An Empirical Study of F2FS on Mobile Devices

Yu Liang[1], Chenchen Fu[1], Yajuan Du[1], Aosong Deng[2], Mengying Zhao[3], Liang Shi[2], and Chun Jason Xue[1]

[1]Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong
[2]College of Computer Science, Chongqing University, Chongqing, China
[3]College of Computer Science, ShanDong University, Shandong, China

***Abstract*—Flash Friendly File System (F2FS) is getting popular among mobile devices. However, lack of empirical and comprehensive analysis for characteristics of F2FS prohibits better application of F2FS. In this paper, we present a set of comprehensive experimental studies on mobile devices and show several counterintuitive observations on F2FS, including imprecise hot/cold data separation, unexpected trigger condition of background GC, impact of fragmentation on read performance and impact of readahead by fragments and available space. Based on these observations, we further provide several pilot solutions to improve the performance of these mobile devices. The objective is to inspire researchers and users to pay attention to F2FS characteristics, and further optimize its performance.**

***Index Terms*—F2FS; Fragmentation; Readahead; Hot/Cold Data Separation; Background GC.**

TABLE I
THE MANUFACTURE DATE OF MOBILE DEVICES WITH F2FS. [3]

Mobile devices with F2FS	Manufacture Date
Huawei Mate9	2016
Huawei P9	2016
Huawei Honor 8/V8	2016
Oneplus 3T	2016
Moto Z	2016
Moto E LTE	2015
Google Nexus9	2014
Moto X family	2013/2014/2015
Moto G family	2013/2014
Motorola Droid family	2013
Moto MSM8960 JBBL devices	2012

I. INTRODUCTION

Flash Friendly File System (F2FS) has become one of the default file systems in Linux kernel since version 3.8. Because of the good performance and flash friendly characteristics such as hot/cold data separation for improving Garbage Collection (GC) performance of flash devices, F2FS is popular among manufacturers and consumers. Up to now, there are 11 models of mobile devices utilizing this file system, including Moto Z, Huawei Mate9, Google Nexus9, etc, as listed in Table I. Some devices use ext4 file system, but they support the transformation from ext4 to F2FS, such as Nexus6p, Oneplus One, etc.

Existing works aim at solving a specific problem related to F2FS characteristics. Park et al. [1] presented the fragmentation phenomenon of F2FS and provided a defragmentation method during GC. Park et al. [2] discussed the conflicts between sleep mode and background GC and further proposed a background GC scheme to solve this problem. Generally speaking, there are two challenges to perform a comprehensive analysis on F2FS. First, there are no cross-layer tools to perform a systematic analysis. A systematic study needs to collect information from virtual file system (VFS) layer, file system layer and page cache. Second, a comprehensive analysis of F2FS characteristics should consider the impact on other layers of Linux kernel and the overall performance of mobile system. To the best of our knowledge, there is currently no empirical and comprehensive study to present all important characteristics of F2FS, which limits further improvement of F2FS-based mobile systems.

This paper presents a set of empirical and comprehensive studies on several characteristics of F2FS. Linux kernel is annotated and recompiled to gather the information needed for this study. For example, to draw the read patterns of applications, we collect the ino, index and size of every read operation in VFS layer. To figure out the details of F2FS characteristics, we print the information from file system layer. In this way, real and correct data can be collected from the mobile devices. From the experimental results, we identify four main observations about F2FS:

- The default definition of data hotness is imprecise. For example, the experimental results show that in cold data type, there are some data with high update frequency. This definition may bring negative impact on GC performance of flash devices;
- The trigger condition of background GC of F2FS becomes limited when sleep scheme is introduced into mobile devices. This may result in an unexpected cost of foreground GC for reclaiming space, and degraded performance of mobile devices;
- The fragmentation of F2FS is severer with increased number of write operations. The sequential read performance will be degraded by fragments;
- F2FS cannot fully utilize the advantage of the readahead technique because the hit ratio of readahead is largely degraded by fragments and GC operations in F2FS.

Based on these observations, this paper further proposes several pilot solutions for existing problems. The objective is to inspire researchers and practitioners to pay attention to these F2FS characteristics which may result in performance degradation.

The rest of this paper is organized as follows. Section II describes an overview of Android I/O stack and background of F2FS. Section III to Section VI present and analyze four

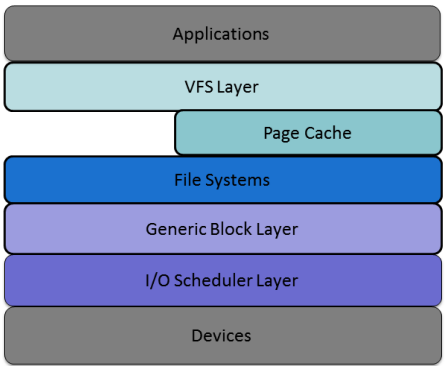

Fig. 1. Android I/O stack overview.

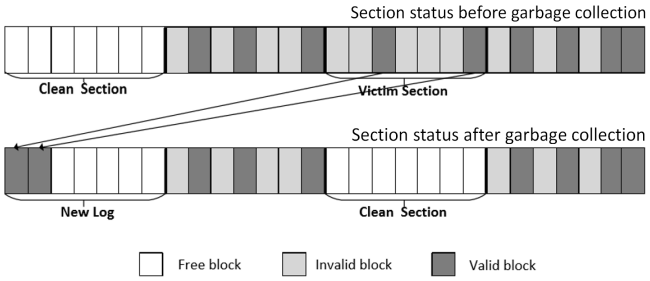

Fig. 2. Procedure of Garbage Collection of F2FS.

observations of F2FS. Section VII lists related works. This paper is concluded in Section VIII.

