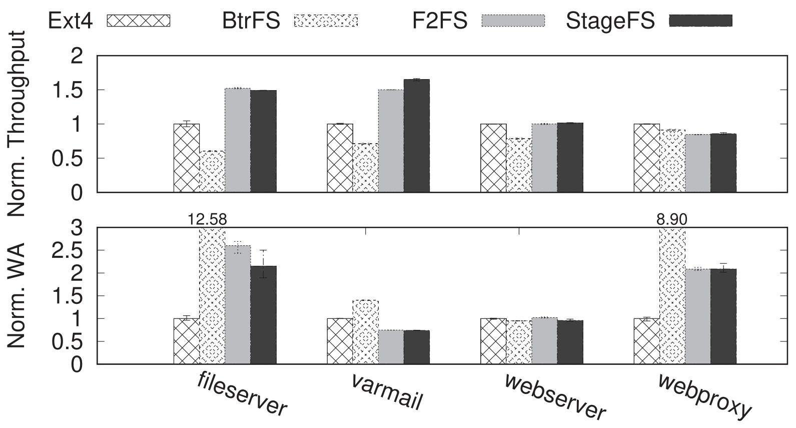


Fig. 6. Performance using enterprise server workloads.

In the evaluation, experiments are conducted on servers with Red Hat operating system at Linux kernel 2.6.32. Each server is equipped with a 2.10GHz Intel Xeon E5-2620 processor and a 8GB main memory.

5.2 Overall Performance

We compare StageFS with aforementioned file systems by measuring the I/O throughput and the write amplification, respectively to evaluate the performance and endurance effect. In this evaluation, we allocate 64GB flash memory space and erase all flash blocks before each experiment, so as to limit the impact from garbage collection. We perform this experiment using both enterprise server workloads and mobile workloads.

In the customized open-channel SSD, we are able to collect the internal metrics in the FTL. To evaluate the write amplification of the whole system, we collect both the total write size of applications (denoted as W_{app}), which is the data traffic that a file system receives, and the total write size of PFTL (denoted as W_{ftl}), which is the data traffic that finally goes to flash memory. The write amplification value is calculated by dividing W_{ftl} using W_{app}. Thus, in the experiments except the commercial SSD evaluation part, we use measure the write amplification using:

$$WA = \frac{W_{ftl}}{W_{app}}$$

Evaluations with Enterprise Server Workloads. Figure 6 shows both normalized throughput (the top half) and normalized write amplification (the bottom half) of all evaluated file systems for server workloads. In the figure, both the throughput and the write amplification value of each file system is normalized to Ext4.

As shown in the top half of Figure 6, we can see that StageFS achieves a better or comparable performance compared to all the other evaluated file systems in terms of I/O throughput. Among these server workloads, varmail is the one that has frequent synchronizations. StageFS outperforms Ext4 by 65.2% and F2FS by 15.2%. As shown in the results, F2FS has better performance than Ext4 in general, while BtrFS has poorer performance. BtrFS writes in a copy-on-write way, which increases write amplification (as shown in the bottom half of Figure 6). In contrary, F2FS is a flash-optimized file system. It mitigates the wandering tree problem by introducing the NAT (Node Address Table) structure. This reduces the write amplification for index updates. StageFS further improves the persistence performance using logical logging, for partial page writes or page writes

Table 3. Throughput Using Server Workloads

Workloads	Average Performance (K IOPS)			
	Ext4	BtrFS	F2FS	StageFS
fileserver	208.07	125.40	316.83	309.97
varmail	6.13	4.37	9.2	10.13
webserver	94.80	74.40	94.87	96.20
webproxy	46.17	42.07	39.03	39.60

Table 4. Write Traffic (GB)

Workloads	Write Traffic from Applications				Write Traffic to Flash Memory			
	Ext4	BtrFS	F2FS	StageFS	Ext4	BtrFS	F2FS	StageFS
fileserver	62.49	39.47	100.37	98.48	4.74	35.76	17.41	13.31
varmail	1.29	0.91	1.93	1.87	3.84	3.82	4.28	4.11
webserver	3.00	2.38	3.00	3.05	2.81	2.08	2.88	2.67
webproxy	4.80	4.41	4.04	4.09	0.42	3.45	0.73	0.71

Notes: Results are from the second run of each test.

