
# FStream: Managing Flash Streams in the File System

Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Joo-Young Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong, *Samsung Electronics. Co., Ltd.*

https://www.usenix.org/conference/fast18/presentation/rho

This paper is included in the Proceedings of the
16th USENIX Conference on File and Storage Technologies.
February 12–15, 2018 • Oakland, CA, USA

ISBN 978-1-931971-42-3

Open access to the Proceedings of
the 16th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

# FStream: Managing Flash Streams in the File System

Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty
Joo-Young Hwang, Sangyeun Cho, Daniel DG Lee, Jaeheon Jeong
*Samsung Electronics Co., Ltd.*

## Abstract

The performance and lifespan of a solid-state drive (SSD) depend not only on the current input workload but also on its internal media fragmentation formed over time, as stale data are spread over a wide range of physical space in an SSD. The recently proposed *streams* gives a means for the host system to control how data are placed on the physical media (abstracted by a stream) and effectively reduce the media fragmentation. This work proposes *FStream*, a file system approach to taking advantage of this facility. FStream extracts streams at the file system level and avoids complex application level data mapping to streams. Experimental results show that FStream enhances the filebench performance by 5%∼35% and reduces WAF (Write Amplification Factor) by 7%∼46%. For a NoSQL database benchmark, performance is improved by up to 38% and WAF is reduced by up to 81%.

## 1 Introduction

Solid-state drives (SSDs) are rapidly replacing hard disk drives (HDDs) in enterprise data centers. SSDs maintain the traditional logical block device abstraction with the help from internal software, commonly known as flash translation layer (FTL). The FTL allows SSDs to substitute HDDs without complex modification in the block device interface of an OS.

Prior work has revealed, however, that this compatibility comes at a cost; when the underlying media is fragmented as a device is aged, the operational efficiency of the SSD deteriorates dramatically due to garbage collection overheads [11, 13]. More specifically, a user write I/O translates into an amplified amount of actual media writes [9], which shortens device lifetime and hampers performance. The ratio of the actual media writes to the user I/O is called *write amplification factor* (WAF).

A large body of prior work has been undertaken to address the write amplification problem and the SSD wear-out issue [3, 7]. To the same end, we focus on *how to take advantage of the multi-streamed SSD mechanism* [8]. This mechanism opens up a way to dictate data placement on an SSD's underlying physical media, abstracted by *streams*. In principle, if the host system perfectly maps data having the same lifetime to the same streams, an SSD's write amplification becomes one, completely eliminating the media fragmentation problem.

Prior works have revealed two strategies to leverage streams. The first strategy would map application data to disparate streams based on an understanding of the expected lifetime of those data. For example, files in different levels of a log-structured merge tree could be assigned to a separate stream. Case studies show that this strategy works well for NoSQL databases like Cassandra and RocksDB [8, 14]. Unfortunately, this application-level customization strategy requires that a system designer understand her target application's internal working fairly well, remaining a challenge to the designer. The other strategy aimed to "automate" the process of mapping write I/O operations to an SSD stream with no application changes. For example, the recently proposed AutoStream scheme assigns a stream to each write request based on estimated lifetime from past LBA access patterns [15]. However, this scheme has not been proven to work well under complex workload scenarios, particularly when the workload changes dynamically. Moreover, LBA based pattern detection is not practical when file data are updated in an out-of-place manner, as in copy-on-write and log-structured file systems. The above sketched strategies capture the two extremes in the design space—*application level customization* vs. *block level full automation*.

In this work, we take another strategy, where *we separate streams at the file system layer*. Our approach is motivated by the observation that file system metadata and journal data are short-lived and are good targets for separation from user data. Naturally, the primary component of our scheme, when applied to a journaling file sys-

tem like ext4 and xfs, is to allocate a separate stream for metadata and journal data, respectively. As a corollary component of our scheme, we also propose to separate databases redo/undo log file as a distinct stream at the file system layer. We implement our scheme, *FStream*, in Linux ext4 and xfs file systems and perform experiments using a variety of workloads and a stream-capable NVMe (NVM Express) SSD. Our experiments show that FStream robustly achieves a near-optimal WAF (close to 1) across the workloads we examined. We make the following contributions in this work.

