File Fragmentation in Mobile Devices: Measurement, Evaluation, and Treatment

Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim and Chun Jason Xue

Abstract—Mobile devices, such as smartphones, have become a necessity in our daily life. However, users may notice that after being used for a long time, mobile devices begin to exhibit sluggish response. Based on an empirical study on a collection of aged smartphones, this work identified that file fragmentation is among the key factors that contribute to the progressive degradation of response time. This study takes a three-step approach: First, this study designed a set of reproducible file-system aging processes based on User-Interface (UI) script replay. Through the aging processes, it confirmed that file fragmentation quickly emerged, and SQLite files were among the most severely fragmented files. Second, based on the workloads of a selection of popular mobile applications, this study observed that file fragmentation did impact on user-perceived latencies. Specifically, the launching time of Chrome on an aged file system was 79% slower than it was on a pristine file system. Third, this study evaluated existing treatments of file fragmentation, including space preallocation, persistent journal, and file defragmentation to understand their efficacies and limitations. This study also evaluated a state-of-the-art copyless defragmenter, janusd, to show its advantage over the existing methods.

Index Terms—Measurements, flash memory, file fragmentation, I/O performance.

1 Introduction

Mobile devices, including smartphones, tablets and wearable devices, have gained increasing popularity among people. Users of mobile devices are highly sensitive to system responsiveness. Towards better user experience on mobile devices, research efforts have been made in various aspects, e.g., reducing energy consumption [3], enhancing mobile network performance [4], and optimizing I/O performance [5], [6], [7]. As reported in recent studies, I/O efficiency is among the key factors that affect the overall responsiveness of mobile devices [5], especially when mobile applications involve frequent transactions through the database middleware [6]. Many of the prior studies attribute the poor system response time to the highly random, synchronous I/O operations produced by the redundant journal mechanisms in the file system and database middleware [7]. However, a common perception among users is that the responsiveness of mobile devices progressively degrades over time. While a factory-reset on a mobile device would usually restore the original system performance, the sluggish system response time recurs again in a short period of time, usually a couple of weeks.

In this study, we identified that the progressive response time degradation in mobile devices is highly related to file fragmentation. After a file system undergoes sufficiently many file creation and deletion operations, it may have to store a new file in non-contiguous storage spaces. As a mainstream file system for mobile devices, Ext4 employs extent-based, delayed allocation strategies for fragmentation avoidance. However, the highly random, synchronous write behaviors of Android applications neutralize the efficacy of the anti-fragmentation mechanisms of Ext4. The goals of this study are to understand how bad file fragmentation in Android devices is and how fragmentation is produced (the measurement part), how fragmentation impacts on user-perceived latencies (the evaluation part), and the efficacy and limitation of existing methods of fragmentation management (the treatment part). These three parts are explained as follows:

Fragmentation Measurement. First, we attempted to understand how bad file fragmentation is in real smartphones. We picked up a selection of aged smartphones, each of which had undergone at least six months of daily use. By examining the file system snapshots of these smartphones, we observed severe file fragmentation. For example, on a one-year-old Google Nexus 5, the file `newsfeed_db-journal` of the Facebook app was fragmented into 7 pieces of 7 KB on average, and these fragments were randomly dispersed

This paper is an extended version of [1] and partially involves our prior work janusd *[2].*

C. Ji, R. Pan and C. Xue are with the Department of Computer Science, City University of Hong Kong, Hong Kong. (e-mail: chengji4-c@my.cityu.edu.hk; riweipan@cityu.edu.hk; jasonxue@cityu.edu.hk).

L. Chang is with the Department of Computer Science, College of Computer Science, National Chiao Tung University, Taiwan. (e-mail: lpchang@cs.nctu.edu.tw).

S.S. Hahn and J. Kim are with the Department of Computer Science and Engineering, Seoul National University, Korea. (e-mail: shanehahn@davinci.snu.ac.kr; jihong@davinci.snu.ac.kr).

S. Lee is with the Daegu Gyeongbuk Institute of Science and Technology (DGIST), Korea.(e-mail: sungjin.lee@dgist.ac.kr).

L. Shi is with the College of Computer Science, Chongqing University, China. (e-mail: shiliang@cqu.edu.cn) (corresponding author).

This work is partially supported by NSFC (61772092 and 61572411), Ministry of Science and Technology of Taiwan (MOST 104-2221-E-009-011-MY3 and 107-2628-E-009-002-MY3). Jihong Kim was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2015M3C4A7065645 and NRF-2018R1A2B6006878). Sungjin Lee was supported by the NRF grant funded by the Korea government (Ministry of Science and ICT) (NRF-2017R1E1A1A01077410) and the DGIST R&D Program of the Ministry of Science and ICT (18-EE-01).

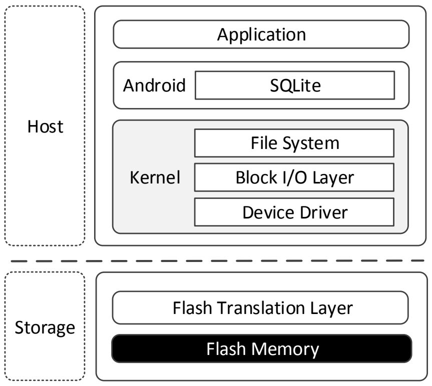


Fig. 1: Android I/O System Overview.

over a range of 1.5 GB storage space. We also identified that database files are among the most severely fragmented files. To understand how file fragmentation was developed in these devices, we designed a tool, AutoAge, which systematically ages a file system in a reproducible manner. We found that, even if the file system in a smartphone is initialized by a factory reset on the smartphone, with the aging process, file fragmentation emerged within a short period of use (in about two weeks). We also identified that the production of file fragmentation is closely related to the file access behaviors of the database middleware and the I/O behaviors of the Ext4 file system. They collaboratively produced highly random, synchronous block write requests, which defeated the anti-fragmentation mechanisms in Ext4.

Evaluation of Performance Impact. In the second part, we evaluated how fragmentation impacts on user-perceived latencies of mobile apps, including application launching time, application installing time, and SQLite query processing time. We found that the Chrome browser took 79% more time to launch on an aged file system of a Samsung S6 than it did on a pristine file system. While the common belief is that flash storage is free from the disk seek penalty and its performance is invulnerable to file fragmentation, we observed that accessing fragmented files resulted in frequent block I/Os, which accumulated a large time overhead on the I/O path. In addition, flash management inside of mobile storage devices is subject to a limited amount of embedded RAM, logical-to-physical address translation for flash employs a demand-based mapping cache. We observed that accessing fragmented files created a highly random I/O pattern, which diminished the locality of I/O requests and amplified the management overhead of the mapping cache inside mobile storage devices.

Fragmentation Treatment. In the third part, we assessed the efficacy and limitation of existing methods for fragmentation treatment, including space preallocation, persistent journal, and file defragmentation. We observed that their usefulness highly depends on file types and file system space utilization. For example, because Android apps created multiple threads that appended new records to different database files in parallel, space preallocation proactively prevented database files from being fragmented. Persistent journal prevented frequent creation and deletion of small journal files from creating free space fragmentation and thus alleviated file fragmentation. File defragmentation restored the space continuity of write-once files, such as executable files, because once these files were installed they never grew. However, when the file system space utilization was high, preallocation and defragmentation could not find contiguous free space and thus their efficacy became limited. On the other hand, when a large number of threads wrote different database files concurrently, the persistent journal did not prevent extents of different database files from being interleaved with one another.

Conventional file defragmentation is based on data copying, and frequent defragmentation is harmful to mobile storage because they employ high-density, low-endurance flash memory [8]. A state-of-the-art copyless defragmenter for mobile storage, janusd, exploits the existing logical-to-physical mapping mechanism to restore file continuity in the logical storage space without copying data in flash memory. We showed that, compared to e4defrag, the existing defragmentation tool of Ext4, janusd achieved the same result of fragmentation reduction but significantly reduced the total amount of data written to flash memory.

In summary, the contributions of this paper are as follows:

1) We measured that file fragmentation is a serious problem in real smartphones;
2) Through a reproducible file system aging process, we identified how file fragmentation is produced in smartphones and evaluated how it negatively impacts user-perceived latencies;
3) We assessed the efficacy and limitation of existing methods for fragmentation treatment and compared them against a state-of-the-art copyless defragmenter.

The remainder of this paper is organized as follows. Section 2 presents the background and related work. Section 3 investigates the degree that files are fragmented on real smartphones. Section 4 verifies that fragmentation is a common problem under realistic mobile usage scenarios. Section 5 evaluates the fragmentation impacts on application performance. Section 6 assesses the efficacy of existing treatments in mitigating fragmentation. Section 7 summaries the insights of fragmentation study for mobile device users. Finally, Section 8 concludes this paper.

2 Background and Related Work

2.1 Mobile Device I/O System Overview

There have been several highly successful operating systems for mobile devices, including Android, iPhone OS (iOS), and Windows Phone (WP). Android is an open-source approach among these popular operating systems. Fig. 1 illustrates an overview view of the Android I/O system architecture, which involves not only multiple software layers in the host but also the firmware and flash memory inside of mobile storage. These components are explained in a top-down order, as follows:

SQLite: SQLite is a popular embedded database library and has been the default relational database engine for Android. The SQLite library is part of the application program, and it stores database files in the local file system. SQLite supports two typical journal modes, roll-back mode, and write-ahead-logging mode, for transactional management of application data. In the roll-back mode, before modifying pages in a database file, SQLite copies these pages to a journal file with a `.db-journal` extension, updates the database file, and then discards the journal file. In the write-ahead-logging mode, SQLite logs changes to a database file in a journal file with a `.db-wal` extension and then reports completion. The changes are later applied to the database file at proper timing. These journaling mechanisms prevent unexpected system crashes or power fails from corrupting application databases.