II. Background

This section introduces an overview of Android I/O stack, the background of F2FS and the merging operation of I/O scheduler.

A. Overview of Android I/O Stack

Android is an open-source operating system designed for mobile devices. Fig. 1 illustrates a simplified architecture of Android I/O stack with applications, five layers of Linux kernel and devices.

- ***Application*** is a program designed to perform some coordinated functions for the benefit of the user, such as YouTube, Google Chrome, etc.
- ***Virtual file system (VFS)*** is an abstract layer on top of file systems. The purpose of VFS is to allow applications to access different types of file systems in a uniform way.
- ***Page Cache*** is part of the main memory (RAM). It provides quick accesses to cached pages and brings overall performance improvement.
- ***File System*** controls how data is stored and retrieved. Starting from the release of Android 4.0.4, F2FS is adopted as one of the default file systems.
- ***Generic Block Layer*** translates host requests into block I/O (bio). One bio is an ongoing I/O block device operation.
- ***I/O Scheduler Layer*** re-organizes I/O operations to decide the order of submitting to the storage devices.

B. Background of F2FS

This part introduces the background of F2FS, including the logging scheme, two garbage collection schemes and the basic F2FS structure.

With the increased number of flash devices, file systems need to be friendly to the flash memory, such as, implementing hot/cold data separation in file system and having no Journaling of Journal problem [4]. Towards this need, Flash Friendly File System (F2FS) was proposed [5], as an enhanced Log-structure file system (LFS) [6]. It inherits the logging scheme of LFS to sequentially log data. This logging scheme brings good write performance but introduces a problem, that the space of logs is quickly consumed. Hence F2FS needs a garbage collection (GC) scheme to reclaim invalid spaces for logging when free space is not enough. The procedure of GC is shown in Fig. 2. Before GC, there are valid blocks (in dark gray color) and invalid blocks (in light gray color) in the victim section. When the victim section is chosen in the execution of GC, the valid blocks are copied back to a clean section. After GC, the original invalid spaces are reclaimed to be free spaces.

In F2FS, there are two types of GC operations, foreground GC and background GC. The foreground GC will be triggered when free space is not enough. This kind of GC blocks I/O and brings an unexpected write latency. The background GC happens when the operating system is idle, and it stops when I/O requests arrive.

On-disk layout, F2FS divides the entire volume into six areas, and all except the superblock area consist of multiple segments which are fixed at 2 MB. A segment is composed of 512 blocks. A section consists of consecutive segments, and a zone is made up of a set of sections. The section is the unit of GC. Each block is typed to be data or node. A node block contains inode or indices of data blocks, while a data block contains either directory or user file data. [5]

C. Merging Operation of I/O Scheduler Queue

In I/O scheduler queue, there are merge operations which reduce the number of I/O requests by merging the requests with same type (read, write) if they has a sequential logical address. An example is illustrated in Fig. 3.

Fig. 3, there are 8 read I/O requests with logical addresses in the I/O scheduler queue. The request with lsn 152 and the request with lsn 153 can be merged. The request with lsn 1 and the request with lsn 2 can be merged, and so on. Finally, according to the illustration, the amount of I/O requests is reduced from 8 to 5.

In the following sections, four interesting observations related to these characteristics are presented: imprecise hot/cold data separation, unexpected trigger condition of background GC, impact of fragmentation on sequential read and impact

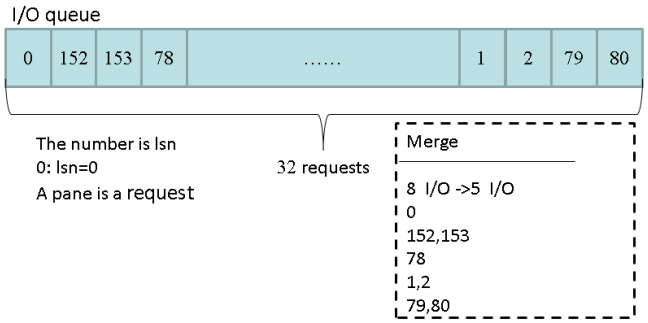

Fig. 3. Merging operation of I/O scheduler queue.

of readahead by fragments and available space. Several pilot solutions are proposed to relieve performance degradation caused by these problems.

III. Hot/Cold Data Separation

Hot/cold data separation is a flash friendly characteristic of F2FS for improving GC performance in flash-based devices. F2FS groups data by their expected lifetime, so that the data with similar life cycles will be written into the same blocks of the device. That is, the data in the same block have similar lifecycles, and all of them are valid or all of them are invalid within a period of time. Consequently, the number of valid data, which are copied by GC, is small. The cost of copying valid data will be reduced during GC. Generally, if data has high update frequency, it is defined to be hot. Otherwise, if data keep unchanged for a long time, it would be cold. This scheme is an optimized design on LFS. In current default design, the six active logs are used for storing Hot/Warm/Cold node and Hot/Warm/Cold data. The detailed definition of hotness in F2FS is listed as follows:

- **Hot node**: Direct node blocks of directories.
- **Warm node**: Direct node blocks except hot node blocks.
- **Cold node**: Indirect node blocks.
- **Hot data**: Directory entry blocks.
- **Warm data**: Data blocks except hot and cold data blocks.
- **Cold data**: Multimedia data or migrated data blocks.