- We provide an automated multi-streaming of different types of file system generated data with respect to their lifetime;
- We enhance the existing journaling file systems, ext4 and xfs, to use the multi-streamed SSDs with minimally invasive changes; and
- We achieve stream classification for application data using file system layer information.

The remainder of this paper is organized as follows. First, we describe the background of our study in Section 2, with respect to the problems of previous multi-stream schemes. Section 3 describes FStream and its implementation details. We show experimental results in Section 4 and conclude in Section 5.

# 2 Background

## 2.1 Write-amplification in SSDs

Flash memory has an inherent characteristic of *erase-before-program*. Write, also called "program" operation, happens at the granularity of a NAND page. A page cannot be rewritten unless it is "erased" first. In-place update is not possible due to this erase-before-write characteristic. Hence, overwrite is handled by placing the data in a new page, and invalidating the previous one. Erase operation is done in the unit of a NAND block, which is a collection of multiple NAND pages. Before erasing a NAND block, all its valid pages need to be copied out elsewhere; this is done in a process called *garbage-collection* (GC). These valid page movements cause additional writes that consume bandwidth, thereby leading to performance degradation and fluctuation. These additional writes also reduce endurance as program-erase cycles are limited for NAND blocks. One way to measure GC overheads is *write amplification factor* (WAF), which is described as the ratio of writes performed on flash memory to writes requested from the host system.

$$WAF = \frac{Amount\ of\ writes\ committed\ to\ flash}{Amount\ of\ writes\ that\ arrived\ from\ the\ host}$$

WAF may soar more often than not as an SSD experiences aging [8].

## 2.2 Multi-streamed SSD

The multi-streamed SSD [8] endeavors to keep WAF in check by focusing on the placement of data. It allows the host to pass a hint (stream ID) along with write requests. The stream ID provides a means to convey hints on lifetime of data that are getting written [5]. The multi-streamed SSD groups data having the same stream ID to be stored to the same NAND block. By avoiding mixing data of different lifetime, fragmentation is reduced inside NAND blocks. Figure 1 shows an example of how the data placement using multi-stream could reduce media fragmentation.

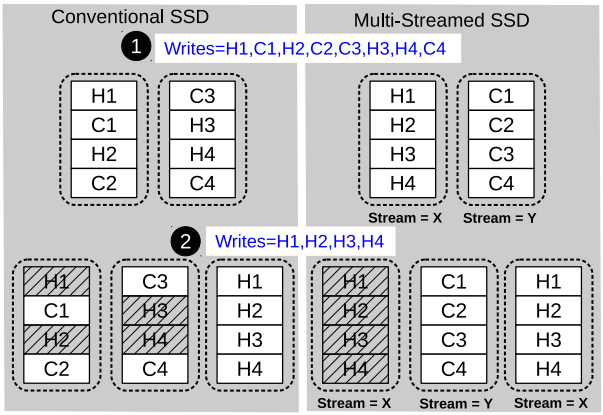



Figure 1: *Data placement comparison for two write sequences.* Here H=Hot, C=Cold, and we assume there are four pages per flash block.

Multi-stream is now a part of T10 (SCSI) standard, and under discussion in NVMe (NVM Express) working group. NVMe 1.3 specification introduces support for multi-stream in the form of "directives" [1]. An NVMe write command has the provision to carry a stream ID.