Table 5. Throughput and Write Traffic Using Mobile Workloads

Workloads	Average Throughput (IOPS)				Write Traffic to Flash Memory (MB)			
	Ext4	BtrFS	F2FS	StageFS	Ext4	BtrFS	F2FS	StageFS
seq-async	61.76	23.39	57.33	60.02	310.69	319.66	311.61	328.73
seq-sync	0.18	0.04	0.33	0.68	403.15	271.66	220.89	114.65
rand-async	38.09	11.47	39.55	39.88	266.36	1576.62	267.04	503.71
rand-sync	0.17	0.04	0.32	0.68	419.69	286.67	237.58	112.81

with good temporal locality. As a result, StageFS achieves comparable or better performance than F2FS. Table 3 also shows the absolute numbers of I/O performance for these workloads.

As shown in the bottom half of Figure 6, StageFS has low write amplification for the varmail workload, which has frequent `fsync` operations. For varmail, the write amplification in StageFS is 74.1% of that in Ext4. Table 4 shows both the write traffic W_{app} from the benchmark applications (which are sent to the file system), and the write traffic W_{ftl} (from the FTL) to the flash memory. Note that, it is not fair to directly compare values of the write traffic to flash memory W_{ftl}. Because filebench is set to run a fixed time, the write traffic for each file system is different for different file systems, according to the execution speed of each file system. Instead, we can divide the right W_{app} using the left W_{ftl} value for each workload to compare the write amplification of each file system. For the bottom half of Figure 6 and Table 4, we can also see that varmail has write amplification larger than 1, while the other workloads have write amplification smaller than 1. This is because, with less synchronization operations, the buffered I/O is effective in reducing the write traffic. In these workloads, SSD writes do not directly slow down the application executions. This is also the reason why StageFS achieves much better performance in varmail, while only comparable performance in other workloads.

Evaluations with Mobile Workloads. Figure 7 shows the normalized results of mobile workloads, and Table 5 gives the absolute throughput and write traffic numbers. StageFS significantly improves performance for the mobile workloads, as mobile applications have frequent synchronization and lead to frequent writes in SQLite [23, 25]. StageFS outperforms Ext4 by 390.5%, 286.3%,

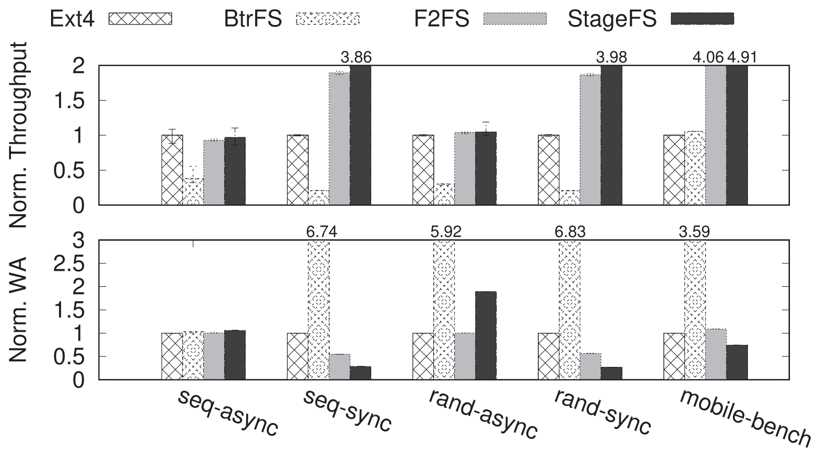


Fig. 7. Performance using mobile workloads.

and 298.0%, and outperforms F2FS by 20.9%, 197.1%, and 211.4%, respectively, for mobibench, seq-sync, and rand-sync workloads.

For workloads with frequent synchronizations, StageFS has the lowest write amplification. StageFS has write amplification that is only 74.3%, 28.4%, and 26.9% of that in Ext4, respectively, for mobibench, seq-sync, and rand-sync workloads. StageFS effectively improves flash endurance for sync-intensive workloads. StageFS provides efficient persistence, which reduces write size and thus the write latency of synchronous I/Os. Since synchronous I/Os block other I/Os, the persistence performance improvement leads to noticeable performance improvement of the application.

In Table 5, we can see that in synchronous workloads (seq-sync and rand-sync), the write traffic in StageFS is more significantly reduced than in other file systems. For asynchronous workloads, StageFS detects the sequential pattern and bypasses the staging phase in seq-async. It achieves almost the same write traffic as others. However, StageFS has higher write amplification than F2FS. This is because, in rand-async, updates are spanned to different pages, and StageFS first writes the dirty parts in the staging phase and then writes those pages back in the patching phase. In rand-async, the staging phase does not help but instead leads to higher write traffic. StageFS is not able to detect this pattern or bypass the staging phase. Fortunately, the performance is not affected. This is because the patching phase is performed in the background and does not affect the performance in rand-async workload.