The question is that whether it is reasonable to define the hot and cold data in this way. To find the answer, this section conducts a set of experiments to calculate the update frequency of data when running typical apps in mobile device.

A. Experimental Methodology

The experiments of this section are conducted on Google Nexus9 with Android platform. Android uses several partitions to organize files and folders on the device. Data partition of this mobile device uses F2FS file system, and other partitions use ext4 [7]. Data partition contains the users' data including contacts, messages, settings and apps that were installed. Google Nexus9 is equipped with 16GB internal eMMC NAND flash based storage and runs an Android Open Source Project (AOSP) version of Android 5.1. We added some codes to Linux kernel 3.4.0 for hot/cold data information collection and recompiled the kernel. To show the update frequency of hot and cold data, this work collects the number of write and update operations for each type of data on the file system layer in runtime.

TABLE II
Operations of typical apps

Apps	Operations
Angry Birds	Load and play Angry Birds.
Twitter	View news: (a) drag the screen to load friends' news; (b) load the news for displaying; (c) repeat (a) and (b).
Google Chrome	Search information: (a) type in keywords and enter; (b) open a web page; (c) drag the screen to load information; (d) repeat (a) (b)and (c).
Google Earth	Scan satellite maps: (a) drag the screen, zoom in and zoom out the map in online mode; (b) repeat (a).
YouTube	Play online video.

Inspired by work [8], five typical mobile apps are selected for the experiments. These apps cover a variety of scenarios that a user may have: Angry Birds (game), Twitter (Social Network), Google Chrome (browser), Google Earth (map), YouTube (online video). Operations of these apps are listed in Table II.

B. Experimental Results

Table III presents the update frequencies of hot and cold data when using five typical mobile apps. The mobile device was shut down before every test to flush the dirty data remained by last app. In addition, every test lasts 30 minutes to avoid the influence of page cache, so that a small number of data will be fully cached.

The results show that the update frequencies of cold data are very high as denoted in red font,which means that the definition of data hotness is imprecise. Some hot data are defined as cold data. In F2FS, cold data includes two types: data with cold flag and data belonging to cold file. The question is what kind of data are defined by mistake. To find the answer, another set of experiments are conducted. The results show that update operations come from the data with cold flag. And most of the updates are contributed by a few data. For example, out of 599 updates on cold data of YouTube, 106 updates are caused by ino:109788.

C. Pilot Solution

Based on these experimental results, we propose two pilot solutions for the imprecise hot/cold data separation. The first solution is to find the commonality of these cold data with a large number of updates and pull them off from cold data. The second solution is to redefine the data hotness by referring to hot data tracking [9]. Otherwise, a mixture of hot/cold data will hurt the GC efficiency in devices and degrade the performance [10] [11] [12] [13].

Similarly, another operation of F2FS that may lead to an unexpected performance degradation is background GC. The detailed analysis are presented in the next Section.

TABLE III
FREQUENCIES OF HOT DATA AND COLD DATA UPDATES.

Apps	Hot Data	Warm Data	Cold Data	Hot Node	Warm Node	Cold Node
YouTube Write	10	77111	0	1	1214	1208
YouTube Update	2098	2875	599	1717	3699	0
Update ratio	99.5%	3.6%	100%	99.9%	75.3%	0%
AngryBrid Write	13	69021	2444	13	943	912
AngryBrid Update	1421	8408	538	1106	6489	7
Update ratio	99%	10.9%	18%	98.8%	87.3%	0.8%
Earth Write	12	175164	0	12	911	760
Earth Update	1414	440613	15	1131	256519	155
Update ratio	99.2%	71.6%	100%	99.0%	99.6%	16.9%
Chrome Write	90	46712	14	17	5114	5126
Chrome Update	4452	10555	166	2578	12180	13
Update ratio	98%	18.4%	92.2%	99.3%	70.4%	0.3%
Twitter Write	106	25426	971	69	3451	3517
Twitter Update	4901	24618	0	3388	20245	0
Update ratio	97.9%	49.2%	0%	98%	85.4%	0%

TABLE IV
CONFIGURATIONS OF DEVICES.

Devices	Android	Storage	Recompile	Sleep setting
Nexus9	5.1	16G	kernel 3.4.0	7
HuaweiP9	6.0	64G	kernel 3.10	6

IV. TRIGGER CONDITION OF BACKGROUND GC

Garbage Collection (GC) scheme is a characteristic of LFS. F2FS inherits it as an important factor. Due to GC impacts performance, background GC, as a method to make up for the deficiency of foreground GC, should be designed carefully. The desired trigger condition of background GC in F2FS is the time when system is idle and invalid blocks are enough. Typical mobile devices apply sleep mode to save power. Hence the background GC in F2FS is affected unexpectedly. This section conducts a set of experiments to figure out the trigger condition of background GC in F2FS.

A. Experimental Methodology

In this section, considering the sleep mode variations of different devices, a set of experiments is conducted on two mobile devices with F2FS: Google Nexus9 and Huawei P9. The configurations of these two devices are listed in Table IV.