## 2.3 Leveraging Streams

While the multi-streamed SSD provides the facility of segregating data into streams, its benefit largely depends upon how well the streams are leveraged by the host. Identifying what should and should not go into the same stream is of cardinal importance for maximum benefit. Previous work [8, 14] shows benefits of application-assigned streams. This approach has the benefit of determining data lifetime accurately, but it involves modifying the source code of the target application, leading to increased deployment effort. Also when multiple applications try to assign streams, a centralized stream assignment is required to avoid conflicts. Instead of direct assignment of stream IDs, recently Linux (from 4.13 kernel) supports `fcntl()` interface to send data lifetime hints to file systems to exploit the multi-streamed SSDs [2, 6]. AutoStream [15] takes stream management to the NVMe device driver layer. It monitors requests

from file systems and estimates data lifetime. However, only limited information (e.g., request size, block addresses) is available in the driver layer, and even worse, the address-based algorithm may be ineffective under a copy-on-write file system.

Our approach, *FStream*, implements stream management intelligence at the file system layer. File systems have readily the information about file system generated data such as metadata and journal. To detect lifetime of user data, we take a simple yet efficient method which uses file's name or extension. For the sake of brevity, we do not cover the estimation of user data lifetime in detail at the file system layer.

## 3 FStream

We start by highlighting the motivation behind employing multi-stream in file systems. This is followed by overview of ext4 and xfs on-disk layout and journaling methods. Then we delve into the details of Ext4Stream and XFStream, which are stream-aware variants of ext4 and xfs, respectively.

### 3.1 Motivation

Applications can have better knowledge about the lifetime and update frequency of data that they write than file systems do. However, applications do not know about the lifetime and update frequency of file system metadata. The file system metadata usually have different update frequencies than applications' data, and are often influenced by on-disk layout and write policy of a particular file system. Typically file systems keep data and metadata logically separated, but they may not remain "physically" separated on SSDs. While carrying out file operations, metadata writes may get mixed with data writes in the same NAND block or one type of metadata may get mixed with another type of metadata. File systems equipped with stream separation capability may reduce the mixing of applications' data and file system metadata, and improve WAF and performance.

### 3.2 Ext4 metadata and journaling

The ext4 file system divides the disk-region in multiple equal-size regions called "block groups," as shown in Figure 2. Each block group contains data and their related metadata together which helps in reducing seeks for HDDs. Ext4 introduces *flex-bg* feature, which "clubs" a series of block groups whose metadata are consolidated in the first block group. Each file/directory requires an inode, which is of size 256 bytes by default. These inodes are stored in the *inode table*. *inode bitmap* and *block bitmap* are used for allocation of inodes and data blocks, respectively. *Group descriptor* contains the location of other metadata regions (inode table, block bitmap, inode bitmap) within the block group. Another type of metadata is a directory block.

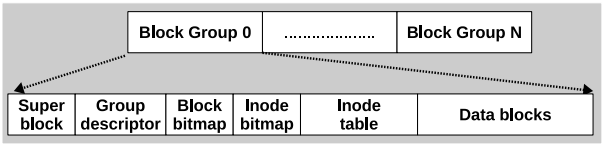



Figure 2: Ext4 on-disk layout. For simplicity, we have not shown flex-bg.

File-system consistency is achieved through write-ahead logging in journal. Journal is a special file whose blocks reside in user data area, pre-allocated at the time of file system format. Ext4 has three journaling modes; *data-writeback* (metadata journaling), *data-ordered* (metadata journaling + write data before metadata), and *data-journal* (data journaling). The default mode is data-ordered.

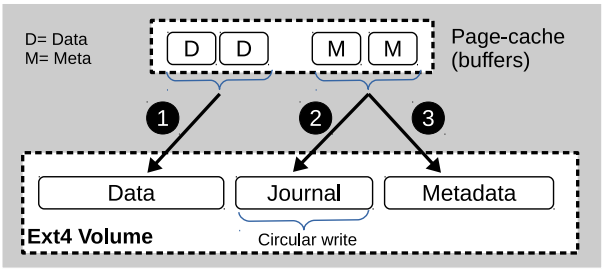



Figure 3: Ext4 *journal in ordered mode.* Ext4 writes data and journal in sequence. Metadata blocks are written to their actual home location after they are persisted to the journal.