We conclude that the staging phase of StageFS is effective in improving persistence performance and lowering the write amplification for sync-intensive workloads.

5.3 Evaluation on Commercial SSDs.

The staging phase optimizations of StageFS are also applicable to commercial SSDs, while optimizations in the patching phases are not clear whether they are effective due to black-box FTL designs in commercial SSDs. To understand the performance of StageFS on commercial SSDs, we also compare it with other file systems on a 120GB Intel 530 SSD.

Figure 8 shows the normalized throughput of all evaluated file systems for each workload on the commercial SSD. The values are normalized to that of Ext4 in the figure. Commercial SSD evaluation has similar results with open-channel SSD evaluation. StageFS outperforms Ext4 by 142.6%, 103.1%, and 104.1%, respectively for varmail, seq-sync and rand-sync workloads. For the three workloads, StageFS also outperforms F2FS by 99.6%, 10.7%, and 10.9%. Performance gain

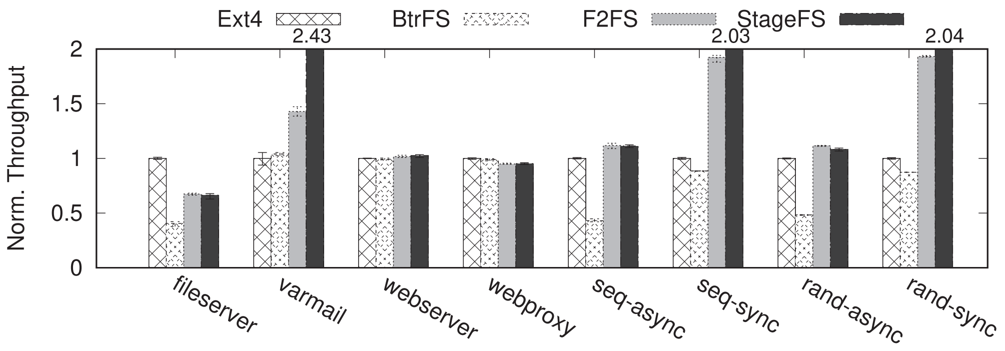


Fig. 8. Evaluation with commercial SSDs.

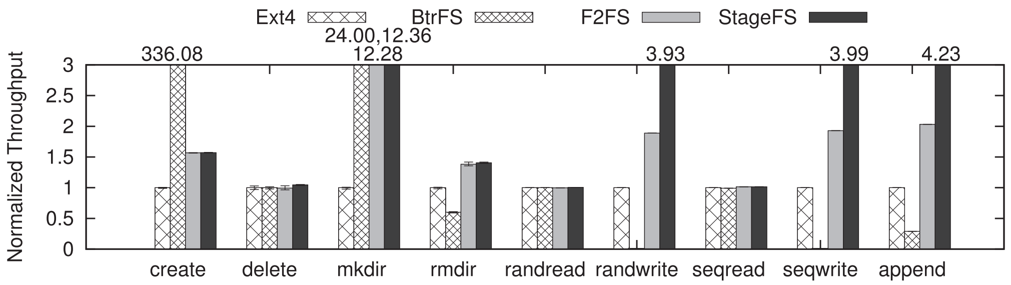


Fig. 9. Single-operation performance evaluation.

differences between open-channel and commercial SSDs are because of the FTL implementations, e.g., internal read/write buffer management.

5.4 Single-Operation Performance

To understand the performance of common file system operations, we compare StageFS with aforementioned file systems using a microbenchmark. We write the microbenchmark following the settings in Refs. [35] and [45]. The file creation and deletion benchmarks create or delete 100K files spread over 100 directories. The mkdir and rmdir benchmarks create and remove 10K directories spread over 10 directories. The randread, randwrite, seqread, seqwrite, and append benchmarks respectively perform 512B value access operations. *fsync* is performed following each operation.

