mStream: Stream Management for Mobile File System Using Android File Contexts

Yunji Kang
Sungkyunkwan University
Suwon, Korea
oso41@skku.edu

Dongkun Shin*
Sungkyunkwan University
Suwon, Korea
dongkun@skku.edu

ABSTRACT

The Flash-Friendly File System (F2FS) is a widely-used mobile file system. Since it is a log-structured file system (LFS), its segment cleaning operation is a performance bottleneck. To reduce cleaning overhead, F2FS uses the multi-head logging technique, which enables user to write different lifetimes of files into different segments. Currently, F2FS adopts the file-extension-based separation technique. However, it is difficult to predict the lifetime of a file only with the file extension information. In this paper, we analyze the lifetime of each directory of Android mobile application to make a lifetime prediction policy based on directory name and file extension. Each Android application is implemented using a standard Android library and manages files in a similar directory structure. As a result of the analysis, we identified that a single directory contains the file with similar lifetimes. Based on the analysis, we propose a new stream management technique for F2FS, called mStream, which separates the directories with different lifetimes into different segments. At the experiment, mStream reduced the segment cleaning cost by up to 35% compared to the original F2FS.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; **Architectures**;

KEYWORDS

Flash memory, Multi-streamed SSD, Log-structured file system, Segment cleaning

ACM Reference Format:
Yunji Kang and Dongkun Shin. 2021. mStream: Stream Management for Mobile File System Using Android File Contexts. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3412841.3442115

*Corresponding author

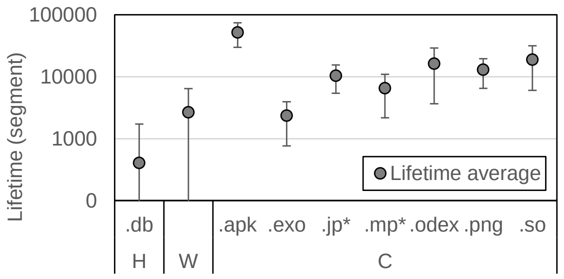

Figure 1: Lifetime Distribution of F2FS

1 INTRODUCTION

Storage performance has been recognized as one of the significant factors that affect the application performance for mobile devices [4, 6–8]. Mobile device uses a flash memory-based storage system such as embedded multimedia card (eMMC) and universal flash storage (UFS). To make flash-friendly write requests, a log-structured file system (LFS) [9] is a good solution since it generates only sequential writes. When there is an update on files, LFS writes it at the newly allocated free space called segment. These invalidated blocks become holes in segments and have to be reclaimed by segment cleaning. Because lots of copy operation occurs from cleaning, reducing the cleaning overhead is the most important part of LFS performance optimization.

Flash-Friendly File System (F2FS) [10], one of the LFS, is widely used for mobile platforms. F2FS provides multi-head logging so that the data with a similar lifetime will be placed in the same segment to reduce cleaning overhead. F2FS supports file-extension-based separation technique which classifies data to `HOT`, `WARM`, and `COLD`. For example, database files with .db extension are assigned through `HOT` segment while executable files such as .apk and .odex are assigned at `COLD` segment. In Figure 1, .exo file, which is the video cache file, is frequently updated similar to the files in `WARM` but F2FS allocates .exo file at `COLD` segment. Also, all files without any extension are allocated at `WARM` segment, so some `HOT` cache files are also included. By this observation, the file-extension-based separation technique is not enough to reflect the file lifetime precisely.