Ext4 journals at a block granularity, i.e., even if few bytes of an inode are changed, the entire block (typically 4KiB) containing many inodes is journaled. For journaling it takes assistance from another component called journaling block device (JBD), which has its own kernel thread called jbd2. The journal area is written in a circular fashion. Figure 3 shows journaling operation in the ordered mode. During a transaction, ext4 updates metadata in in-memory buffers, and informs the jbd2 thread to commit a transaction. jbd2 maintains a timer (default 5 seconds), on expiry of which it writes modified metadata into the journal area, apart from transaction related bookkeeping data. Once changes have been made durable,the transaction is considered committed. Then, the metadata changes in memory are flushed to their original locations by write-back threads, which is called *checkpointing*.

### 3.3 Xfs metadata and journaling

Similar to ext4, xfs also divides the disk region into multiple equal-size regions called *allocation groups*, as shown in Figure 4. The primary objective of allocation

groups is to increase parallelism rather than disk locality, unlike EXT4 block groups. Each allocation group maintains its own superblock and other structures for free space management and inode allocation, thereby allowing parallel metadata operations. Free space management within an allocation group is done by using B+ trees. Inodes are allocated in chunks of 64. These chunks are managed in another B+ tree meant exclusively for inode allocation.

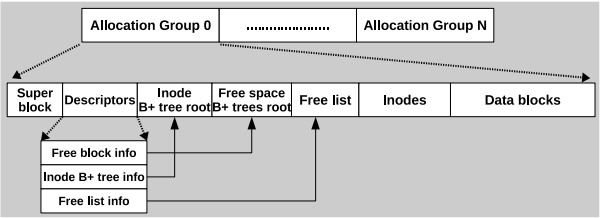



Figure 4: Xfs on-disk layout.

For transaction safety, xfs implements metadata journaling. A separate region called "log" is created during file system creation (`mkfs.xfs`). Log is written in a circular fashion as transactions are performed. Xfs maintains many log buffers (default 8) in memory, which can record the changes for multiple transactions. Default commit interval is 30 seconds. During commit, modified log buffers are written to on-disk log area. Post commit, modified metadata buffers are scheduled for flushing to their actual disk locations.

## 3.4 Ext4Stream: Multi-stream in ext4

Table 1 lists the streams we introduced in ext4. These streams can be enabled with the corresponding mount option listed in the table.

| Mount-option | Stream |
|---|---|
| journal-stream | Separate journal writes |
| inode-stream | Separate inode writes |
| dir-stream | Separate directory blocks |
| misc-stream | Separate inode/block bitmap and group descriptor |
| fname-stream | Assign distinct stream to file(s) with specific name |
| extn-stream | File-extension based stream |

Table 1: Streams introduced in Ext4Stream.

The **journal_stream** mount option is to separate journal writes. We added a `j_streamid` field in the `journal_s` structure. When ext4 is mounted with the `journal_stream` option, a stream ID is allocated and stored in the `j_streamid` field. jbd2 passes this stream ID when it writes dirty buffers and descriptor blocks in a journal area using `submit_bh` and related functions.

Ext4 makes use of buffer-head (`bh`) structures for various metadata buffers including inode, bitmaps, group descriptors and directory data blocks. We added a new field `streamid` in buffer-head to store a stream ID, and modified `submit_bh`. While forming an I/O from buffer-head, this field is also set in `bio`, taking stream ID information to a lower layer. Ext4Stream maintains stream IDs for different metadata regions in its superblock, and, depending on the type of metadata buffer-head, it sets `bh->streamid` accordingly.

The **inode_stream** mount option is to separate inode writes. Default inode size is 256 bytes, so single 4KiB FS block can store 16 inodes. Modification in one inode leads to writing of an entire 4KiB block. When Ext4Stream modifies inode buffer, it also sets `bh->streamid` with the stream ID meant for the inode stream.

The **dir_stream** mount option is to keep directory blocks into its own stream. When a new file or subdirectory is created inside a directory, a new directory entry needs to be added. This triggers either update of an existing data block belonging to the directory or addition of a new data block. Directory blocks are organized in a `htree`; leaf nodes contain directory entries and non-leaf nodes contain indexing information. We assign a same stream ID for both types of directory blocks.