Figure 9 shows the normalized throughput of all evaluated file systems for each file system operation benchmark. The throughput of each evaluated file system is normalized to that of Ext4. We have two observations from the figure:

(1) StageFS has comparable or better performance than F2FS, while F2FS outperforms other traditional file systems for most file system operations. The exception is that BtrFS has extremely high performance in create and mkdir operations. This is because BtrFS releases the inode mutex before directory metadata gets persisted and does not block the succeed operations to the directory. Besides the two operations, StageFS is among the best.
(2) StageFS outperforms F2FS by 204.1%, 206.3%, and 219.7%, respectively, for randwrite, seqwrite, and append benchmarks. This is because existing file system updates multiple pages, including the index page and the data page, even for a 512B update. In contrast, StageFS performs logical logging, which only writes the 512B data as well as its logical ID to the persistent staging area. Multiple records in the logical logging share the same page

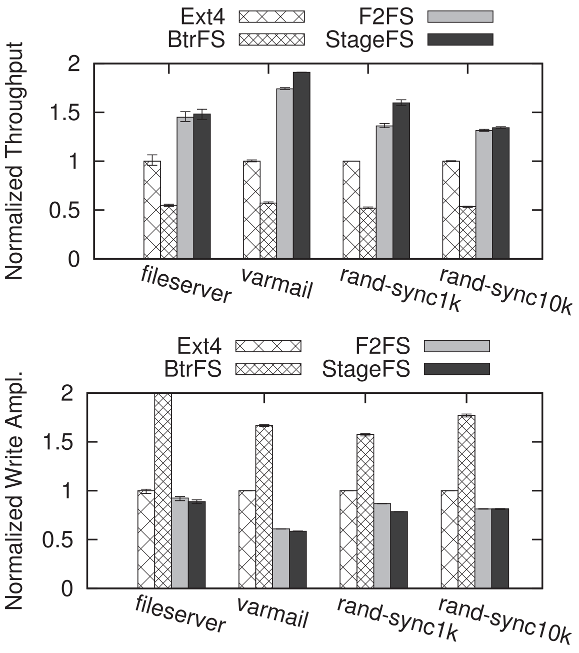


Fig. 10. Garbage collection performance.

(e.g., 4KB), so that the write amplification can be reduced. Thus, StageFS benefits from the efficient persistence in the staging phase.

5.5 Garbage Collection Performance

To understand the StageFS effect on garbage collection performance, we collect garbage collection statistics, including the total number of both erased blocks and moved pages in the FTL. In this evaluation, we limit the flash memory space to 8GB and enlarge the workload size. The fileserver workload uses three times the default number of files. The varmail workload uses 10 times the default number of files and doubles the runtime. The rand-sync workload has two configurations. One (denoted as *rand-sync1k*) performs 2 million 100B random writes on 40 million records, and there is one synchronization for every 1,000 writes; The other (denoted as *rand-sync10k*) performs 4 million 100B random writes on 20 million records, and there is one synchronization for every 10,000 writes.

Performance with Heavy Garbage Collection. Figure 10 shows both the normalized throughput (the top half) and the normalized write amplification (the bottom half) of each workload on different file systems, to evaluate the garbage collection performance in the FTL. For evaluations with garbage collection, StageFS outperforms F2FS ranging from 3.0% to 23.5% for the four evaluated write-intensive workloads. And the write amplification reduction ranges from 0.1% to 8.2%.

Table 6 gives the end-to-end write traffic numbers, including both the write traffic from applications and the write traffic to flash memory. In this experiment, the flash memory space is limited to 8GB. In PFTL, garbage collection is triggered when the free space drops below 10%, which is 7.2GB in this experiment. Since all tests have write traffic to flash memory that is higher than 8GB, garbage collection are triggered in all cases. The internal statistics of garbage collection are presented in the next part.

Table 6. Write Traffic (GB)

Workloads	Write Traffic from Applications				Write Traffic to Flash Memory			
	Ext4	BtrFS	F2FS	StageFS	Ext4	BtrFS	F2FS	StageFS
fileserver	9.92	5.61	14.22	15.46	9.38	10.30	12.47	12.55
varmail	2.15	1.26	3.73	4.12	7.86	7.64	8.33	8.85
rand-sync1k	10	10	10	10	12.92	20.45	11.22	10.15
rand-sync10k	10	10	10	10	26.77	47.44	21.70	21.61

Note that the flash memory space is limited to 8GB in the garbage collection experiments.

Table 7. Erase Overhead in StageFS

Workloads	Tot. # of Erased Blocks				Tot. # of Moved Pages			
	Ext4	BtrFS	F2FS	StageFS	Ext4	BtrFS	F2FS	StageFS
fileserver	2,429	2,356	3,522	3,421	126,088	203,191	197,019	145,001
varmail	1,533	1,300	1,733	1,700	2,146	70	0	0
rand-sync1k	3,289	6,422	1,791	1,400	32,613	124,697	52,325	262
rand-sync10k	9,985	19,485	6,798	6,850	33,524	860,845	220,134	122,329