Huawei P9 is a mobile phone with 64GB of internal eMMC NAND flash based storage and runs Android 6.0. We added some codes to Linux kernel 3.10 for collecting background GC information and recompiled the kernel. When the system is idle, device will fall into sleep after the time period ranging from 15s to 10min. There are overall six user-defined sleeping trigger time settings: 15s, 30s, 1min, 2min, 5min, 10min.

Testing results indicate that when connecting the power cable, the device will never fall into sleep. The experiments are conducted by both online and off-line. Online means the device is connected to the power cable and off-line means the device is without the power cable. We use adb shell command ***cat*** and ***demsg -c*** to collect GC information online and off-line, respectively.

This work identifies the current trigger condition of background GC and its impacts on performance under several user-defined sleeping trigger settings in two mobile devices.

B. Experimental Results

In order to identify the trigger conduction of background GC after introducing sleep mode, a set of experiments is conducted in two mobile devices.

The information of background GC is online collected for about 7 minutes once under several user-defined sleeping trigger settings. To avoid interference, the system update is turned off. The experimental results show that when the power cable connects to the device, after the sleeping trigger time, the screen is turned off. However, the device does not fall into sleep mode, and the background GC neither stops. The number of background GC depends on the status of F2FS. More invalid block and less free space can trigger more background GC.

Furthermore, the information of background GC is off-line collected under several user-defined sleeping trigger settings. The experimental results show that without connecting the power cable when system is idle, the background GC will stop once the screen is turned off, no matter automatically or by the user.

It's worth noting that when some apps are running, the background GC could be triggered. For example, when user uses YouTube or Google map, the background GC can be triggered, no matter what the power cable connects to the devices or not. That is because these apps produce a few number of I/O operations. When background GC is checking, the system is considered to be idle.

In summary, the trigger condition of background GC becomes scarce. It only is happened in three cases. First, the system is idle and the power cable is connected, i.e., when the user is charging power and does not use the mobile device. Second, before device sleeps, the system is idle and is not connected to the power cable. For example, if user set "5 min", then the devices can do background GC during that 5 minutes. However, if the sleeping trigger time is too short, such as 15s or 30s, the background GC cannot be triggered. Additionally,

if that user manually turns off the screen after 1 minute, the devices can do background GC for the first 1 minute. Third, user is using some apps, such as YouTube, Google map.

To figure out the impacts of changed trigger condition of background GC on performance, the number of invalid blocks is collected as listed Table V.

TABLE V
NUMBER OF INVALID BLOCKS RECLAIMED BY BACKGROUND GC ONCE.

Available space	# of invalid blocks claimed by one background GC
4.3G	507 (512-5)
4.3G	510 (512-2)
4.3G	510 (512-2)
4.3G	508 (512-4)
4.3G	511 (512-1)
4.3G	511 (512-1)
618M	334 (512-178)
618M	333 (512-179)
618M	337 (512-175)
618M	249 (512-263)
618M	248 (512-264)
618M	307 (512-205)

Table V randomly collects the number of valid blocks in the section chosen by 12 background GCs. The number of invalid blocks is calculated by the formula:

$$\#ofblocks - \#ofvalidblocks = \#ofinvalidblocks.$$

The number of blocks per section is 512. Experimental results show that a large number of invalid blocks are reclaimed by background GC, especially when the available space is enough. That because when space is enough, the background GC will not be triggered frequently, and more blocks have time to become invalid. When available space is 618M, 36 background GC is triggered for 30 minutes, but when available space is 468M, 99 background GC is triggered for 20 minutes. That is because the number of background GC is affected by invalid blocks and available space.

In a word, the trigger condition of background GC becomes scarce and there is a large number of invalid blocks needed to be reclaimed. When the trigger condition of background GC becomes scarce, foreground GC will be triggered to reclaim these invalid blocks. However, the foreground GC will degrade the performance of mobile devices [2].

C. Pilot Solution

To solve this problem, two pilot solutions are proposed. First, do not use the mobile device (except some apps with few I/O, such as YouTube) when it is charging. During this time, the background GC can be performed due to power cable connection. Second, if user uses the mobile device when it is charging, we highly recommend that user set the system to fall into sleep mode after a longer time and do not turn off the screen manually. By this way, the device can get the time to do background GC. Otherwise, the mobile device may suffer an unexpected delay for foreground GC to reclaim space, which will significantly hurt user experience.

Another characteristic of F2FS, which can degrade the performance of mobile devices, is fragmentation. In the next Section, we will present that how much fragments are in F2FS and how fragmentation affects the performance of mobile devices.

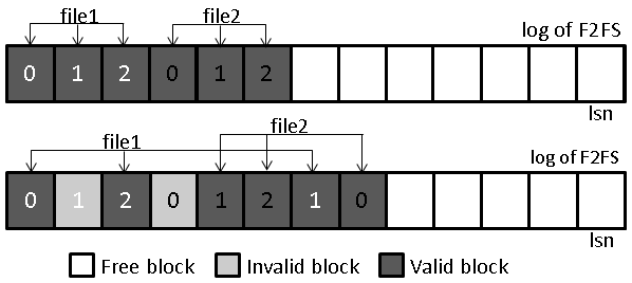

Fig. 4. Logging of F2FS.

V. IMPACT OF FRAGMENTATION

Fragmentation was previously discussed in some other file systems. Ji et al. [14] and Sato et al. [15] investigate the fragmentation and defragmentation operations in mobile storage with Ext4. In [16], the DFS file system performs file defragmentation when severe fragmentation is detected. However, because of the specific logging scheme and the rough GC scheme, F2FS accumulates fragments more easily when there are more and more operations in the mobile devices. Thus, it is important to study more details on its effect on system performance.