The **misc_stream** is to keep inode/block bitmap blocks and group descriptor blocks into a stream. These regions receive updates during the creation/deletion of file/directory and when data blocks are allocated to file/directory. We group these regions into a single stream because they are of small size.

The **fname_stream** helps to put data of certain special files into distinct stream. Motivation is to use this for separating undo/redo log for SQL and NoSQL databases.

The **extn_stream** is to enable file extension based stream recognition. Data blocks of certain files, such as multimedia files, can be considered cold. Ext4Stream can parse extension of files during file creation. If it matches with some well-known extensions, file is assigned a different stream ID. This helps prevent hot or cold data blocks getting mixed with other types of data/metadata blocks.

## 3.5 XFStream: Multi-stream in xfs

Table 2 lists the streams we introduced to xfs. These streams can be enabled with the corresponding mount options listed in the table.

Xfs implements its own metadata buffering rather than using page cache. The `xfs_buf_t` structure is used to represent a buffer. Apart from metadata, in-memory buffers of log are implemented via `xfs_buf_t`. When the buffer is flushed, a `bio` is prepared out of `xfs_buf_t`. We

| Mount-option | Stream |
|---|---|
| log_stream | Separate writes occurring in log |
| inode_stream | Separate inode writes |
| fname_stream | Assign distinct stream to file(s) with specific name |

Table 2: Streams introduced in XFStream.

added a new field called `streamid` in `xfs_buf_t`, and used that to set the stream information in `bio`.

The **log_stream** enables XFStream to perform writes in the log area with its own stream ID. Each mounted xfs volume is represented by a `xfs_mount_t` structure. We added the field `log_streamid` in it, which is set when xfs is mounted with `log_stream`. This field is used to convey stream information in `xfs_buf_t` representing the log buffer.

The **inode_stream** mount option enables XFStream to separate inode writes into a stream. A new field `ino_streamid` kept in `xfs_mount_t` is set to stream ID meant for inodes. This field is used to convey stream information in `xfs_buf_t` representing the inode buffer.

Finally, the **fname_stream** enables XFStream to assign a distinct stream to file(s) with specific name(s).

## 4 Evaluation

Our experimental system configurations are as follows.

- System: Dell Poweredge R720 server with 32 cores and 32GB memory,
- OS: Linux kernel 4.5 with io-streamid support,
- SSD: Samsung PM963 480GB, with the allocation granularity [1] of 1.1GB,
- Benchmarks: filebench [12], YCSB (Yahoo! Cloud Serving Benchmark) 0.1.4 [4] on Cassandra 1.2.10 [10].

The SSD we used supports up to 9 streams; eight NVMe standard compliant streams and one default stream. If a write command does not specify its stream ID, it is written to the default stream.

We conduct experiments in two parts; the first part is to measure the benefit of separating file system metadata and journal. Each test starts with a fresh state involving device format and file system format. To reduce variance between the runs, we disable lazy journal and inode table initialization at the time of ext4 format. As a warming workload for filebench, we write a single file sequentially to fill 80% of logical device capacity, to ensure that 80% of the logical space stays valid throughout the test. Remaining logical space involves the actual experiment. The varmail and fileserver workloads included in the filebench are used to simulate mail server and file server workloads, respectively. The number of files in both workloads is set to 900,000; default values are used for other parameters. Each filebench workload is run for two hours with 14GB of files, performing deletion, creation, append, sync (only for varmail), and random read. Since the size of the workloads is smaller than that of RAM, vast majority of the operations that actually reach the device are likely to be write operations. In order to acquire WAF, we retrieve the number of NAND writes and host writes from FTL, and divide the former by the latter.