Ext4 File System: Starting from the release of Android 4.0.4, Ext4 became the default file system for internal storage. Ext4 employs extent-based space allocation to support large volume size and to mitigate file fragmentation. An extent represents a set of contiguous disk blocks. It is also equipped with a journaling mechanism to protect itself from crash-induced metadata inconsistencies.

Mobile Storage Device: Mobile devices employ flash-based storage devices for low-power, non-volatile secondary data storage. Mobile storage devices are highly sensitive to hardware costs, and typically they are equipped with a small number of flash chips and very limited amount of embedded RAM. Embedded MultiMedia Card (eMMC) [9] and Universal Flash Storage (UFS) [10] are two representative interface designs for mobile storage. This study involves both eMMC-based and UFS-based mobile storage devices.

Flash Translation Layer: Flash storage devices employ an internal firmware layer, called Flash Translation Layer (FTL), to provide block storage emulation on top of NAND flash. The basic functions of FTL include logical-to-physical address mapping, garbage collection, and wear leveling. Based on the mapping scheme, FTLs can be classified into block mapping, hybrid mapping, and page mapping [11], [12].

Mapping Cache: Due to the stringent hardware budget, mobile storage devices are equipped with a very limited amount of embedded RAM for write buffering and address mapping. A page-mapping FTL delivers good random write performance, but it requires a large mapping table. Map caching methods [12], [13] have been introduced to reduce the RAM space requirement of page-level mapping by caching a small active portion of the mapping table.

Flash Memory: Flash memory has several advantages, such as low random access latency and shock resistance. However, due to the erase-before-write constraint, page updates are serviced in an out-of-place manner, and garbage collection is necessary to timely reclaim the flash space occupied by outdated data.

2.2 Fragmentation in File Systems

File systems, such as Ext4, FAT and even the log-structured F2FS [14], all suffer from fragmentation. A file is fragmented if its contents are not entirely stored in a piece of contiguous space. We say that a file system is aging if file fragmentation is developing in it. File fragmentation is closely related to file deletion, which leaves non-contiguous free space in the file system and the file system cannot find contiguous free spaces for new files [15]. File fragmentation amplifies the disk seek frequency and largely degrades the performance of hard-drive-based storage systems [16]. Smith and Seltzer characterized the production of file fragmentation over time based on the daily file-system snapshots of five servers [15]. They also proposed a file system aging process that can reproduce file fragmentation based on several key factors of realistic file usages in server workloads. Conway et al. [17] observed that read performance of data servers could severely degrade due to poor file layouts on hard disks or Solid State Drives (SSDs). It is a common belief that file defragmentation has little effect on SSDs and is even considered harmful to SSD lifespan [18]. However, in this study, we found that file fragmentation is a serious problem in real smartphones, and we also demonstrated that file fragmentation could significantly amplify user-perceived latencies.

2.3 File Defragmentation

There have been a series of studies on file fragmentation management, mostly on file defragmentation. DFS is a file system that automatically re-clusters data blocks of small, fragmented files in sequential free space [19]. On many file systems, users can manually invoke the defragmentation process on selected files. For example, `e4defrag` [20] is a file defragmentation tool for Ext4 file system. To defragment a file, `e4defrag` creates a large, sequential donor file, copies data from the fragmented file to the donor file, and then re-direct original file pointers to the donor file. However, the defragmentation process is based on data copying. Our prior study reported that frequent file defragmentation could shorten the lifespan of TLC-flash-based mobile storage by 10% or more [2]. Exploiting the existing logical-to-physical address mapping inside of mobile flash storage, we proposed janusd, which implements file defragmentation by re-mapping a file from fragmented logical addresses to sequential ones. This way, file defragmentation only modifies the mapping inside of flash storage but does not massive data copy in flash memory. In addition to file defragmentation, other system software layer can also employ anti-fragmentation methods. For example, SQLite can preallocate free space for files or use persistent journal. However, there is little work toward understanding the efficacy and limitation of these existing methods for fragmentation treatment.

3 An Empirical Study of Real Smartphones

Our first step is to empirically investigate whether file fragmentation is a real problem in real Android smartphones and, if so, how severe is file fragmentation in these devices.

3.1 Study Setup

We collected five Android smartphones for our empirical study, and the specifications of these devices are listed in Table 1. These smartphones include Google Nexus 5 (N5), Google Nexus 6 (N6), Samsung Galaxy S3 (S3), Huawei Ascend P7 (P7), and Samsung Galaxy S6 (S6). These smartphones were selected based on different manufacturers (Google, Samsung, and Huawei) and different storage specifications (eMMC and UFS). These smartphones had undergone at least six months of daily use, which involved a set of popular Android apps, including Facebook, Twitter,

TABLE 1: A summary of specifications of the Android smartphones and iOS devices under inspection.

	Nexus 5	Nexus 6	Galaxy S3	Ascend P7	Galaxy S6
Usage Duration	12 months	6 months	24 months	12 months	16 months
Release Date	2013	2014	2012	2014	2015
Linux Kernel	3.4.0	3.10.40	3.0.8	3.0.8	3.10.61
Android Version	5.0.1	5.1.1	4.3	4.4.2	5.0.2
Storage Standard	eMMC	eMMC	eMMC	eMMC	UFS
Data Partition	26.8 GB	26 GB	11.6 GB	11.8 GB	25.6 GB
Utilization	93%	57%	63%	51%	92%

	iOS Device 1†	iOS Device 2†
iOS Version	Before 10.3	Since 10.3
XNU Version	2782.70.3	4570.41.2
Darwin Version	14.5.0	17.4.0
File System	HFS+	APFS

† Because iOS prohibits installing unauthorized applications and modified kernels, we used a MacBook Pro with two different software configurations to simulate two iOS devices.

TABLE 2: Fragmentation Size Levels

Fragmentation Size Level	Size of fragmented file piece x (unit: KB)
Level 1	$0<x\leq 16$
Level 2	$16<x\leq 32$
Level 3	$32<x\leq 64$
Level 4	$64<x\leq 128$
Level 5	$128<x\leq 256$
Level 6	$256<x\leq 512$
Level 7	$512<x$

Chrome, Gmail, Camera, and Google Earth. We inspected file fragmentation in the `data` partition of every smartphone only, because it is the main read-write partition which occupied 73% and 84% of the entire storage capacity. By the time of inspection, the space utilization of the data partition in each smartphone was between 51% and 93%.

Degree of Fragmentation (DoF): We used the degree of fragmentation (DoF) of a file as a major evaluation metric for our study. The DoF value of a file is defined as the ratio of the number of extents allocated to the file x to the ideal (i.e., smallest) number of extents necessary for the file x. A single extent can cover up to 128 MB in Ext4 [21]. For example, ideally, a file of 1 GB has at least 8 extents (1024/128=8). If the file had 24 extents, then its DoF would be 3 (i.e., 24/8=3). The larger the DoF of a file is, the more severely the file is fragmented.

Fragmentation Size (FS): While DoF shows how severely a file is fragmented, it does not indicate the sizes of fragments. A large file and a small file may have the same DoF, but the small one would have smaller fragments. Because file fragments must be accessed through separate I/O operations, file fragmentation has a larger performance impact on small fragments than it does on large fragments. For this reason, we employ $Fragmentation\ Size$ (FS), which represents fragment sizes in seven levels, as shown in Table 2. For example, a level-1 fragment is not larger than 16 KB.

3.2 File System Snapshot Analysis

We developed an Android application FragCheck, which is based on a cross-compiled version of e4defrag, for our analysis of DoF and FS values in the smartphones [1]. FragCheck collects file fragmentation information in the `data` partition, including size and file type of each fragment. We took file system snapshots of the five smartphones and analyzed 16,440 files in total. Notice that, as reported in [22], real smartphones contain applications which are barely used after installation. We excluded files from our analysis if they were not accessed in last 30 days because they do not affect user experience.

Fig. 2(a) shows cumulative distributions of DoF values on the five smartphones [2]. Interestingly, in each smartphone, a noticeable portion of its files, between 14% and 27%, were fragmented (i.e., their DoF values were larger than 1). For example, 27% of the files in N5 were fragmented (717 out of 2,704 files), and 66% of these fragmented files even had DoF values larger than 2.

We are interested in which system parameters could be highly correlated with file fragmentation. We observed that the higher the file system space utilization was, the more files were fragmented. For example, N5 had the highest space utilization ratio (93%) and the largest portion of fragmented files. We also observed that files in older (i.e., used longer) smartphones tend to be more fragmented. For example, S3, which had been used for two years, had more of its files fragmented than N6 and P7, which were used no more than one year. The ratio of fragmented files in S3 was 27% while that of N6 and P7 was 15% and 14%, respectively. Note that S3, N6, and P7 had similar file system space utilization. Although the number of inspected smartphones was rather small, our observation indicates that file fragmentation is correlated with both file system space utilization and age of a smartphone. Since these factors would get worse as smartphones get older, file fragmentation is expected to be more severe as smartphones get older.