In this section, a set of experiments is conducted to reveal the fragmentation of F2FS and its impacts on the performance of mobile devices. Three aspects are involved in our experiment. First, the results of experiments show that there is fragmentation in F2FS by collecting the distribution of files read by applications. Second, this work discusses how the fragmentation affects the performance of sequential read. Third, this work presents the portion of sequential reads in applications and further explains why the fragmentation hurts system performance.

A. Experimental Methodology

The experiments of this section are conducted on a Google Nexus9. The experiments take three steps to show the fragmentation of F2FS and its impact on performance: First, in order to collect logical fragments, the command ***printk*** is used to print logical addresses of files on file system layer and then adb shell command ***cat*** is used to collect information online. Second, to figure out the fragmentation impact on sequential read, ***IOzone*** file system benchmark is employed to simulate the relationship between fragmentation and performance of sequential read. Random read operation is used to simulate the sequential read under fragmentation of file system. According to the user instruction of ***IOzone*** benchmark, if the amount of test data is smaller than memory size, Linux will cache all data and largely reduce the latency of read operation, which will lead an incorrect result from ***IOzone***. So 4G data (Google Nexus9

Fig. 5. The fragmentation of files in Google Chrome.

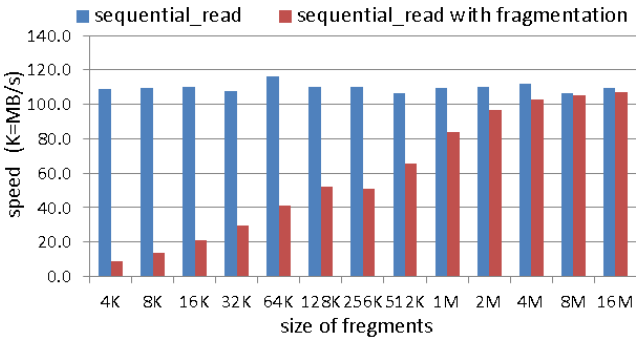

Fig. 6. The impact of fragmentation on sequential read.

has a 2G memory) are tested to avoid the influence of cache. Third, to reveal the ratio of sequential read in applications on mobile devices, indexes and sizes of requests were collected in VFS layer.

B. Logical Fragments in F2FS

Due to the logging scheme and the rough GC scheme, when there are more and more operations conducted, F2FS easily generates logical fragments. For example, when two files are updated at the same time, fragments are generated as shown in Fig. 4. Given two files: file 1 and file 2, where file 1 is written in sequential, and file 2 is appended at the end of the log. When part of file 1 needs to be updated, this part will be appended to the end of file 2. And then, when part of file 2 needs to be updated, this part will be appended behind file 1. This update process will introduce fragments. In addition, the GC scheme does not perform defragmentation during the procedure of copying valid data. Thus, fragmentation will not be relieved.

The distribution result of logical read addresses is shown in Fig. 5 by collecting when using Google Chrome to browse websites. This work randomly selects a logical address space from logical sector number (lsn) 4608 to 5120 and presents the information within these 16M space. There is no file read by Google Chrome in dark blue. Other colors present the files read by Google Chrome. The X-axis is the logical address from lsn: 4608 to lsn: 5120. The location of each color is the logical address of according file, and the same color means that addresses belong to the same file. The result shows that the same file are scattered, i.e., there are fragments in F2FS, and these fragments are generated by reads in applications.

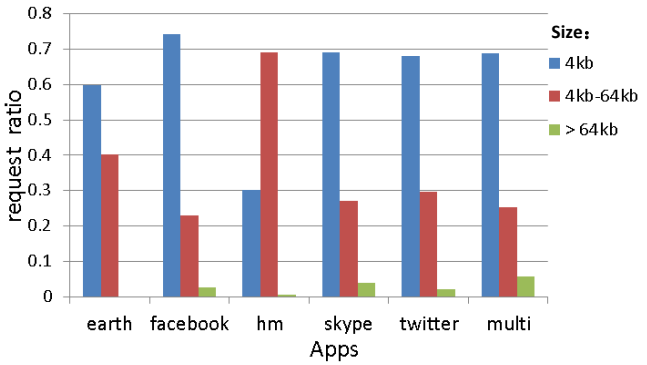

Fig. 7. Percentage of requests with various sizes over total requests .

C. Impact of Fragmentation on Sequential Read Performance

How logical fragments affect the performance of sequential read? To answer this question, this section uses ***IOzone*** to simulate the influence of fragmentation. ***IOzone*** is a filesystem benchmark tool to test the performance of file system [17]. Because the random read has the same features as sequential read with fragmentation, this work uses random read to simulate the sequential read with fragmentation. We compare the latency of reading 4G data in sequential and in random with various sizes. For example, if 4G data scattered in 16K fragments are sequentially read, the sequential read becomes random reads with the same size. This experiment scales 4K-16M size of fragments to simulate the sequential read without fragments and with fragments. The results are shown in Fig. 6.

In Fig. 6, sequential_read represents the sequential read without fragments. The random_read simulates the sequential read with fragments. The size presents the size of fragments. The results show that the performance of sequential read is significantly reduced when the fragmentation is severe. When the sizes of fragments are smaller than 64K, the sequential read performance is degraded by 64%.