(a) Total Number of Erased Blocks and Moved Pages

Workloads	Avg. # of Moved Pages			
	Ext4	BtrFS	F2FS	StageFS
fileserver	51.89	85.47	56.00	42.46
varmail	1.39	0.05	0	0
rand-sync1k	9.92	19.40	29.20	0.19
rand-sync10k	3.36	44.17	32.38	17.84

(b) Average Number of Moved Pages per Block

Garbage Collection Statistics. Table 7 gives the collected garbage collection statistics for four evaluated workloads. Among the four file systems, even Ext4 shows relatively low garbage collection overhead; it has low I/O throughput. Ext4 is an in-place update file system. It overwrites data in the logical address space, which effectively tells FTL the invalid pages. Comparatively, the other three file systems are out-of-place update file systems. They have better performance, but higher garbage collection overhead in some cases. In the three file systems, StageFS has the lowest garbage collection overhead in terms of the numbers of both erased blocks or moved pages. By comparing the number of moved pages per block in F2FS and StageFS, we can observe that the number of moved pages per block in StageFS is smaller than F2FS. While the staging phase contributes more in the reduction of total number of erased blocks, the patching phase, including the lazy allocation and hot/cold grouping, helps more in the average number of moved pages per block. The garbage collection statistics show the benefits of the proposed techniques in StageFS.

We conclude that data reorganization, including lazy allocation and hot/cold grouping, in the patching phase of StageFS improves garbage collection performance.

5.6 Recovery Overhead

Recovery overhead includes two parts: the stage scan time and the degraded I/O time. As discussed in Section 3, the recovery process scans the staging data and caches them in the memory. It then checks the obsolete flag and performs merge operations for following I/O accesses, which results

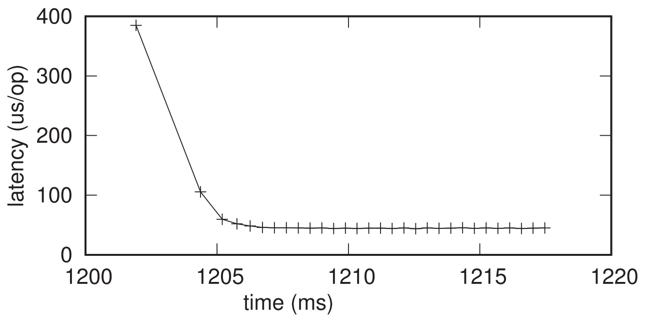


Fig. 11. Recovery overhead.

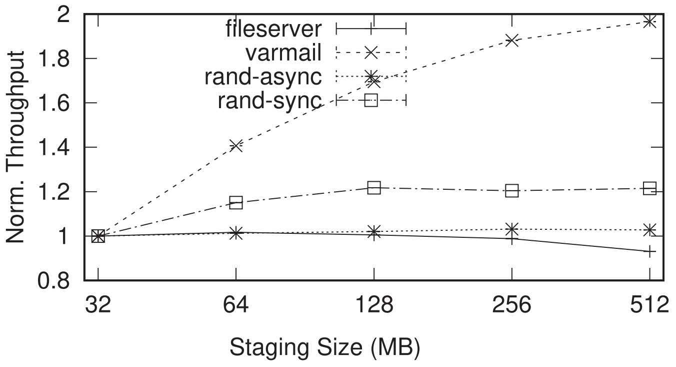


Fig. 12. Sensitivity to the staging size.

in degraded I/O performance. To evaluate the recovery overhead, we run rand-sync workload to generate the staging data, and then unmount StageFS without writing them to the file system image. After that, we remount StageFS and run rand-async workload to measure the degraded I/O performance. Figure 11 shows the degraded performance during recovery. In the evaluation, the mount operation (including the stage scan) lasts 1,200 milliseconds, and the degraded I/O period lasts 5 milliseconds. The recovery is very fast. We conclude that the recovery in StageFS is lightweight.

5.7 Sensitivity Analysis

5.7.1 Sensitivity to the Staging Size. We also evaluate StageFS's sensitivity to the staging size by varying the staging size to 32, 64, 128, 256, and 512 megabytes in this experiment.