Fig. 2(b) shows a breakdown of fragment size levels of fragmented files of DoF >1 in the five smartphones. We found that in every smartphone, the level 1 (smallest) fragment size contributed to the largest fraction of the fragment size distribution, ranging from 45% to 60%. In other words, a fragmented file was very likely to be fragmented into small

1. FragChecker is available at https://github.com/cityu-mobile/
2. All the inspected files have integer DoF values. If a file is smaller than 128 MB, its DoF is always an integer. Only one among the inspected files was larger than 128 MB and the file had an integer DoF value

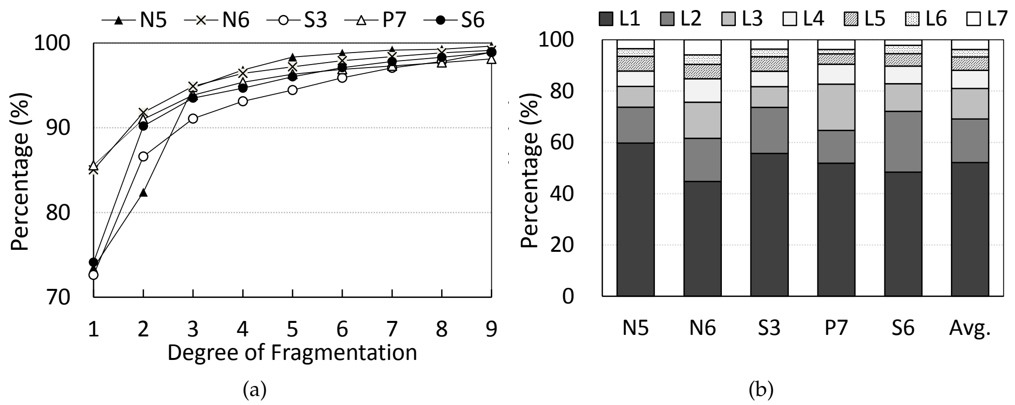


Fig. 2: (a) Cumulative distributions of DoF values. (b) A breakdown of files with different Fragmented Levels of fragmented file pieces.

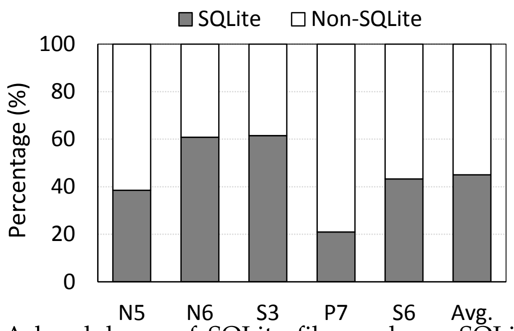


Fig. 3: A breakdown of SQLite files and non-SQLite files among fragmented files with DoF > 1.

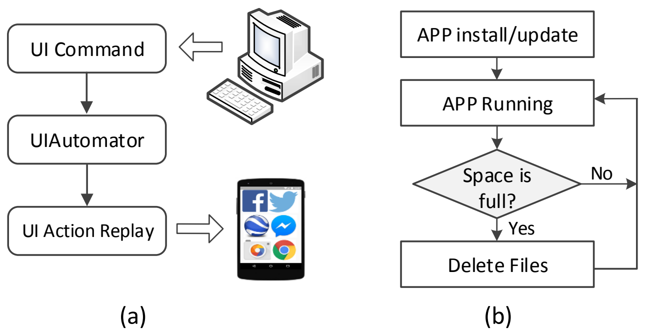


Fig. 4: (a) An overview of AutoAge framework. (b) Flow diagram of the daily usage by AutoAge.

pieces.

We further investigated whether fragmentation is correlated with file types. Fig. 3 shows a breakdown of SQLite files and non-SQLite files among fragmented files of DoF > 1. We observed that many of the fragmented files were SQLite files (with extensions `.db`, `.db-journal`, and `.db-wal`). On average, 45% of the fragmented files were SQLite files. The average DoF and file length of the fragmented SQLite files were 3.8 and 270 KB, respectively. This is rather counter-intuitive because these SQLite files were small, and small files should be less prone to fragmentation. Another observation from Fig. 3 is that non-SQLite files, including executable files and multimedia files, also significantly contributed to file fragmentation, accounting for about 55% of the total number of fragmented files. Their average DoF and file length were 6 and 3,579 KB, respectively. It was not surprising that non-SQLite files were fragmented in aged file systems because non-SQLite files were more than ten times bigger than SQLite files and the aged file systems cannot allocate enough contiguous free space for them.

In summary, our empirical study demonstrates that file fragmentation is a real problem in aged smartphones and most of the fragmented file pieces are of small sizes. Although Android-specific SQLite files are one of the key contributors to file fragmentation, our empirical study confirms that non-SQLite files also suffer from severe fragmentation.

4 File Fragmentation Reproduction

Our empirical study revealed that aged smartphones are prone to file fragmentation. The next question is how file fragmentation is produced in smartphones. In this section, we will systematically reproduce file fragmentation and investigate the correlations between system parameters and fragmentation production. Our fragmentation reproduction and analysis were conducted on both Android devices and iOS devices. iOS devices were involved in selected tests only because iOS prohibits installing unauthorized applications and modified kernels.

4.1 Scenario-Driven Application Replay

4.1.1 Workloads and Procedure

To produce file fragmentation in a controlled, reproducible manner, we developed a UI testing tool AutoAge[3] based on the UIAutomator framework [23]. As plotted in Fig. 4(a), the smartphone under test is connected to a remote desktop computer. The remote computer stores a collection of UI scripts, each of which consists of a series of UI events such as button clicks, dialogue inputs, and screen gestures. The UI scripts are downloaded to the UIAutomator on the smartphone, one at a time, for UI Action Replay to execute on the smartphone. A UI script simulates a user scenario of

3. The source code is available at https://github.com/cityu-mobile/autoage

TABLE 3: Application scenarios

Application	Type	Scenarios
Facebook	Social networking	Posting 1 image; Viewing news feeds for 5 minutes.
Twitter	Social networking	Posting 1 image; Viewing news feeds for 5 minutes.
Instagram	Social networking	Posting 1 image; Viewing news feeds for 5 minutes.
Chrome	Web browser	Surfing 10 webpages.
Messenger	Instant messaging	Typing in 10 characters and 1 picture; Repeating previous step for 10 times; Deleting some chatting history.
WeChat	Instant messaging	Typing in 10 characters and 1 picture; Repeating previous step for 10 times; Deleting some chatting history.
Earth	Map	Viewing online satellite maps for 5 minutes.
Gmail	Email	Sending an email with 10 characters; Repeating previous step for 10 times; Deleting 1 email.
Gallery	Multimedia	Viewing and deleting pictures.
Camera	Multimedia	Taking 30 pictures.
Youtube	Multimedia	Viewing online videos for 10 minutes.

an Android application. Table 3 is a summary of the user scenarios we used.

Fig. 4(b) shows the AutoAge execution flow of the application scenarios. These application scenarios involve applications installing, updating, and running. In our experiments, the application scenarios were repeated for 3.7 hours a day, which is the average daily smartphone using time reported in [24], 30 days in total. We performed the experiment on a Nexus 6 (N6). Before the experiment started, we filled the file system of the N6 using a large file until 70% full, which is the average file system utilization ratio of the five smartphones we inspected in the prior section. Because the large file was sequentially written as a whole, it entirely occupied a number of Ext4 block groups, each of which is 128MB, and did not introduce "holes" in the remaining free space. File system space utilization increased gradually as the aging procedure proceeded. When the file system was full, we deleted 300 MB of randomly selected image files, from the file system for the aging procedure to proceed further. We believe that Android phones and tablets, especially those models with moderate or small storage capacity, can easily experience a high file system fullness because of frequent application installing, temporarily file caching, and multimedia-content producing [25]. This issue would become more serious when such devices are used to store more digital assets [26].

4.1.2 *Reproduction Results*

Fig. 5(a) shows the DoF values of files with different types over 30 days. Notably, the space fullness was not high (about 70%) on the first day, but the DoF value of SQLite files was already as high as 3. On the eleventh day and onward, the fragmentation of SQLite files progressively developed as the file system aged. This is because every time when SQLite committed a new transaction to a database file, it appended a small record to the database file through synchronous block write requests. The highly synchronous write behavior defeated the fragmentation avoidance strategies of Ext4 such as delayed allocation. As a result, the small extents of SQLite files were interleaved with extents from other files. Regarding `Multimedia` and `Executable` files, their DoF values noticeably increased on the eleventh day and onward. This is because the file system was nearly full (almost 98% full) since the eleventh day. In this case, free space was severely fragmented and the file system could hardly allocate new files in contiguous free spaces.

Fig. 5(b) shows the fragment counts of different FS levels over 30 days. The number of fragments of the smallest FS level (fragment size between 0 and 16 KB) was the highest at all times, and the number significantly increased on the eleventh day. This is because when the file system was nearly full, free space was severely fragmented into small holes and new files were stored in these small holes. When files are fragmented into many small pieces, file access would involve frequent block I/O operations and an amplified total I/O overhead. On the other hand, the fragment count of the highest FS level 7 barely changed over time except on the eleventh day. The count suddenly increased on the eleventh day because some very large free spaces were fragmented into multiple holes not smaller than 512 KB, and these holes were quickly filled up by new files.