In mobile platform, especially in the Android platform, each application store its files on the formalized structure of a directory, as shown in Figure 2. As a result of the analysis, the mobile application has a characteristic of storing files with similar purposes in the same directory, so files in the directory have a similar lifetime. This paper proposes mStream, which provides semantic-aware stream management for mobile, by augmenting the file-extension-based semantic to directory-based semantic stream management. For mStream,

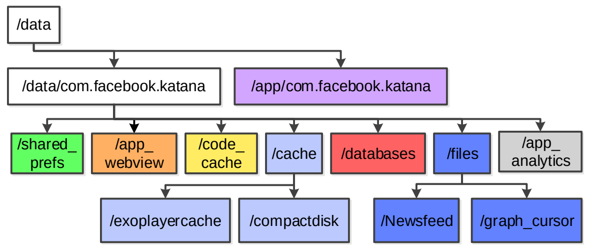

Figure 2: Facebook Application Directory Structure

we first collect the traces from the popular mobile workloads and measure each file's lifetime, directories, and file extensions. Based on this observation, we calculate the lifetime similarity between directory depth and lifetime difference inside the same directory. And then, we found the optimal point of file grouping and assigned each group to the same segment. These steps repeatedly occur in run-time so that mStream can dynamically adapt to the sudden change of the user's application usage pattern.

2 LIFETIME ANALYSIS

Android applications are implemented based on the Android library so that the directory structure of application data is similar to each other. Android application files are stored in the `/data/app/[package]` and `/data/data/[package]` directories of the data partition as shown in Figure 2. In the `/data/app/[package]`, the executable files and library files are stored. In the `/data/data/[package]`, there are sub-directories such as `/shared_prefs`, `/app_webview`, `/code_cache`, `/cache`, `/databases`, and `/files`.

- `/shared_prefs` contains the `SharedPreferences` objects.
- `/app_webview` contains `Webview` data that displays web content in the application. .
- `/code_cache` contains cached code.
- `/cache` contains the cache files that temporarily store the data required for the application.
- `/databases` contains the SQLite database file.
- `/files` contains other related files for application. (e.g. configuration, cache, font, etc.)

2.1 Environment Setup

In the measurement, we use a Google Nexus 5 to collect file usage patterns of the application. The kernel version and Android version of the smartphone is 3.10.73 and 7.11, respectively. We collect the system call trace to analyze which files are created/removed/updated. By a small fix of the Android device kernel, we enable the ftrace tool and trace the open, create, unlink, mkdir, rmdir, fsync, rename, write, link, truncate, fallocate, and mmap system calls with the system call arguments. File operation is collected from the VFS layer. For the mmap case, we gathered the write trace when a dirty page of memory invokes the file system to deliver a write request into the storage device.

We collected various workloads from popular applications from the social networking, game, web browser, and web-based application. We selected total 14 applications which are `Twitter`, `Instagram`, `Facebook`, `Trivago`, `Airbnb`, `Ebay`, `Amazon`,

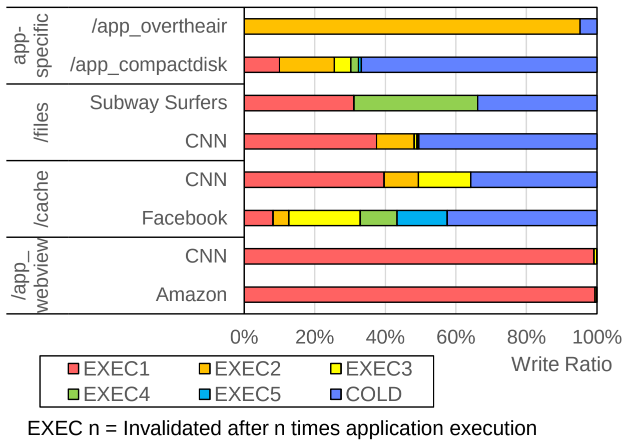

Figure 3: `/data/data/[package]` Sub-directory Lifetime Analysis

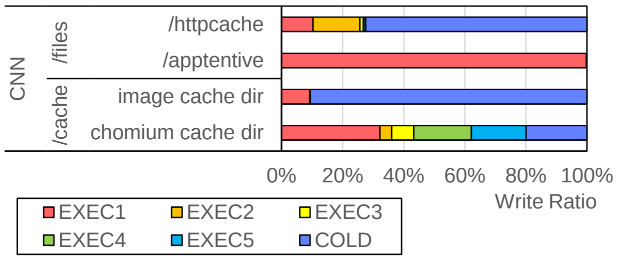

Figure 4: Depth-2 Directory Lifetime Analysis (`CNN`)

`Aliexpress`, `CNN`, `Angry Birds`, `Clash Royale`, `Subway Surfers`, `Firefox`. From these applications, we obtain the system call trace of installing apk, execution(`EXEC`) application with 5 different scenarios, update(`UPDATE`) to a newer version with `Google Play`, and uninstalling the application from the device.