Figure 12 plots the normalized throughputs of the fileserver, varmail, rand-async, and rand-sync workloads with different staging size settings. All throughputs are normalized to that in the 32MB setting of each workload. As observed from the figure, different workloads show different curves with changed staging sizes. The varmail workload has constantly increased performance as the staging size increases. The rand-sync workload has increased performance when the staging size is increased from 32MB to 128MB, and then keeps steady. The rand-async workload has steady performance. And the fileserver workload has a slight drop in performance as the staging size increases. While increased staging sizes can absorb more synchronous I/Os in the staging phase, the overhead including dirty tracking and memory pinning is also increased. In conclusion, different workloads demand different staging sizes. For sync-intensive workloads, a small staging size could

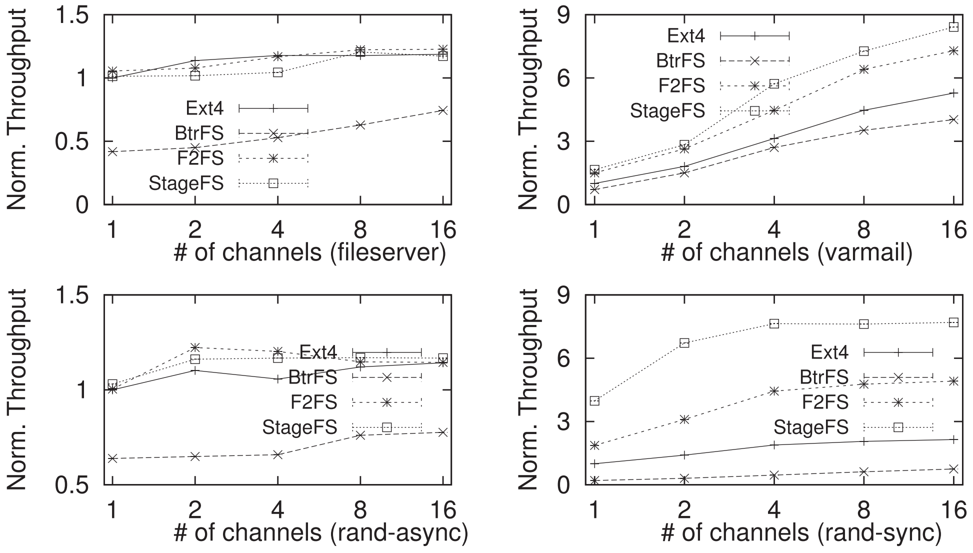


Fig. 13. Sensitivity to the internal parallelism.

bring a noticeable performance improvement. Also, the memory pinning required in the staging phase has little side effect.

5.7.2 Sensitivity to the Internal Parallelism. To study the impact from the internal parallelism, we vary the parallel units in the open-channel SSD for evaluation. In the open-channel SSD, the parallel units are connected to the NAND flash controller through different channels, and they can be accessed in parallel.

Figure 13 shows the throughput change as the number of channels increases for workloads fileserver, varmail, rand-async, and rand-sync. For the synchronization-intensive workloads like varmail and rand-sync, I/O throughput has a large increase when the number of channels is increased. In contrast, for asynchronous workloads, like fileserver and rand-async, I/O throughput has a slight increase. Thus, we conclude that the device write performance is the bottleneck for workloads with synchronous I/Os. Therefore, optimizing the persistence efficiency as in StageFS can effectively improve the performance of such workloads.

We also observe that StageFS is the performance leader for evaluations with different parallel units. While performance is improved with better internal parallelism, StageFS gains constant performance improvement.

6 RELATED WORK

Flash File Systems. General-purpose file systems have been recently proposed for flash storage. Direct File System [24] is a file system that leverages the storage management functions (e.g., block allocation, atomic write) in the FTL to manage storage space. It removes the duplicated functions in the file system and improves system performance. ReconFS [35] addresses the directory tree metadata problem on flash storage. It decouples the volatile and persistent directory trees, and proposes record-level logging for file system metadata updates. The proposed directory tree management in ReconFS improves both performance and endurance of flash storage. F2FS [26] is a file system that is designed to better exploit the benefits of flash memory. It employs flash-friendly on-disk layout and optimizes the indexing and logging schemes. F2FS has been merged into Linux kernel and shows potential performance benefits.

OFSS [36] re-architects the flash storage stack by moving the FTL to the file system (i.e., the open-channel SSD) and introduces the object storage to flash management. OFSS focuses on the endurance problem. It points out the *write amplification from file systems* problem and redesigns file system mechanisms (e.g., indexing, free space management) to reduce the write traffic inside file systems. ParaFS [52] is a recent file system designed on top of open-channel SSDs. It effectively exploits the internal parallelism of flash devices by co-designing the allocation, garbage collection, and I/O scheduling of the software and FTL layers.

In this article, our proposed StageFS is designed to address new challenges from synchronous I/Os. It introduces a persistent staging layer inside the file system to improve both the persistence efficiency and the garbage collection efficiency.