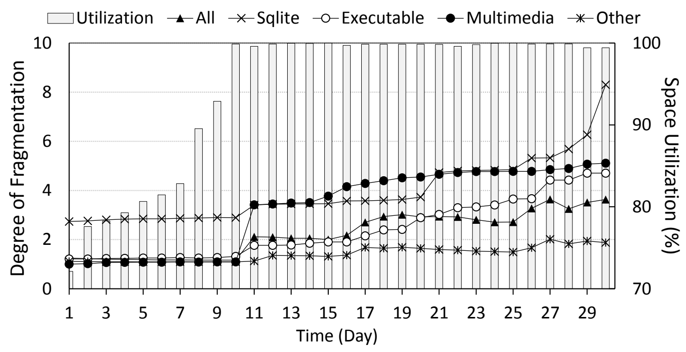
4.2 Multi-threaded SQLite Operations

4.2.1 *Workloads and Procedure*

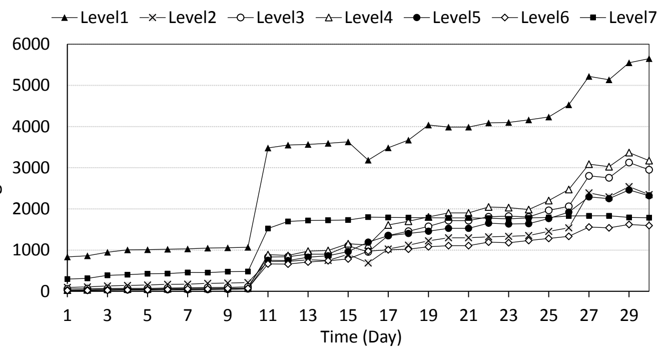
So far we have confirmed that file fragmentation is a real problem in Android devices. In this experiment, we inspected file fragmentation in another type of mobile devices, the iOS devices. Because iOS prohibits installing unauthorized applications and modified kernels, it is difficult to inspect file fragmentation directly in iOS devices. Alternatively, we chose to inspect file fragmentation in a MacBook Pro, which runs macOS and the kernel Darwin-XNU [27] is similar to that of iOS devices. The file system for iOS version 10.3 or later is APFS (Apple File System), while that for earlier iOS versions is based on HFS+. Both APFS and HFS+ are supported by the Darwin-XNU kernel on the MacBook Pro. We formatted two disk partitions, one in APFS and the other in HFS+, and conducted experiments on these partitions. Because iOS also uses SQLite [28], we employed Mobibench [29] to generate I/O workloads on top of the two partitions. Mobibench was configured to perform multi-threaded, concurrent insertion operations to different SQLite databases. The same test was also performed on an Ext4 partition for comparison.

4.2.2 *Reproduction Results*

Fig. 6 is a visualization of the final distributions of file fragments under APFS, HFS+, and Ext4. The fragments of



(a)



(b)

Fig. 5: (a) DoF values of different file types over 30 days. (b) Fragment counts of different FS levels over 30 days.

the same file are shown in the same color. The total number of threads was 1, 2, or 5. Overall, no matter what the file system was, when the total number of concurrent threads was large, small fragments from different files were severely interleaved with one another. When the total number of threads was 5, the average DoF values of SQLite files were 10, 15, and 5 under APFS, HFS+, and Ext4, respectively. In other words, file fragmentation is also a severe problem in iOS file systems, at least the problem is not lighter than it is in Ext4. The root cause of the problem pertains to the highly synchronous, multi-threaded writing behaviors of mobile applications (through SQLite).

4.3 Causes of Fragmentation

Write behaviors and file system space utilization are the two key factors in producing fragmentation as previously analyzed. To understand how these factors produced fragmentation, we performed a case-study under different file space utilization and then measured the DoF values of files. The results of SQLite and non-SQLite files are separately presented because they have different causes for fragmentation productions.

Analysis with Case-Study: The case-study was conducted using `Facebook` and `Google Earth` as representatives of applications because these two applications exhibited different file usage patterns in terms of fragmentation production. We employed another file system aging procedure based on that described in [30]: This procedure began with a pristine file system. We alternatively created large files ($\geq$ 10 MB) and small files ($\leq$ 500 KB) until the file system space utilization reaches a predefined high value. Once this predefined value was reached, small files were randomly deleted until the file system space utilization dropped down to a target value. We considered four different target values: `Low` (the pristine state), `Moderate` (the space utilization was less than 80%) , `High` (between 80% and 95%) and `Full` (higher than 95%) [4].

Impact of High Space Utilizations: Fig. 7(a) and 7(b) show the DoF values of the `Facebook` and `Google Earth` scenarios, respectively. The results of non-SQLite files are highly consistent with those in Fig. 5(a): When the file system utilization reaches `Full`, free space was fragmented into small holes and thus many non-SQLite files were stored in a large number of extents. For example, we found that 21 out of the executable files of Facebook (with the `dex` extension) were fragmented into 278 pieces in total, and the average size of these fragments was 243 KB. As the file system utilization increases, DoF values of SQLite files in the `Google Earth` scenario also increased noticeably. This is because the SQLite files of `Google Earth` were much larger (8,958 KB on average) than those of `Facebook` (64 KB on average). Compared to small files, large files would have a large DoF value when they are fragmented.

Impact of Write Behaviors: Even when the space utilization was `Low`, the DoF values of SQLite files were as high as 3 and 7 under `Facebook` and `Google Earth`, respectively. This result is closely related to frequent deletion of journal files and concurrent writing to multiple database files. The two applications operated most of their database files in DELETE mode. In this mode, a rollback journal is created at the beginning of a transaction and deleted on the completion of the transaction. Frequent deletion of the journal files left many small holes in the file system. On the other hand, in the `Facebook` scenario, 18 concurrent threads wrote to their database files. Because writes to SQLite files are highly synchronous [5], under frequent transaction commits, file system allocated small extents to different SQLite files, and these extents were interleaved with each other.

Impact of Storage Capacity: We have shown that a high space utilization in file systems has a direct impact on file fragmentation. It is then a question whether file fragmentation can be completely avoided by using a large storage device or decreasing the space utilization. To find this out, we replayed the application scenarios in Table 3 on a new 64GB Nexus 6P and the previous 32GB one. The space utilization of the 32GB model and the 64GB model were 0.5% and 0.2%, respectively, before the replays started.

Fig. 8 shows that, no matter how large the storage device was (32GB or 64 GB), the SQLite files were still badly fragmented. The DoF values of SQLite files were about 3 in most of the application scenarios, and the DoF values were not lower than 6 under Google Earth (whose database files were relatively large). This is because the mobile applications employed multiple threads to write SQLite files, creating interleaved fragments from different SQLite files regardless how large the free space was. By contrast, we

4. Random deletions of small files created holes in the file system, also known as `free space fragmentation`. To verify that free space was fragmented by the aging procedure, with the `Full` space utilization, we created a single large file to completely fill up the file system space. We found that the file was fragmented into 3,098 pieces and the average fragment size was 88 KB.

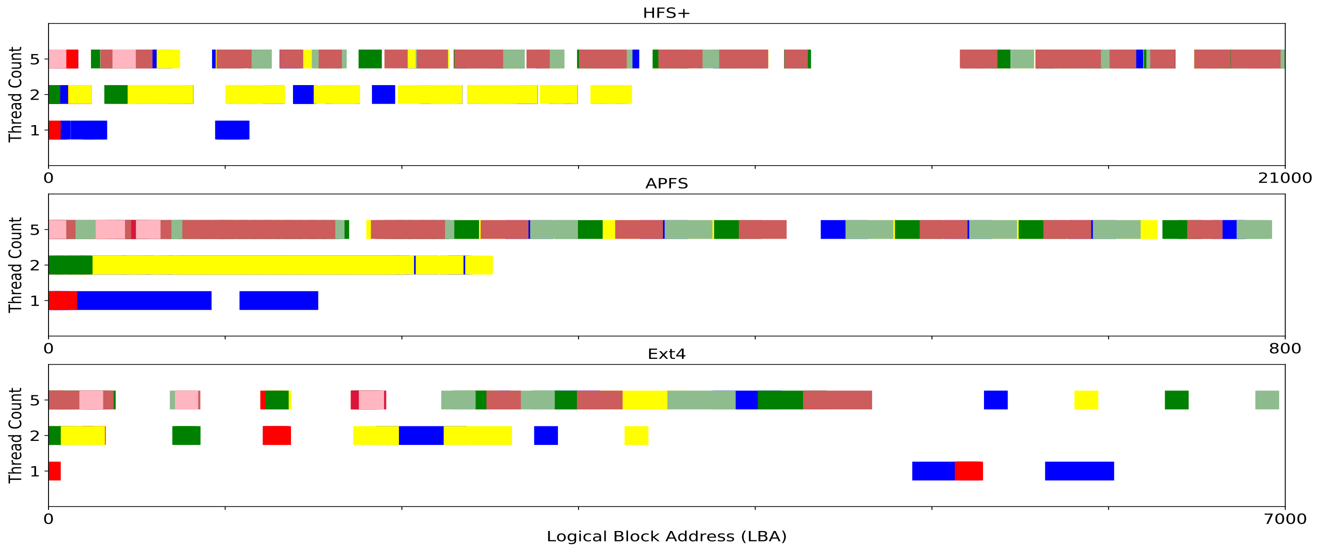


Fig. 6: The layouts of file fragments in HFS+, APFS, and Ext4. The fragments of a file are shown in the same color. The X-axis shows the relative logical block address (LBAs), and Y-axis corresponds to the total numbers of threads.

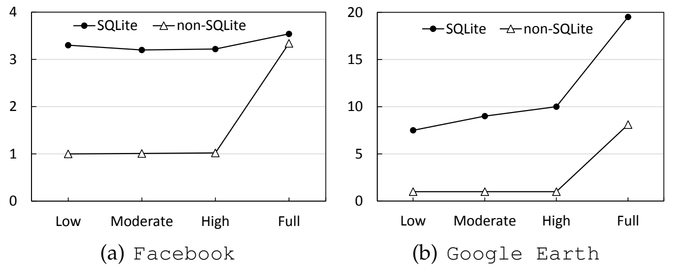


Fig. 7: Changes in DoF values for SQLite files and non-SQLite files under different file system utilizations. The Y-axis shows the DoF values.

observed that the average DoF values of non-SQLite files were as low as 1.1. This is because when there were plenty of free spaces in a file system, the free spaces were less prone to fragmentation and therefore new files can be stored in sequential storage space.