2.2 Directory Lifetime Analysis

The files in `/data/app/[package]`, which contains the executable and library files, are updated simultaneously while the application `UPDATE` is triggered. For each `/data/data/[package]` subdirectory, we chose the applications based on the number of writes, and illustrated the lifetime distribution on Figure 3. The `/app_webview` stores the web contents to be displayed so that the files are updated when the application is executed. However, other directories such as `/files`, and `/cache` shows a mixed pattern of various lifetime. It means that simply grouping the files within the same directory can cause significant overhead to the storage device.

We distinguish the hot-cold distribution analysis on the deeper sub-directory for the `/files` and `/cache`, which show the mixed pattern of various lifetime data. We assume that the `/data/data/[package]` as a base (Depth-0) and the sub-directories directly located on `/data/data/[package]` (e.g., `/cache`, `/files`) as Depth-1, and the same method was repeated to define the depth value for each directories. For the `CNN` case, `EXEC1/2` and `COLD` are 50:50 ratio shown in Figure 3, but for Depth-2 analysis in Figure 4, most of the `EXEC1` category files are placed at `/apptentive`. Other

Table 1: The Weighted Arithmetic Mean of Lifetime Similarity

	Depth-1	Depth-2	Depth-3
/cache	0.62	0.66	0.66
/files	0.60	0.79	0.80
app-specific dir	0.58	0.64	0.64

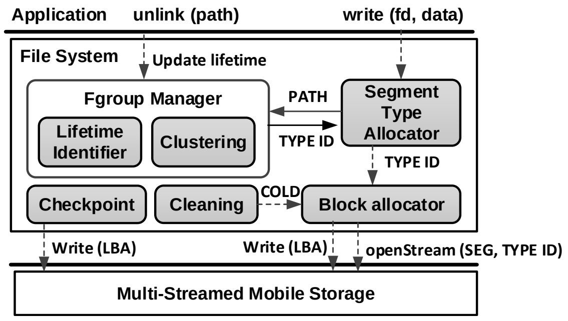

Figure 5: mStream Architecture

cases show similar observations, so that the in-depth analysis of the sub-directory level is useful.

We quantified the lifetime distribution correlation of files within the same directory through the similarity function described below.

$$S_{dir} = \max(W(T))$$

where T is a lifetime type (e.g., EXECn, UPDATE, COLD), W(T) is a write ratio of data corresponding to T lifetime. Table 1 is the weighted arithmetic mean of lifetime similarity, with the amount of write for each directory as a weight value. For /cache, /files, and app-specific directory, by choosing the Depth-2 directory, similarity factor increases 6.5%, 31.8%, 10.8% in average. Unfortunately, similarity factor of /cache is not that increased in Depth-3 compared to Depth-2, because /cache rarely contains the Depth-3 directories. Also, deeper depth causes the increase of investigation directories, which are directly matched to the stream. Choosing too deep depth for analysis can waste the stream resource. Therefore, Depth-2 is a good selection, and we choose this value for mStream.

3 DESIGN

Our key insight was that, the directory structure of the applications is similar in mobile applications. Moreover, the frequently accessed directory tends to have distinct lifetime patterns. We propose mStream, the novel architecture of managing stream by considering both file extension and the directory structure, which help file lifetime estimation. According to our analysis result, mStream provides a grouping methodology of files with similar lifetime. The group of files with similar file semantics (e.g. file extension, file PATH) is abbreviated as fgroup.