Persistence Efficiency. In flash storage, better persistence efficiency means not only better write performance but also improved endurance with reduced write traffic. To improve write performance, many file systems have tried to reduce random writes. ReiserFS [7] uses the *tail packing* scheme, to allocate the free space in the last block for small files. ReconFS [35] uses record-level logging for metadata updates of the file system directory tree. The fine-grained record-level write way reduces the write traffic. F2FS [26] uses roll-forward recovery to accelerate the fsync performance. It writes only data pages and node pages, excluding other F2FS metadata pages. These compaction or selective writing techniques reduce the write traffic and improve performance. However, they do not provide a system-wide substrate for efficient persistence of both data and metadata.

Log-structured merge tree (LSM-Tree) [40] is a write-efficient data structure that is designed for database workloads. LSM-Tree and its variants are widely used in key-value stores, such as Bigtable [11] and LevelDB [5]. And it was recently introduced to file systems. TableFS [45] organizes file system metadata using LSM-Tree and improves the metadata access performance. A variant of LSM-Tree, VT-Tree [46], is also used to reduce the merge cost for file system workload with large chunk accesses. However, for unstructured file system data, reads or writes are often of arbitrary range. It is a challenge to keep independent file ranges as value objects. While LSM-Tree normally merges data objects that have no overlap, the file system data access pattern is not friendly for LSM-tree. In addition, LSM-Tree (as in TableFS) has to perform merge operations for multiple levels, which amplifies the write traffic [30, 31]. StageFS is designed for both metadata and data, and the patching phase writes data directly from the memory (rather than reading from the log) and thus does not have the merge overhead.

BetrFS [22] introduces the write-optimized index, Fractal Tree, to file systems to improve write efficiency. Fractal tree provides both locality as in B-trees and efficient writes as in LSM-trees. While StageFS optimizes the write flows without changing the indexing, this indexing optimization in BetrFS is orthogonal to StageFS. They can be used together for further improvement.

File systems also improve performance by introducing a cache layer. NetApp WAFL [21] provides data durability using a non-volatile RAM. This delays the writes to persistent storage and improves performance. A buffer cache architecture with Unioning of the Buffer cache and Journaling layers (UBJ) [27] provides journaling using non-volatile main memory, and significantly reduces the consistency overhead. Recently, file systems that are designed for non-volatile memories, like NOn-Volatile memory Accelerated (NOVA) file system [51], HiNFS [41], and Strata [16], take similar ideas of record-level logging to reduce the write amplification caused by the page-aligned write restriction in memory. Unfortunately, these designs require byte-addressable non-volatile memories.

To improve flash endurance, file systems have started to pay attention to write traffic. File system mechanisms are redesigned leveraging flash memory characteristics to reduce the write amplification from the file systems [36]. ReconFS [35] uses record-level logging for file system directory metadata updates, which reduces write traffic from the file system metadata.

In contrast, StageFS introduces a system-wide logical logging technique by allocating a built-in persistent staging area. This system wide logging is different to LSM-Tree in that the staging log is patched to the page-aligned indexed file system image. StageFS absorbs both data and metadata of synchronous I/Os, improving both performance and endurance.

Garbage Collection Efficiency. Data grouping algorithms have also been intensively studied in flash-based storage systems to reduce garbage collection overhead. Among these works, SSD-based File System (SFS) [38] is the one that is studied in the file system level. It improves the data grouping algorithm. It groups data with similar update likelihood using an iterative segment quantization algorithm. Instead of improving the grouping algorithm itself, StageFS improves the grouping effect by enlarging the input datasets leveraging the staging phase.

7 CONCLUSION

Open-channel SSDs still suffer from I/O synchronizations in terms of both amplified write traffic and poor data layout, due to controversy layout choices between flash write and read/erase operations. Our proposed StageFS introduces a staging phase, a persistent storage layer in the file system, to absorb synchronized I/Os in flash storage. This file system architecture brings two benefits. First, it efficiently provides data durability to synchronized writes using logical logging in the staging phase, keeping the write amplification low. Second, it delays the file system image update by providing data durability in the staging phase so as to enable better data placement and grouping in a file system image and gain better garbage collection performance. With the introduced staging phase, StageFS achieves both high write efficiency and reduced garbage collection overhead for workloads with frequent synchronization.

ACKNOWLEDGMENTS

We thank Xiaojian Liao and Bohong Zhu at Storage Research Group of Tsinghua University for their help with the experiments, and Liyang Pan at Institute of Microelectronics of Tsinghua University for his feedback on usage of flash memory hardware.