However, what sizes of the requests do the mobile devices have? To figure out this problem, the request sizes were collected when using apps on mobile devices. The I/O size is listed in Fig. 7. The results show that more than 90% requests are smaller than 64KB, which means that F2FS easily generates fragments with size smaller than 64KB because of its logging scheme. Hence, the fragments severely affect the performance of sequential read.

Fragments affect both the hit ratio of readahead and merge operations of I/O scheduler and thus impact the performance of sequential read. Details of the effect on the first aspect will be presented in next Section. The reason of affecting merge operations is that in I/O scheduler, merge operations are used to reduce the number of I/O requests by merging the requests of the same type if they has a sequential logical address. However, the merge operation could be affected by fragmentation of file system because the fragment deranges the location of blocks in logical address. In this case, some requests can not be merged.

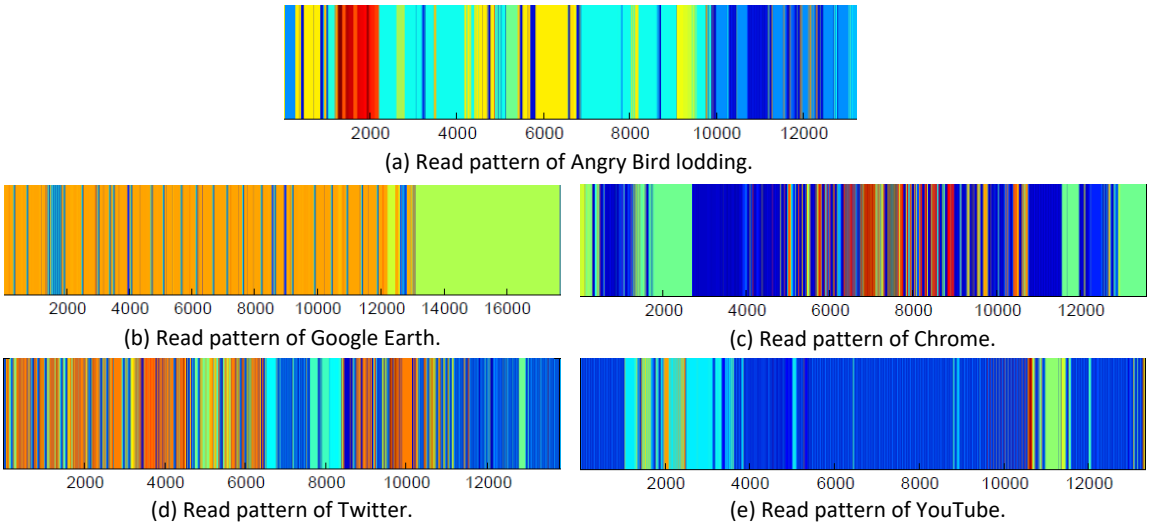

Fig. 8. Read pattern of various apps. The X-axis is the page number read by each application.

D. Sequential Read in Application of Mobile Devices

In order to figure out the portion of sequential read in various applications, read patterns of five typical apps are collected in VFS layer of the mobile device. We further analyze the number of inode (ino), index and the number of pages of each read operation, and present read patterns from these applications. In Fig. 8, the same color represents the pages from the same file.

Without loss of generality, the experiments choose the same times of read operations from five applications. The X-axis is the number of pages read by 19378 read operations of an application.

Fig. 8(a) show that when Angry bird is loading, the width of color is big, that means most of the operations are sequential read operations. Fig. 8(b) is the read pattern of Google Earth. The green color is **playlog.db**. It is read in the sequential way when using Google Earth. Fig. 8(c) is the read pattern of Google Chrome. The green color is **playlog.db-wal**. These files are read in the sequential way when Google Chrome is operating. Fig. 8(d) is the read pattern of Twitter. Most of the read operations are random read when using Twitter. Fig. 8(e) is the read pattern of YouTube. The blue color and the green color are files of database, which introduce sequential read. Thus, sequential read commonly exist in current applications of mobile devices.

In summary, the results show that applications have their specific read patterns and sequential reads are very common in these applications. The fragments severely affect the sequential read performance, degrade the performance of mobile devices and hurt user experience.

E. Pilot Solution

Defragmentation can be used to solve this problem. However, it gives additional overheads in order to read and write data from devices. For example, a defragmentation I/O control (ioctl) in F2FS [18] will reduce the life time of devices. User needs to tradeoff the performance and the lifetime to decide defragmentation setting in F2FS.

The fragmentation of F2FS also influences the hit ratio of readahead. The readahead scheme prefetches some pages which have sequential logical addresses with the page user reading. However, due to the fragmentation and GC of F2FS, the pages, prefetched by readahead scheme, are not what user expected. The details of readahead in the storage with F2FS will be discussed in Section VI.

VI. Impact of F2FS on Hit Ratio of Readahead

Readahead is an optimization technique of Linux kernel. The default setting in Linux kernel 3.4.0 is that when system detects that current request is a sequential read, readahead will be called. Readahead prefetches pages with the following logical addresses of current requested page into the page cache. If the prefetched page is what user expects in the next read, that is a "readahead hit", otherwise a "readahead miss". If readahead hits, user can read data from the page cache (RAM) rather than from a device, and this results in much lower file access latency. When the readahead hits in the current read, the system will increase the prefetched request size in next readahead. Otherwise, the prefetched request size will be degraded.