Figure 5 shows the overall organization of mStream. The assumed multi-streamed mobile storage provides a new API called *openStream(Segment #, Type ID)* which notify storage of the type ID allocated by each segment. When a write request is issued, the segment type allocator passes the file PATH to the fgroup manager, and the segment type ID is returned from the fgroup table. In the block allocator, data is allocated to the current segment corresponding to the segment type ID.

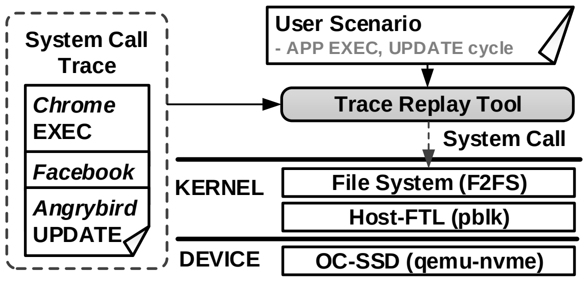

Figure 6: Evaluation Setup

Assigning fgroup. In the previous section, we analyzed the lifespan of the files in the application directory. As a result, we categorized data into fgroups in five ways as follows.

- File extension (e.g., .db-journal, .png)
- /data/app/[package]
- /data/data/[package] sub-directory (e.g., /app_webview)
- /data/data/[package]/cache sub-directory
- /data/data/[package]/files sub-directory

Lifetime Prediction. mStream manages fgroup table for the lifetime prediction of fgroup. Each fgroup entry keeps track of the invalidation history of files, which are included in fgroup. We use the exponential moving average(EMA) for estimating fgroup lifetime as follow:

$$EMA_T = (1 - \lambda)EMA_{T-1} + \lambda L_T$$

which L_T is the lifetime of Tth invalidate data, and EMA_T is an EMA of fgroup after Tth invalidate.

When the root directory of fgroup is unlinked, the fgroup manager removes the corresponding fgroup entry in fgroup table. Our investigation found a special case that the application deleted the directory and re-created the same directory. The Fgroup manager manages fgroup history table that stored information of recently removed fgroup. The fgroup manager searches the fgroup history table when a new fgroup is created and reuses the lifetime information if the history exists.

Clustering. After estimating lifetimes of fgroups, the fgroup manager attempt to cluster fgroups with similar predicted lifetime into segment type because F2FS supports a limited number of segment types. To classify fgroup, the fgroup manager uses K-means [5], which is a widely used clustering algorithm. We use the difference of the moving average lifetime between two fgroups as a clustering distance. To adapt changing user workload, the fgroup manager performs the clustering process periodically.

4 EXPERIMENT

4.1 Setup

To evaluate the performance of mStream, we implemented a host FTL supporting multi-stream on open-channel SSD (OC-SSD). The host-level FTL used in our implementation was pblk [3]. Figure 6 illustrates our evaluation setup. Our evaluation setup consists of a server with an Intel Xeon E5-2630 v3 (2.40 GHz, 16 cores) and 32 GB

Table 2: Experimental Scenario

	Workload A	Workload B
Number of Applications	7	22
EXEC Cycle	0.13-5.1 days	0.13-5.1 days
UPDATE Cycle	7 days	7 days
Initial Multimedia Size	3.8GB	2.4GB
Multimedia Update Size	40MB per a day	25MB per a day

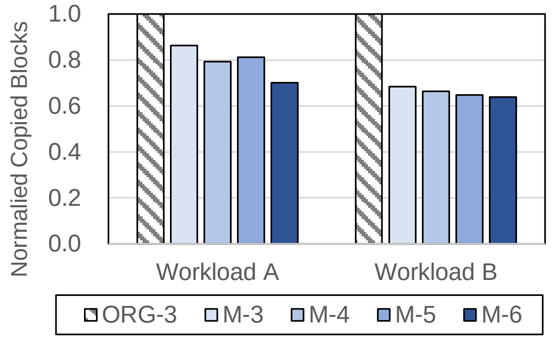

Figure 7: Impact of Multi-Stream Technique

DRAM. The Linux kernel version is 4.13.0. A qemu-based emulator, qemu-nvme [1], was used to emulate the OC-SSD. The OC-SSD emulation environment is as follows: 16 LUNs(Logical Unit Number, which is organized in planes, blocks, and pages), four planes, 4GB block, 16KB page, 4KB sector. We implemented mStream scheme on F2FS and configured the F2FS section size as 16MB that equal to the physical superblock size of OC-SSD.