As previously shown in Fig. 5(a), when the file system utilization became very high (nearly 98% on the eleventh day), both SQLite files and non-SQLite files suffered from severe fragmentation. Therefore, we conclude that using a large storage device or decreasing the file system space utilization effectively avoids fragmentation of non-SQLite files, but it has no effect on SQLite files. This is because the fragmentation of non-SQLite is subject to free space fragmentation, while the fragmentation of SQLite files is mainly caused by the highly synchronous, parallel file-writing behaviors.

In summary, both SQLite and non-SQLite files suffer from severe fragmentation in an aged file system, but the reasons why they are fragmented are different. The DoF values of non-SQLite files are subject to free space fragmentation, which is highly correlated to the file system fullness. On the other hand, SQLite files suffer from severe fragmentation regardless of the file system space utilization or how long the file system has been used. This phenomenon is attributed to frequent deletion of small journal files and concurrent writing to multiple database files.

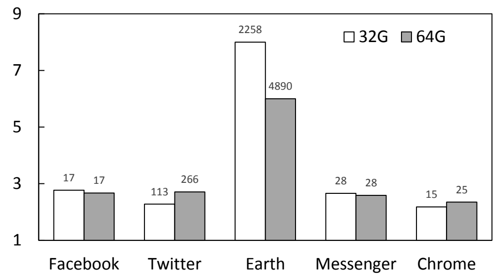


Fig. 8: DoF values (on the Y-axis) of SQLite files in a 32 GB smartphone and a 64 GB one. Above each bar shows the average Fragmentation Size (unit:KB).

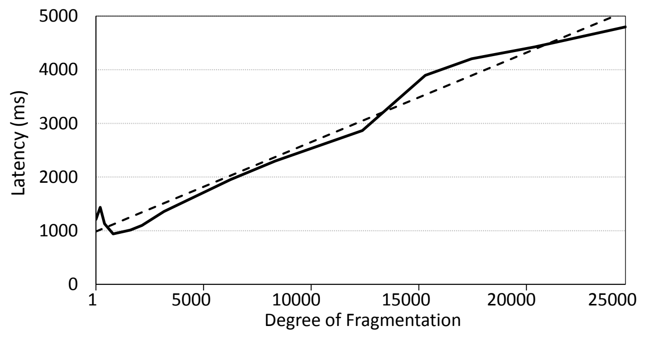


Fig. 9: The latencies of scanning a large file under different degrees of file fragmentation. The X-axis shows the DoF values of the file, and the Y-axis shows the elapsed time of scanning the file. The solid line and dotted line plot the real measurements and an approximation model, respectively.

5 PERFORMANCE IMPACT

In this section, we first introduced a performance model to reveal how I/O performance changed with different degrees of fragmentation. Next, we evaluated whether file fragmentation creates user-perceived latencies in three typical operations on smartphones, including application launching, application installing, and SQLite query processing.

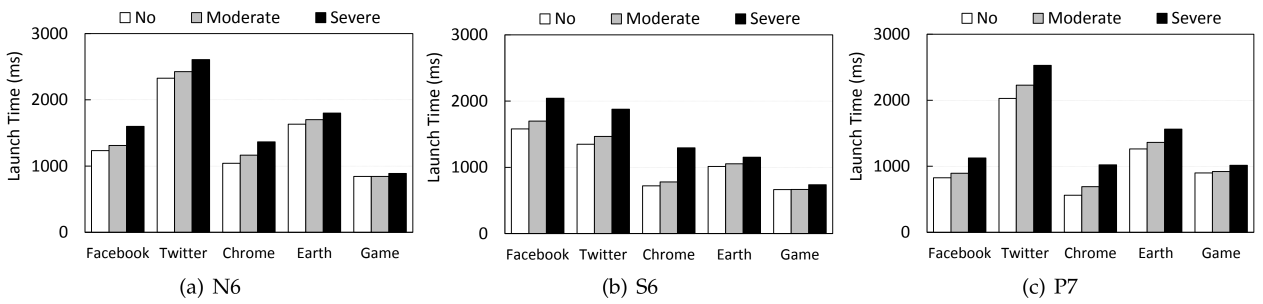


Fig. 10: Application launching times under different degrees of file system aging.

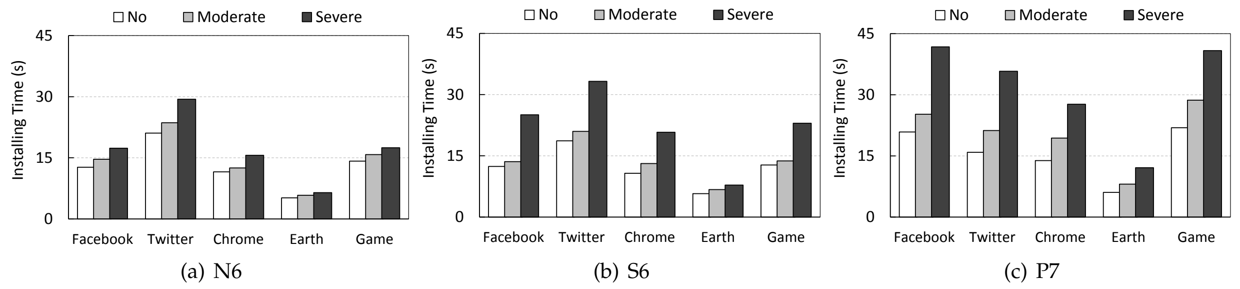


Fig. 11: Application installing times under different degrees of file system aging.

Finally, we investigated how I/O performance of mobile flash storage is affected by file fragmentation. The tests in this section were performed on both Android and iOS devices. Due to the same reason previously mentioned in Section 4, iOS devices were involved in selected tests only.

5.1 A Basic Performance Model

In this section, we attempt to characterize the correlation between the time overhead of accessing a file and the DoF value of the file. Because each I/O request is associated with an overhead on the I/O path, the more severely a file is fragmented, the longer time it requires to entirely access the file. To find out the correlation, we first created a 100 MB file without any fragmentation. The file was sequentially scanned using the `cat` utility [31], and we measured the total elapsed time of the scan. Next, we artificially increased the DoF value of the file by appending data to multiple files in parallel until the DoF value of the file reached a desired value, and then the file was scanned again to measure the elapsed time. This procedure repeated until the DoF value of the file reached 25,000.

The solid line in Figure 9 depicts the measured elapsed times (latencies) of different DoF values. Overall, the latency linearly increased with the DoF value. The relation between the latency t and the DoF value can be closely approximated by a linear function

$$t = 0.17 \times DoF + 990,$$

which is depicted by the dotted line. In particular, the latency of the worst fragmentation (DoF=25,000, each fragment was 4 KB) was 3.5 times larger than that without fragmentation (DoF=1, no fragmentation). Interestingly, the latency slightly dropped when the DoF increased from 1 to 400. We surmise that the performance fluctuation was related to the internal data placement of the eMMC device, i.e., the slightly increased I/O overhead was compensated by the performance gain of parallel accesses among flash chips. However, when the DoF value was larger than 400, the time overhead due to an increased I/O frequency started to dominate the latency. We must point out that this basic performance model is not intended for a general solution. Instead, it shows that the negative impact of file fragmentation on the file access overhead is evident, and the impact can be characterized by a linear relation.

5.2 User-Perceived Latencies

The experiment of user-perceived latencies employed the file system aging procedure previously described in Section 4.3. The file system fullness was set to High. We slightly modified the aging procedure to produce different degrees of file system aging: No, Moderate and Severe. The state No means that there were no small files for deletion. In this state, the file system was not aged at all because there were no small holes in the file system. To produce the Moderate state and Severe state, the average sizes of small files (which were deleted in the aging procedure) were 64KB and 8KB, respectively, and the total size of deleted small files was 500 MB. The smaller the deleted files, the more severe free space fragmentation was.

5.2.1 Application Launching

In this experiment, we installed five applications, including Facebook, Twitter, Chrome, Earth and Game (Angry Birds) on N6, S6, and P7. We measured the launching times of these applications under different degrees of file system aging. To measure the launching time of an application on a smartphone, we called the activity manager (`am`) from a remote PC using the `adb` command. The application launching time reported by the `adb` command is the time interval between the issue of the command and the time when all processes and Activities associated with the application have been started [5].

5. Activities are Android application objects responsible for user interaction.

Fig. 10(a), (b) and (c) shows the launching times of the five applications on the three phones. Let us first focus on the results of S6 in Fig. 10(b). We found that the launching times increased as the degree of file system aging became worse. In particular, the launching time of Chrome under the Severe state was 0.6 seconds longer (79% slower) than that under the No state. For the Severe state, we observed that the highest DoF value of the executable files of Chrome was as high as 968, and the average fragment size of these executable files was only 15 KB. The trends in the launching time slowdowns of the five applications appear highly consistent on N6, S6, and P7. Because users have to wait on application launching, our results clearly show that file fragmentation contributes to user-perceived latencies.

5.2.2 *Application Installing*

As mobile applications offer sophisticated functionalities, their installation packages (`*.apk` files) are increasingly large. Even though existing applications can be updated in the background, users usually wait when installing new applications. In this experiment, we evaluated the application installing times (latencies) under the three fragmentation states. To install an application, we called the packet manager (`pm`) from a remote PC using the `adb` command. The application installing time reported by the `adb` command is the time interval between the time when the install command is issued and the time when the installation finishes.