Readahead is a common scheme to improve the performance of sequential read. However, readahead scheme becomes controversial because the fragmentation and GC of F2FS will impact the hit ratio of readahead. For example, when a user reads 2 pages by 2 sequential read operations, the read procedure and the impact of fragmentation on readahead are illustrated in Fig. 9.

Fig. 9 shows how the user sequentially reads *index1* and *index2*, which are allocated in devices (SSD, eMMC, .ect). Fig. 9(a) illustrates that if the there is no fragmentation when

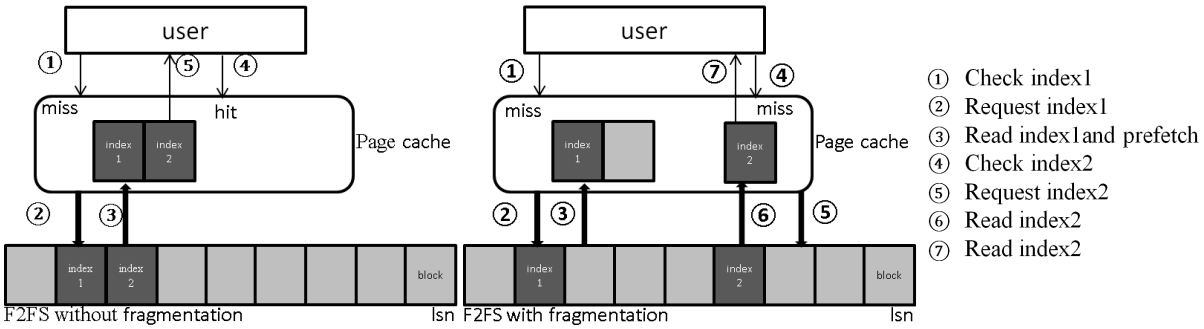

Fig. 9. Impact of fragmentation on readahead.

user reads *index1*, the readahead scheme prefetches *index2* to page cache. Thus, user can read *index2* from page cache in next read operation. There is only one I/O operation. Fig. 9(b) shows that if there is fragmentation, the fragments may lead to discontinuous logical addresses of *index1* and *index2*. When an user reads *index1*, the readahead scheme will prefetch another page into page cache. So user still needs to read *index2* from the device, not page cache. There are two I/O operations. That is why the fragmentation may affect hit ratio of readahead. To figure out the impact of fragmentation on the hit ratio, a set of experiments is conducted.

A. Experimental Methodology

The experiments of this section are conducted on a Google Nexus9. To reveal the impact of fragmentation on the hit ratio of readahead, a set of experiments is conducted in two cases: space is enough and space is limited in mobile devices. When space is not enough, more GC will be triggered to reclaim space, and the GC may affect the hit ratio of readahead by utilizing page cache. The experiments collect the information of readahead in memory management when using the five typical apps.

To figure out the impact of F2FS on hit ratio of readahead, this work compares the hit ratio with fragmentation and without fragmentation when space is enough. This work also compares the hit ratio in the scene without fragmentation when space is enough and when space is limited. The experimental scenes are created by these operations: mobile device is reset to eliminate fragmentation; some large *.rmvb* files are pushed into mobile device to occupy the space without introducing additional fragmentation; fragments are generated by running apps.

In order to avoid the impact of dirty data in page cache, every time app is run after a restart operation. We turned off system updates and closed other apps to reduce the influence from other apps. To avoid the impact of page cache, every experiment lasts 30 minutes, so that a small number of data will be fully cached.

B. Experimental Results

The comparison results of fragmentation and defragmentation are listed in Table VI. The results show that when the mobile device has enough available space, the hit ratio of readahead is reduced by fragmentation. The reduction of hit ratio is different in each app, because the readahead is only called by sequential read, and proportions of sequential read are different in five apps, as shown in Fig. 8. Most of read operations are sequential in Angry Bird loading, so it is affected significantly. The most read operation of Twitter are random, so it is seldom affected.

TABLE VI
IMPACT OF FRAGMENTATION ON HIT RATIO OF READAHEAD WHEN SPACE IS ENOUGH

Apps	defragmentation	fragmentation
Angrybrid loading	80.4%	73.2%
Google Chrome	61.5%	58.7%
Google Earth	72.8%	37.8%
YouTube	69.8%	65.1%
Twitter	35.9%	49.2%

TABLE VII
IMPACT OF SPACE SIZE ON HIT RATIO OF READAHEAD WITHOUT FRAGMENTATION

Apps	hit ratio with enough space	hit ratio with limited space
Angrybrid loading	80.4%	59.6%
Google Chrome	61.5%	58.7%
Google Earth	72.8%	52.6%
YouTube	69.8%	60.7%
Twitter	35.9%	28.1%

The comparison results of enough space and limited space are listed in Table VII. The results show that when the mobile device without fragmentation has a limited available space, the hit ratio of readahead is lower than enough space situation. There are two reasons: one is that GC of F2FS will take some spaces in page cache and the space for readahead will be reduced; another one is that SSR, which is a logging scheme different from sequential logging, will generate more fragments.

In summary, the results of above experiments show two crucial conclusions. First, the fragmentation reduces the hit ratio of readahead, so F2FS cannot fully utilize the advantage

of the readahead to improve performance; Second, hit ratio of readahead is affected by remaining available space in mobile devices.