In the second part, we measure the benefits in data-intensive workloads by applying automatic stream assignment on certain application specific files. Previous work [8] has reported improvement by modifying Cassandra source to categorize writes into multiple streams. FStream assigns distinct stream to Cassandra *Commitlog* through fname_stream facility. Load phase involves loading 120,000,000 keys, followed by insertion of 80,000,000 keys during run phase.

[1] A multi-streamed SSD allocate and expand stream in the unit of allocation granularity

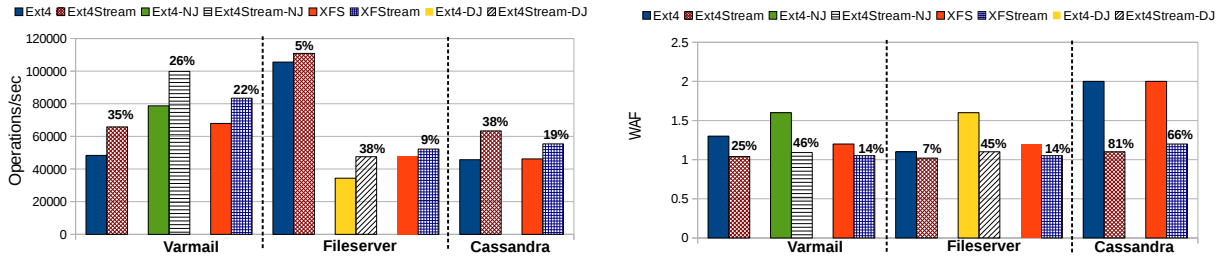
### 4.1 Filebench results

| Category | varmail | | | fileserver | |
|---|---|---|---|---|---|
| | ext4 | ext4 -nj | xfs | ext4 | xfs |
| Journal | 61% | - | 60% | 26% | 16% |
| Inode | 8% | 21% | 9% | 16% | 32% |
| Directory | 4% | 15.8% | - | 3% | - |
| Other meta | 0.2% | 0.2% | - | 0.2% | - |
| Data | 26.8% | 63% | 31% | 54.8% | 52% |

Table 3: Distribution of I/O types during a filebench run.

The benefit of separating metadata or journal into different streams depends on the amount of metadata write traffic and degree of mixing among various I/O types. As shown in Figure 5, Ext4Stream shows 35% performance increase and 25% WAF reduction than baseline ext4 for varmail. XFStream shows 22% performance increase and 14% WAF reduction for varmail compared to xfs. Both Ext4Stream and XFStream show more enhancements for varmail than for fileserver, because varmail is more metadata-intensive than fileserver, as shown in Table 3.

To investigate the effect of the stream separation for file systems without journaling, we disable the journal in ext4, denoted as ext4-nj. Under varmail workload, ext4-nj performs better than ext4 by 62%, which is mainly due to the removal of journal writes. Stream separation




(a) Throughput comparison

(b) WAF comparison

Figure 5: Above graphs show performance and WAF improvement percentage obtained with multi-stream variants of file-systems. Here Ext4-NJ denotes Ext4 without journal, and Ext4-DJ denotes Ext4 with data journal.

improves the performance and WAF of ext-nj by 26% and 46%, respectively.

A key observation in fileserver is that reducing metadata writes is more important for performance than reducing journal writes. xfs generates 16% more inode writes and 10% less journal writes for fileserver than ext4. Though the sum of metadata and journal writes is similar, xfs's performance is less than half of ext4's performance. The reason is that the metadata writes are random access while journal writes are sequential. Sequential writes are better for FTL's GC efficiency, and hence are good for performance and lifetime.

Another important observation comes from ext4 fileserver write distribution in Table 3 which shows 16% inode write, but only 0.2% other-meta which includes inode-bitmap as well. This is because of a jbd2 optimization. If a transaction modifies an already-dirty meta buffer, jbd2 delays its writeback by resetting its dirty timer, which is why inode/block bitmap buffer writes remain low despite large number of block/inode allocations.