We perform experiments using a mobile workload generator called the trace replay tool. The system call trace, which is the trace replay tool's input, is collected from the system call tracer on real-world smartphones. System call traces for install, execution, update, and uninstall were collected from various applications. We perform experiments using the workload of two user scenarios, and user scenarios are as shown in Table 2.

In the experiment, we compare the original F2FS and mStream. The F2FS format tools use f2fs-tools 1.4 version [2]. In the mStream, the λ value of EMA was set to 0.3.

4.2 Impact of Multi-Stream Technique

Figure 7 shows the cleaning costs of F2FS and mStream. We vary the host stream number limit of mStream from 3(M-3) to 6(M-6). For both workloads, mStream outperforms the original F2FS by up to 32% while adapting the same stream number limit. As the number of streams increases, mStream can classify data in a fine-grained manner, which improves cleaning efficiency.

To evaluate the effect of stream management, we measured per-stream lifetime distribution under workload B. Figure 8 shows the average and variance of data lifetime. First of all, we compared the original F2FS and mStream with 3 streams(M-3). Stream 0 of ORG-3 has a higher lifetime variance than stream 0 of M-3. Because for ORG-3 case, stream 0 contains both `.db-journal` and `.db` together whose lifetime distribution is different. M-3 has a higher lifetime

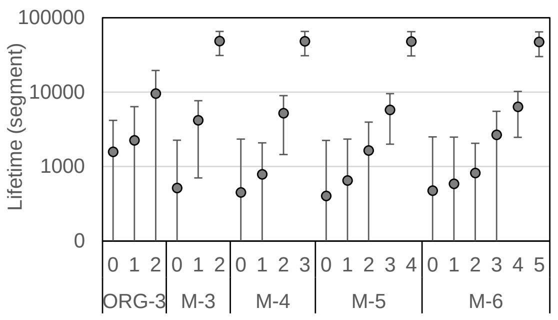

Figure 8: Lifetime Distribution

similarity of data of each stream compared to ORG-3. Stream 0 in M-3 gathers hotter data than ORG-3. Moreover, stream 2 also stores colder data than ORG-3. If the number of streams increases in the mStream technique, hot and warm data are more classified precisely.

5 CONCLUSION

This paper proposed mStream, a novel stream management technique based on file lifetime and I/O pattern on mobile platforms. mStream predicts file lifetime based on a directory structure and file extension. We implement mStream on top of the open-channel SSDs and reduce the cleaning costs. We observe that the directory depth analysis is important to handle the stream allocation and successfully merge this scheme on top of the F2FS. By using our mStream, we can reduce up to 35% of cleaning cost.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2020R1F1A1073758)

REFERENCES

[1] 2018. qemu-nvme. https://github.com/OpenChannelSSD/qemu-nvme
[2] 2020. f2fs-tools 1.4. https://github.com/jaegeuk/f2fs-tools
[3] Matias Bjørling, Javier González, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proc. of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. 359–374.
[4] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. 2017. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In *Proc. USENIX Annu. Tech. Conf.*
[5] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)* 28, 1 (1979), 100–108.
[6] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Proc. USENIX Annu. Tech. Conf.* 309–320.
[7] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Chun Jason Xue. 2020. Inspection and characterization of app file usage in mobile devices. *ACM Transactions on Storage (TOS)* 16, 4 (2020), 1–25.
[8] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting Storage for Smartphones. In *Proc. of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*. 1–25.
[9] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux Implementation of a Log-structured File System. *ACM SIGOPS Operating Systems Rev.* 40, 3 (Jul. 2006), 102–107.
[10] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proc. 13th USENIX Conference on File and Storage Technologies (FAST '15)*. 273–286.