Fig. 10(a), (b) and (c) show the installing times (latencies) of five applications on N6, S6, and P7, respectively. The results appear highly consistent with those of application launching, showing that file fragmentation noticeably degraded the installing times. In particular, the installing time of Facebook on S6 in the Severe state was 25 seconds, while that in the No state was 12 seconds only. It is worth noting that while the application installing times on S6 and P7 were significantly affected by fragmentation states, those on N6 were less sensitive to fragmentation states. This phenomenon is also true in the experiment of application launching. We believe that the slowdown is closely related to design parameters of mobile storage, such as mapping cache size. While application installing is write-centric, application launching is read-intensive. Our results show that the latencies of write-oriented operations are more susceptible to file fragmentation.

5.2.3 *SQLite Query Processing*

In this experiment, we employed Google Earth to examine how file fragmentation affects SQLite query performance. Google Earth stores map tiles in database files. The performance of SQLite query affects the display of map tiles, and poor query performance introduces user-perceived latencies. Before this experiment, we prepared database file and SQLite queries using the following steps: We installed Google Earth on N6 and browsed online maps for 5 minutes. After this, we extracted the main database file of Google Earth, mirth_cache.db, whose size is 201MB, from the phone. Next, we switched the phone to the airplane mode and browsed the off-line maps for 30 seconds. During the off-line browsing, we collected the SQL queries on the main database file using the `adb logcat` command. We observed that all the collected SQLite queries are read-only, and most of them are table scans.

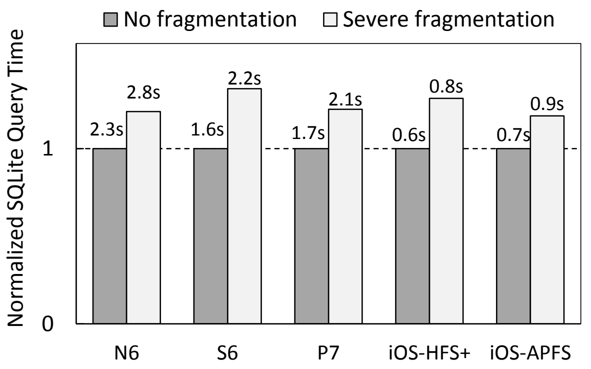


Fig. 12: Normalized total execution times of SQLite queries under different file system fragmentation states in Android and iOS devices. The absolute time value (in seconds) is labeled on top of each bar.

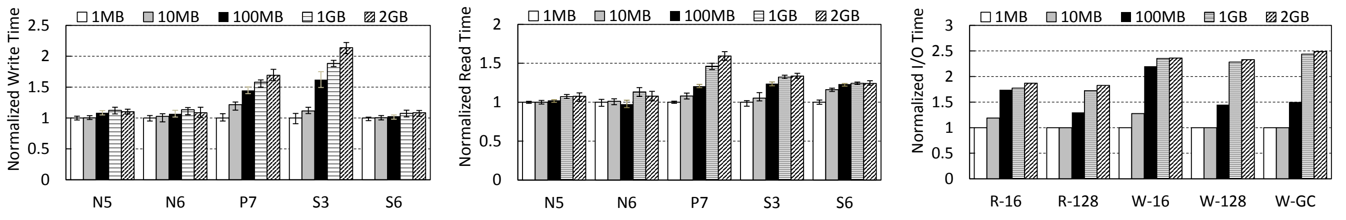
We conducted this experiment on N6, S6, and P7, and the two simulated iOS devices (iOS-HFS+ and iOS-APFS, see Table 1). For each run of the experiment, we pre-conditioned the fragmentation state of the file system using the same aging procedure described in the prior section, and then copied the main database file into the phone. We then replayed the collected SQL queries on the main database file using the `sqlite3` utility [32]. As Fig. 12 shows, the total execution time of the SQLite queries increased noticeably as the file system fragmentation state degraded. The query performance degradation caused by file fragmentation appeared highly consistent between Android devices and iOS devices. In other words, file fragmentation negatively impacts user-perceivable latencies in different types of mobile devices. With slow SQLite queries, users would have to wait until all high-resolution map tiles gradually pop up.

5.3 Causes of Performance Impact

5.3.1 *Increased Block I/O Frequency*

Accessing fragmented files would result in an increased I/O frequency. In this experiment, we attempt to investigate the correlation among file fragmentation, block I/O frequency, and file access latency.

This experiment was conducted on N6. We prepared a severely fragmented file by writing multiple files in parallel until the total size of these files reached 100 MB. One of these files, whose DoF value was 12,365, was selected, and the file was sequentially read once using the `cat` utility. While the file was being read, we used blktrace [33] to collect the I/O traces. We repeated the same procedure on a pristine file system, but this time only one single file of the same size was written to the file system to prevent fragmentation. We observed that the block I/O count with file fragmentation was significantly higher than that without fragmentation (47,560, vs. 812). This amplified block I/O frequency increased the total time of file reading by 1.6 seconds (2.9 seconds vs. 1.3 seconds). These results confirm that file fragmentation significantly amplifies I/O frequency and degrades file accessing performance.



(a) Random sync writes (b) Random reads (c) Random I/O on simulator

Fig. 13: Normalized (a) write elapsed time and (b) read elapsed time when changing the size of I/O region. I/O size: 4 KB, I/O number: 10000, direct IO mode is used in read. (c) Mapping cache validation. R-16 and R-128: read with 16 KB and 128 KB cache, W-16 and W-128: write with 16 KB and 128 KB cache, W-GC: write with 128 KB cache in presence of GC.

5.3.2 Dispersed I/O pattern

Since database files contributed to the majority of all block writes [5] and are among the most severely fragmented files, these files produced a large number of dispersed I/O over the entire storage [1]. We are interested in whether dispersed I/O patterns affect the I/O performance. We performed an experiment on the phones listed in Table 1 using the following procedure: We first prepared a large 1 GB file in the file system. To ensure that the file was as sequential as possible in the storage space, it was defragmented beforehand. We performed 10,000 synchronous file write operations on random offsets within the first 1 MB of the large file and measured the total file writing time. We repeated the same amount of file write operations on regions of the first 10 MB, 100 MB, 1 GB, and 2 GB of the large file. The larger the write region was, the more dispersed the write operations were in the storage space. Fig. 13(a) shows that the total write times on P7 and S3 with the largest write region were 69% and 113% slower than those with the smallest write region, respectively. On N5 and N6, the total write times of were less sensitive to the write region size, and a noticeable degradation appeared only when the write region was as large as 2GB.

We performed the same test again on the same phones, but this time we replaced file write operations with read operations. Interestingly, Fig. 13(b) also indicates that dispersed I/O patterns were unfavorable to the total read time. It had been reported in [34] that writes to random storage locations can seriously degrade the efficiency of garbage collection and increase write latency. However, as flash reads do not require garbage collection, the results in Fig. 13(b) appear counter-intuitive and need further investigation.

5.3.3 Pattern-Induced I/O Overhead

We surmise that the pattern-induced I/O overhead is closely related to the logical-to-physical map caching mechanism inside of mobile flash storage. Due to the stringent hardware budget, mobile flash storage is equipped with a small amount of embedded RAM. Instead of storing the entire page-level mapping table in RAM, mobile flash storage devices only cache active portions of the mapping table [35], [36]. This map caching design exploits temporal and spatial localities in I/O patterns [12], [13]. However, if an I/O pattern exhibits high randomness, the mapping cache could suffer from frequent cache misses, whose management significantly amplifies I/O latencies.

To verify our theory, we conducted an experiment on Flashsim [37], which provides a simulation environment of flash storage with map caching. We collected block read and write I/O traces of the procedure used in the prior Section 5.3.2 on the same phone N5. Flashsim employed the following simulation parameters: each mapping entry was 4 bytes, and the flash page size was 4 KB. A page stores the mapping entries of 1,024 sequential logical pages if it is a mapping page. Otherwise, it stores user data. Based on the design of [13], the mapping cache fetches and replaces mapping entries in terms of mapping pages, and the cache replacement policy was Least-Recently-Used (LRU). We evaluated a large cache size (128 KB) and a small cache size (16 KB). These parameters were based on the most common features of eMMC devices that we learned from our industry partners.

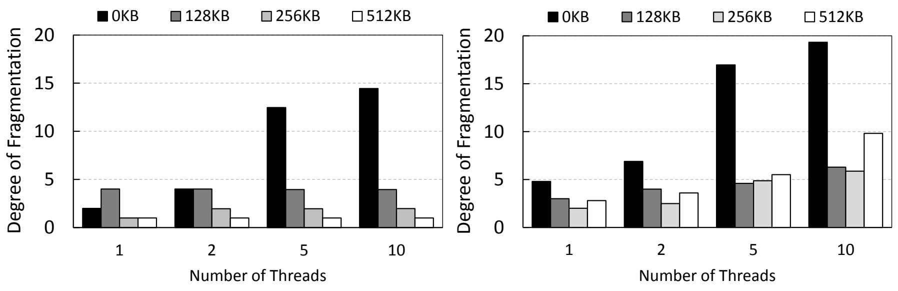
Fig. 13(c) shows our simulation results. We found that as the I/O region was enlarged, both the total read time and write time increased accordingly. The trends in the results of Fig. 13(c) appeared highly consistent with the results of P7 and S3 in Fig. 13(a) and Fig. 13(b). On the other hand, a large mapping cache, e.g., 128 KB, mitigated the extra I/O overhead due to mapping cache management under I/O region sizes not larger than 100 MB. We further noticed that, in the presence of garbage collection activities, the effect of the I/O pattern-induced performance degradation was even more significant. This is because garbage collection further amplified the overhead of cache miss management. In summary, our simulation results in general agree with the real measurements in Fig. 13(a) and Fig. 13(b) and thus confirmed the cause of the pattern-induced I/O overhead.