C. Pilot Solution

To solve this problem, two pilot solutions are presented. First, turn off readahead scheme when fragmentation of F2FS is severe and the available space is not enough. In this case, the hit ratio is low, readahead cannot improve the performance. Second, do not use the F2FS in cache partition unless this partition is very large because the space occupied by F2FS is 114M while ext4 is 4M [19]. The hit ratio of page cache is largely impacted by its size.

VII. Related Work

Related works on F2FS can be classified into three groups: hot/cold data separation, background GC optimization and defragmentation in F2FS.

Separating hot and cold data is a common approach to improve the performance of mobile devices. A series of previous works have proposed the definition of data hotness. Work [11] identified hot data according to their sizes. Another work [20] defined the hot and cold data according to a formula related to age and write count of a block. We will analyze whether these definitions are suitable to re-design F2FS in the future work.

About background GC of F2FS, Park et al. [2] indicated that the background GC has conflict with the sleep mode. A suspend-aware GC is proposed to exploit the time between screen turn off and devices fall into sleep mode to perform background GC. However, they did not clarify the trigger condition of background GC on mobile devices using sleep mode. Our work presents the trigger condition of background GC for better usage of F2FS.

About fragmentation of F2FS, Park et al. [1] provided a defragmentation method during GC. This scheme aggregated the blocks of the same file when copying valid block, which can reduce the fragments of F2FS. Chao Yu [18] proposed a defragmentation enhancement patch for F2FS. In this patch, a new input/output control (ioctl) was proposed to support file defragmentation in a specified range of regular files. There are some works about defragmentation of ext4. Ji et al. [14] analyzed the fragmentation in ext4, and further provided two solutions to enhance file defragmentation for mobile devices. Sato et al. [15] provided an online defragmentation extension for ext4. These previous works mainly focused on defragmentation, that shows the fragmentation is a serious problem for performance.

VIII. Conclusion

F2FS is becoming more popular on Android mobile devices. This paper presented an empirical and comprehensive study on F2FS. The experimental results show that the definition of data hotness is imprecise; the trigger condition of background GC becomes scarce when the sleep mode is introduced; the sequential read performance is largely reduced by fragmentation; F2FS cannot fully utilize the advantage of readahead cache. We further analyzed the impacts of these problems on the performance of mobile devices and proposed several pilot solutions. As a result, this paper shows that large arrays of optimization opportunities exist in F2FS.

IX. Acknowledgments

We are grateful to Long Jin and Lei Ren from Huawei Technologies Co., Ltd. for their help. This paper is supported by National Science Foundation of China 61572411, National 863 Program 2015AA015304 and is partially supported by the Fundamental Research Funds for the Central Universities (106112016CDJZR185512) and NSFC (61402059).

References

[1] J. Park, D. H. Kang, and Y. I. Eom, "File defragmentation scheme for a log-structured file system," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 2016, p. 19.

[2] D. Park, S. Cheon, and Y. Won, "Suspend-aware segment cleaning in log-structured file system." in *HotStorage*, 2015.

[3] L. kernel, "Products with f2fs," 2016. [Online]. Available: https://f2fs.wiki.kernel.org/

[4] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/o stack optimization for smartphones." in *USENIX Annual Technical Conference*, 2013, pp. 309–320.

[5] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2fs: A new file system for flash storage." in *FAST*, 2015, pp. 273–286.

[6] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[7] XDA, "Nexus 9 uses f2fs as default file system in data partition," 2014. [Online]. Available: http://forum.xda-developers.com/nexus-9/general/nexus-9-f2fs-filesystem-default-t2929858

[8] J. Courville and F. Chen, "Understanding storage i/o behaviors of mobile applications."

[9] Z. Y. Wu, "Hot data tracking," 2012. [Online]. Available: https://lwn.net/Articles/515636/

[10] J. Lee and J.-S. Kim, "An empirical study of hot/cold data separation policies in solid state drives (ssds)," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 12.

[11] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 1, pp. 22–40, 2006.

[12] F. Zhang, "Hot-cold data separation method in flash translation layer," Nov. 16 2015, uS Patent App. 14/942,726.

[13] F. Yu, A. C. Ma, and S. Chen, "Green emmc device (ged) controller with dram data persistence, data-type splitting, meta-page grouping, and diversion of temp files for enhanced flash endurance," Aug. 2 2016, uS Patent 9,405,621.

[14] C. Ji, L.-P. Chang, L. Shi, C. Wu, Q. Li, and C. J. Xue, "An empirical study of file-system fragmentation in mobile storage systems," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, 2016.

[15] T. Sato, "ext4 online defragmentation," in *Proceedings of the Linux Symposium*, vol. 2. Citeseer, 2007, pp. 179–86.

[16] W. H. Ahn, K. Kim, Y. Choi, and D. Park, "Dfs: A de-fragmented file system," in *Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on*. IEEE, 2002, pp. 71–80.

[17] IOzone, "Iozone filesystem benchmark," 2016. [Online]. Available: http://www.iozone.org/

[18] C. Yu, "defragmentation in f2fs," 2015. [Online]. Available: https://lkml.org/lkml/2015/10/22/295

[19] Androguide.fr, "cache space utilization of f2fs vs ext4," 2014. [Online]. Available: https://forum.xda-developers.com/showthread.php?t=2697069

[20] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "Sfs: random write considered harmful in solid state drives." in *FAST*, 2012, p. 12.