As shown in Fig 5(b), ext4 WAF remains close to one during the fileserver test. However, when ext4 is operated with *data=journal* mode (shown by ext4-DJ), WAF soars above 1.5 due to continuous mixing between journal and application-data. Ext4Stream-DJ eliminates this mixing and brings WAF down back to near one.

## 4.2 Cassandra results

Cassandra workloads are data-intensive. Database changes are first made to an in-memory structure, called "memtable", and are written to an on-disk commitlog file. The commitlog implements the consistency for Cassandra databases as file system journal does for ext4 and xfs. It is written far more often than file system journal. By separating the commitlog from databases, done by file systems through detecting the file name of commitlog files, we observe 38% throughput improvement and 81% WAF reduction. Cassandra commitlog files are named as commitlog-* with date and time information. With fname_stream option, files with their names starting with commitlog flow into a single stream. Even if multiple instances of Cassandra run on a single SSD, commitlog files are written only to the stream assigned by fname_stream.

# 5 Conclusions

In this paper, we advocate an approach of applying multi-stream in the file system layer in order to address the SSD aging problem and GC overheads. Previous proposals of using multi-stream are either application level customization or block level full automation. We aimed to make a new step forward from those proposals. We separate streams at the file system level. We focused on the attributes of file system generated data, such as journal and metadata, because they are short-lived thus suitable for stream separation. We implemented an automatic separation of those file system generated data, with no need for user intervention. Not only have we provided fully automated separation of metadata and journal, but also we advise to separate the redo/undo logs used by applications to different streams. Physical data separation achieved by our stream separation scheme, FStream, helps FTL reduce GC overheads, and thereby enhance both performance and lifetime of SSDs. We applied FStream to ext4 and xfs and obtained encouraging results for various workloads that mimic real-life servers in data centers. Experimental results showed that our scheme enhances the filebench performance by 5%∼35% and reduces WAF by 7%∼46%. For a Cassandra workload, performance is improved by up to 38% and WAF is reduced by up to 81%.

Our proposal can bring sizable benefits in terms of SSD performance and lifetime. As future work, we consider applying FStream to log-structured file systems, like f2fs, and copy-on-write file systems, e.g., btrfs. We also plan to evaluate how allocation granularity and optimal write size affects the performance and endurance.

## References


[1] The NVM Express 1.3 Specification. http://www.nvmexpress.org/.

[2] AXBOE, J. Add support for write life time hints, June 2017.

[3] CHIANG, M.-L., LEE, P. C., AND CHANG, R.-C. Managing flash memory in personal communication devices. In *Consumer Electronics, 1997. ISCE'97., Proceedings of 1997 IEEE International Symposium on* (1997), IEEE, pp. 177–182.

[4] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[5] EDGE, J. Stream IDs and I/O hints, May 2016.

[6] EDGE, J. Stream ID status update, Mar 2017.

[7] HSIEH, J.-W., KUO, T.-W., AND CHANG, L.-P. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage (TOS) 2*, 1 (2006), 22–40.

[8] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, 2014), USENIX Association.

[9] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *Usenix Winter* (1995), pp. 155–164.

[10] LAKSHMAN, A., AND MALIK, P. Cassandra. http://cassandra.apache.org/, July 2008.

[11] SHIMPI, A. L. The ssd anthology: Understanding ssds and new drivers from ocz. http://db-engines.com/en/ranking/wide+column+store, February 2014.

[12] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *USENIX; login 41* (2016).

[13] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. In *FAST* (2017), pp. 15–28.

[14] YANG, F., DOU, K., CHEN, S., HOU, M., KANG, J., AND CHO, S. Optimizing nosql DB on flash: A case study of rocksdb. In *2015 IEEE 15th Intl Conf on Scalable Computing and Communications, Beijing, China, August 10-14, 2015* (2015), pp. 1062–1069.

[15] YANG, J., PANDURANGAN, R., CHOI, C., AND BALAKRISHNAN, V. Autostream: automatic stream management for multi-streamed ssds. In *Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR 2017, Haifa, Israel, May 22-24, 2017* (2017), pp. 3:1–3:11.