6 Fragmentation Treatment

In this section, we evaluate the efficacy of the existing methods for fragmentation treatment.

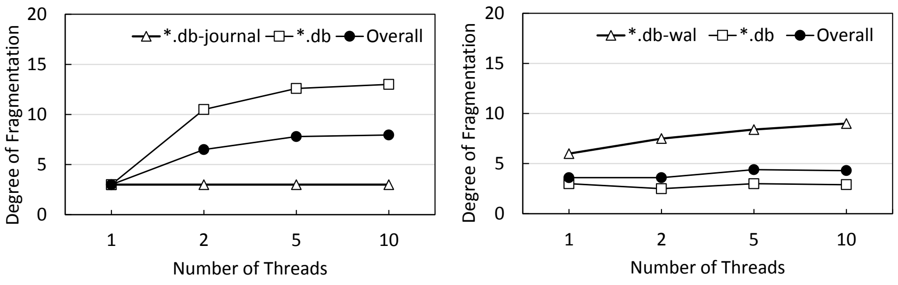
6.1 Fragmentation Avoidance

Frequent changes to file system structures, such as file growth and file deletion, quickly age file systems and introduce severe file fragmentation. In this section, we evaluate two fragmentation avoidance methods, including space preallocation and persistent journal, and observe their efficacy and limitations.



(a) Preallocation at `Low` utilization (b) Preallocation at `High` utilization

Fig. 14: DoF values of files with space preallocation (b) at `Low` space utilization and (c) at `High` space utilization.



(a) `PERSIST` mode (b) `WAL` mode

Fig. 15: DoF values of DB journals, DB files, and all SQLite files.

6.1.1 Space Preallocation

DoF Observation. Preallocating free space for files proactively mitigates file fragmentation because preallocation could obtain large, contiguous free space from the file system and file growth within the preallocated space does not require new extents. As previously discussed in Section 4.1.2, SQLite files were badly fragmented because multiple SQLite files grew concurrently in small size increments. We design a benchmark to assess whether space preallocation can prevent such concurrent, small file growth from fragmenting files into many pieces. The benchmark creates a set of threads, each of which writes its own file. Among these threads, each thread appends 4 KB records to its own file through synchronous writes until the file size reaches 512 KB. A thread calls posix_fallocate() to preallocate more free space for its file when the file is about to grow beyond its preallocated space. We repeated this test under the following parameters: the total number of threads was between 1 and 10, the preallocation size was between 0 (no preallocation) and 512 KB, and the file space utilization was either `High` or `Low` (see Section 4.3 for their definitions). Fig. 14(a) and (b) show the DoF values of files with space preallocation under `Low` and `High` space utilization, respectively.

Fig. 14(a) shows that under `Low` space utilization, using large preallocation sizes greatly helped eliminate file fragmentation, especially when the number of threads was large. When the number of threads was large (>=5) at the `Low` space utilization level, file fragmentation was completely eliminated (DoF value=1) using space preallocation with units of 512 KB, whereas the DoF without space preallocation was as high as 14.5. However, space preallocation for small files could waste storage space due to internal fragmentation. As shown in Fig. 14(a), the file DoF values with space preallocation units of 128 KB are reasonably low compared to those with larger preallocation units. It is worth noting that when there was only one thread, the file DoF values with 128 KB preallocation were slightly larger than those without preallocation. This is because the new extent allocated through posix_fallocate() is not necessarily adjacent to the last existing extent.

The results in Fig. 14(b) show that file DoF values under `High` space utilization were noticeably higher than those under `Low` space utilization. However, space preallocation still effectively reduced file DoF values. When there were 10 concurrently writing threads, compared to the baseline results without preallocation, space preallocation with units of 256 KB significantly reduced the DoF value from 19.3 to 5.9. However, the file DoF value was 9.8 with preallocation units of 512 KB, and this value was slightly higher than those with smaller preallocation units. This is because internal fragmentation in large preallocated units reduced the amount of limited free space in file system, exaggerating the fragmentation of free space and ultimately worsening file fragmentation.

Empirical Preallocation Size. The decision of the preallocation size for SQLite files is subject to a trade-off between file fragmentation and free space utilization. A sufficiently large preallocation space allows an SQLite file to grow without being fragmented, while an excessively large one creates unused free space. Here, we attempt to find a proper preallocation size for SQLite files of real mobile applications.

We empirically evaluated the effects of different preallocation sizes on SQLite files under the Facebook, Twitter, Chrome, and Messenger application scenarios in Section 4.3. We replayed each of the application scenarios for one hour to gather sufficiently many file operations for our analysis. Fig. 16 shows the percentages of un-fragmented SQLite files and the total amounts of unused (wasted) free space within the preallocated spaces under different preallocation sizes, which were between 16 KB and 512 KB. Ideally, space preallocation prevents files from being fragmented while a minimal total amount of unused free space. The results of

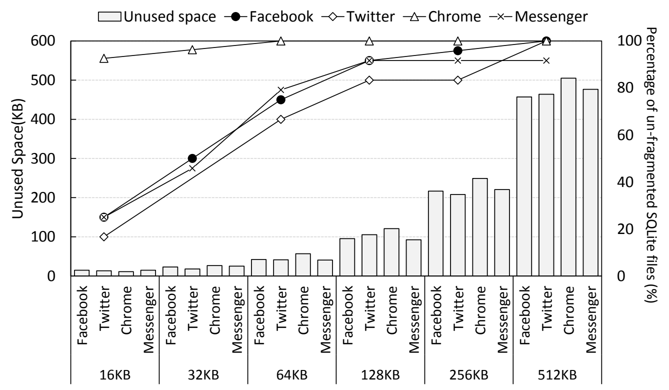


Fig. 16: Total amounts of unused preallocated space and the percentages of un-fragmented SQLite files under four application scenarios with different preallocation sizes. The X-axis depicts the pre-defined pre-allocation sizes.

different application scenarios appeared highly consistent: When the preallocation size was 128 KB, at least 83% percent of SQLite files were not fragmented (DoF value=1), and the marginal gain became very small beyond this preallocation size. On the other hand, the total amounts of unused preallocated file spaces appeared proportional to the preallocation size. This is because most of the SQLite files were very small, usually tens or hundreds of kilobytes, as previously shown in Section 3.2. Based on these empirical observations, we recommend 128KB as a preallocation size for SQLite files.

6.1.2 Persistent Journal

In DELETE mode and TRUNCATE mode, SQLite deletes and truncates the rollback journal every time it has committed a transaction, receptively. The default journal mode of SQLite is DELETE mode. Frequent deletion of small journal files quickly ages the files system through free space fragmentation. By contrast, in PERSIST mode and WAL mode, SQLite preserves the journal file after committing a transaction. Because applications can override the default journal mode, a question is whether or not using persistent journal can relieve SQLite files of fragmentation. We employed `Mobibench` [29] to perform the "insert operation" benchmark on a pristine file system. The benchmark used multiple threads to write different databases concurrently. We selected the number of threads among 1, 2, 5, and 10.

Fig. 15(a) shows the DoF values of database files and journal files in PERSIST mode. Fig. 15(b) shows the corresponding results in WAL mode. Results show that persistent journal helped reduce file fragmentation, but particular types of files were still badly fragmented, especially when the number of concurrently writing threads was large. In PERSIST mode, journal (`.db-journal`) files were lightly fragmented because they were written as a whole and barely grew. However, database files were badly fragmented because of the concurrent, frequent growth of multiple database (`.db`) files. In WAL mode, journal (`.db-wal`) files were badly fragmented because of the same reason above, but database files were lightly fragmented. This is because, in WAL mode, SQLite wrote multiple transactions to the database file in a large batch. Therefore, the growth of database files was relatively large and infrequent.

6.2 File Defragmentation

We have shown that fragmentation avoidance strategies have limitations in certain cases. Because file fragmentation is invertible, file defragmentation is useful to restore file continuity in aged file systems. Here, we evaluate the efficacy of file defragmentation under various system parameters.

6.2.1 Conventional Defragmentation

We considered e4defrag as the representative of conventional defragmentation approaches because the conventional defragmentation [14], [19], [38] are copy-based and essentially they are the same as the e4defrag utility. To defragment a file, e4defrag attempts to find as large extents as possible and re-locate the fragments of the file to the large extents. Ideally, a file would have the ideal DoF (i.e., 1) after defragmentation. Notice that e4defrag does not handle free space fragmentation. In addition, it is possible that a fragmented file still has a DoF value larger than 1 after defragmentation with e4defrag, because e4defrag cannot find a sufficiently large extent to store the fragments of the file. In this experiment, we performed the same benchmark previously described in Section 6.1 on a pristine file system, whose initial space utilization was either `High` or `Low` (see Section 4.3 for definition). After this, we used e4defrag to defragment all files and collected the file DoF values.

Fig. 17 shows the average DoF values of files before (`before_defrag`) and after (`e4defrag`) defragmentation using e4defrag. Results show that, when the file system space utilization was `Low`, e4defrag successfully eliminated all file fragmentation, achieving the best average DoF value of files (i.e., 1). By contrast, when the file system space utilization was `High`, e4defrag largely reduced but not eliminated file fragmentation, especially when the number of concurrently writing threads was large. For example, e4defrag decreased the average DoF only by about one half compared to no defragmentation at all when the number of threads was 10. In this case, free space was severely fragmented in this case, and e4defrag could not find sufficiently large free extents for defragmentation.