REFERENCES

[1] [n.d.]. Btrfs. Retrieved from http://btrfs.wiki.kernel.org.
[2] [n.d.]. Ext4. Retrieved from https://ext4.wiki.kernel.org/.
[3] [n.d.]. Filebench benchmark. Retrieved from http://sourceforge.net/apps/mediawiki/filebench.
[4] [n.d.]. FusionIO Virtual Storage Layer. Retrieved from http://www.fusionio.com/products/vsl.
[5] [n.d.]. LevelDB, A fast and lightweight key/value database library by Google. Retrieved from http://code.google.com/p/leveldb/.
[6] [n.d.]. LevelDB Benchmarks. Retrieved from http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html.
[7] [n.d.]. ReiserFS. Retrieved from http://reiser4.wiki.kernel.org.
[8] 2014. SQLite. Retrieved from http://www.sqlite.org/.
[9] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX ATC)*. USENIX, Berkeley, CA.
[10] Matias Bjorling, Javier Gonzalez, and Bonnet Philippe. 2017. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*.
[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX, Berkeley, CA, 205–218.
[12] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, Vol. 11. USENIX, Berkeley, CA.
[13] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. 1996. The rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*. ACM, New York, NY, 74–83.

[14] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, 228–243.

[15] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 133–146.

[16] Youngjin Kwon Henrique Fingler, Tyler Hunt, Simon Peter, and Emmett Witchel Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 460–477.

[17] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. 2007. Generalized file system dependencies. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 307–320.

[18] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX, Berkeley, CA.

[19] Aayush 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, New York, NY, 229–240.

[20] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2011. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY.

[21] Dave Hitz, James Lau, and Michael Malcolm. 1994. File system design for an NFS file server appliance. In *Proceedings of 1994 USENIX Winter Technical Conference*. USENIX, Berkeley, CA.

[22] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Santa Clara, CA, 301–315. https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen.

[23] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O stack optimization for smartphones. In *Proceedings of 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. USENIX, Berkeley, CA, 309–320.

[24] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. 2010. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX, Berkeley, CA.

[25] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX, Berkeley, CA.

[26] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX, Santa Clara, CA. https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[27] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2013. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX, Berkeley, CA.

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

[29] Sang-Won Lee and Bongki Moon. 2007. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, NY, 55–66.

[30] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. LocoFS: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. ACM, New York, NY, Article 4, 12 pages. DOI: https://doi.org/10.1145/3126908.3126928

[31] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Berkeley, CA, 133–148. http://dl.acm.org/citation.cfm?id=2930583.2930594.

[32] Youyou Lu, Jiwu Shu, Jia Guo, and Peng Zhu. 2016. Supporting system consistency with differential transactions in flash-based SSDs. *IEEE Trans. Comput.* 65, 2 (2016), 627–639. DOI: https://doi.org/10.1109/TC.2015.2419664 to appear.

[33] Youyou Lu, Jiwu Shu, and Long Sun. 2015. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st Conference on Massive Storage Systems and Technologies (MSST'15)*. IEEE, 1–13.

[34] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-ordering consistency for persistent memory. In *Proceedings of the IEEE 32nd International Conference on Computer Design (ICCD'14)*. IEEE.

[35] Youyou Lu, Jiwu Shu, and Wei Wang. 2014. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX, Berkeley, CA, 75–88.

[36] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX, Berkeley, CA.

[37] Marshall K. McKusick, Gregory R. Ganger, et al. 1999. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of 1999 USENIX Annual Technical Conference (FREENIX Track'99)*. USENIX, Berkeley, CA, 1–17.

[38] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX, Berkeley, CA.

[39] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX, Berkeley, CA, 1–14.

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

[41] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A high performance file system for non-volatile main memory. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, 12.

[42] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 471–484. DOI: https://doi.org/10.1145/2541940.2541959

[43] Stan Park, Terence Kelly, and Kai Shen. 2013. Failure-atomic msync (): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. ACM, 225–238.

[44] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 265–276.

[45] Kai Ren and Garth A. Gibson. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of 2013 USENIX Annual Technical Conference (USENIX ATC)*. USENIX, Berkeley, CA, 145–156.

[46] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 17–30.

[47] Stephen C. Tweedie. 1998. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo*.

[48] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H. Campbell, et al. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 61–75.

[49] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. ACM, New York, NY, Article 16, 14 pages. DOI: https://doi.org/10.1145/2592798.2592804

[50] Guanying Wu and Xubin He. 2012. Delta-FTL: Improving SSD lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. ACM, 253–266.

[51] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 323–338.

[52] 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 2016 USENIX Annual Technical Conference (USENIX ATC'16)*.

Received May 2018; revised December 2018; accepted March 2019