6.2.2 Copyless Defragmentation

Our prior study reports that file fragmentation in smartphones is a recurring problem [2]. This is mainly because popular applications are updated on a regular basis, typically once every 10 days [39], and the regular updates quickly age the file system and lead to severe file fragmentation. Our prior study recommends weekly file defragmentation to prevent file fragmentation from creating user-perceived latencies. However, conventional defragmentation approaches are based on data copying to relocate data from fragmented files to contiguous free space. The data copying introduces extra write stress to mobile storage. Because high-density flash memory such as TLC and 3D flash endure limited program-erase (P/E) cycles [40], our prior study also reports that weekly defragmentation could reduce the lifespan of mobile storage by at least 10%.

In this experiment, we evaluated the state-of-the-art technique, janusd [2], for defragmenting files on flash-based mobile storage. The basic idea behind janusd is to exploit the existing logical-to-physical mapping in the firmware layer

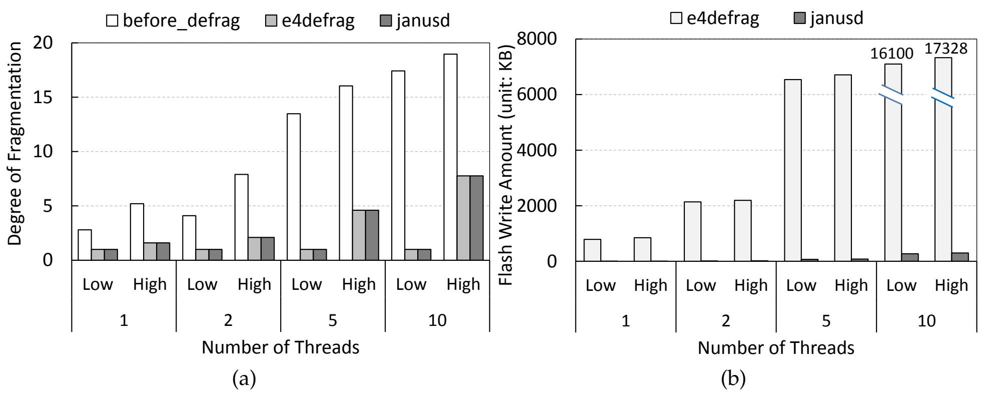


Fig. 17: (a) DoF values of files before defragmentation (before_defrag), after defragmentation with e4defrag and with janusd. (b) Flash write amount for defragmentation with e4defrag and janusd.

inside of mobile storage. To defragment a file, janusd sends individual commands to mobile storage to remap existing data in flash memory from fragmented logical addresses to continuous ones. This way, janusd eliminates the need for data copying during file defragmentation but modifies only the mapping information inside of mobile storage. We conducted the experiment described in the prior section again, but this time we replaced e4defrag with janusd.

Fig. 15(a) shows that janusd achieved the same average DoF values as e4defrag did. This is because janusd is based on the same defragmentation logic as that of e4defrag, but the difference is that janusd issues remap commands to mobile storage instead of copying data. Since the main advantage of janusd is write stress reduction, Fig. 15(b) presents the total amount of data physically written to flash memory by e4defrag and janusd. Results show that, on average janusd produced 98% fewer flash writes than e4defrag did. In other words, frequent defragmentation with janusd will not be harmful to mobile storage lifespan. The small amounts of flash write of janusd were produced by writing file system metadata such as inodes and updating logical-to-physical mapping information inside of the storage firmware.

7 Discussion and Insight

This work attempts to characterize the file fragmentation problem in mobile storage through a series of quantitative analyses, systemic reproduction, and comparative evaluation. We showed that file fragmentation is a real, recurring problem that could noticeably degrade user-perceivable latencies and identified the root causes of the recurring file fragmentation. Furthermore, we showed that existing defragmentation tools and a state-of-the-art copyless defragmenter both have their merits and limitations. We believe that this study provides useful insights into the design and implementation of file management algorithms for various layers in the I/O stack, including the SQLite library, file system, and performance maintenance tools. This section summarizes a few lessons learned from this study.

7.1 Causes of File Fragmentation

Through our experimental results, we identify two key factors responsible for the production of file fragmentation in smartphones:

High Space Utilization: High space utilization significantly impacts on all types of files concerning fragmentation production. Not only those write-once files, such as executables and multimedia files but also the read-write SQLite files are severely fragmented. This is because when the file system space is highly utilized, deletion of files can easily leave discontinuous free spaces (i.e., holes) in the file system. As a result, the file system cannot find sufficiently large free extents for new files to write or for existing files to grow.

Multi-threaded, Synchronous Writing: As mentioned previously, the Facebook application created 18 concurrent threads that wrote different SQLite files in parallel. In Ext4, files belonging to the same directory are preferably assigned to nearby spaces. Because writes of SQLite files are highly synchronous, when multiple database files grow concurrently in small size increments, small extents from different database files are interleaved with one another. This way, no matter what the file system space utilization is, it is nearly impossible for a database file to be allocated in a single, large extent.

7.2 Fragmentation Treatment: Efficacy and Limitation

Space Preallocation: Space preallocation proactively prevents file growth from creating new file fragments at the cost of wasting some storage space. Provided that applications frequently append data to files, especially those SQLite files with frequent, small size increments, space preallocation effectively reduces file fragmentation of such type of files. The preallocation unit size should be judiciously selected to minimize the amount of wasted storage space, preferably 128 KB for SQLite files. However, the efficacy of space preallocation is limited when the file system utilization is exceptionally high, because the newly allocated space for a file may not be adjacent to the last extent of the file.

Persistent Journal: Using persistent journal avoids frequent deletion of journal files and thus alleviates free space fragmentation. Between the two modes that use the persistent journal, i.e., WAL and PERSIST, we recommend WAL mode because in this mode, SQLite writes multiple transactions in the journal file and then flushes a relatively large batch of updates to the database file. This way, database files grow in large size increments and thus they are less prone to fragmentation. However, when the number of concurrently writing threads is large, multiple database files grow in parallel, and their extents are still interleaved with one another.

File Defragmentation: The fragmentation avoidance strategies mentioned above cannot eliminate file fragmentation, and therefore file defragmentation is still necessary. The efficacy of defragmentation is subject to not only file type but also space utilization. When space utilization is low, defragmentation can completely eliminate fragmentation of write-once files, e.g., `executable` and `multimedia` files. However, defragmentation is not recommended for SQLite journal files operated in DELETE mode and TRUNCATE mode, because these files are deleted or truncated upon the conclusion of transactions. When space utilization is extremely high, defragmentation can reduce but not eliminate fragmentation of files. This is because the current implementation of e4defrag may not find a sufficiently large free extent to store a fragmented file. We also showed that a state-of-the-art file defragmenter for mobile flash storage, janusd, achieves the same result as e4defrag but requires a much less amount of flash writes. Since janusd is free from data copy in flash memory, we plan to enhance it with free space defragmentation.

8 Conclusion

In this study, we examined how severe file fragmentation is in real mobile devices. Through systematic file system aging procedures, we confirmed file fragmentation emerges very quickly under daily use of smartphones, and the production of file fragmentation is highly correlated to the frequent deletion of SQLite files, the highly synchronous SQLite writing behavior, and a high file system space utilization. We also evaluated how fragmentation negatively impacted user-perceived latencies, and identified the increased block I/O frequency and the degraded device mapping cache efficiency were the roots of the poor user experiences with fragmentation. Finally, we evaluated existing methods for fragmentation treatment to understand their efficacy and limitations. In general, file defragmentation (conventional and copyless) is useful to write-once files, and space preallocation is effective on SQLite files. However, their effectiveness becomes limited when the file system utilization is extremely high. Our future work is thus directed to enhance our copyless defragmenter toward free space defragmentation for better results under a high space utilization.

References

[1] 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 *Proceedings of HotStorage*, USENIX Association, 2016.

[2] S. S. Hahn, S. Lee, C. Ji, L.-P. Chang, I. Yee, L. Shi, C. J. Xue, and J. Kim, "Improving file system performance of mobile storage systems using a decoupled defragmenter," in *Proceedings of USENIX ATC*, pp. 759–771, 2017.

[3] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pp. 280–293, ACM, 2009.

[4] S. Alsamhi and N. Rajput, "An efficient channel reservation technique for improved qos for mobile communication deployment using high altitude platform," *Wireless Personal Communications*, vol. 91, no. 3, pp. 1095–1108, 2016.

[5] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *ACM Transactions on Storage (TOS)*, p. 14, 2012.

[6] K. Lee and Y. Won, "Smart layers and dumb result: Io characterization of an android-based smartphone," in *Proceedings of the tenth ACM international conference on Embedded software (EMSOFT)*, pp. 23–32, ACM, 2012.

[7] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, "Waldio: eliminating the filesystem journaling in resolving the journaling of journal anomaly," in *Proceedings of USENIX ATC*, pp. 235–247, 2015.

[8] 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.

[9] "Micron eMMC memory, MTFC32GAKAENA-4M, datasheet." 2014.

[10] "Universal Flash Storage (UFS)." https://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs.

[11] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, p. 18, 2007.

[12] Y. K. A. Gupta and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. of ASPLOS*, pp. 229–240, ACM, 2009.

[13] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "A two-level caching mechanism for demand-based page-level address mapping in nand flash memory storage systems," in *Proceedings of RTAS*, pp. 157–166, 2011.

[14] 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*, pp. 19:1–19:7, ACM, 2016.

[15] K. A. Smith and M. I. Seltzer, "File system aging-increasing the relevance of file system benchmarks," in *ACM SIGMETRICS Performance Evaluation Review*, no. 1, 1997.

[16] G. de Nijs, A. Biesheuvel, A. Denissen, and N. Lambert, "The effects of filesystem fragmentation," in *Proc. 2006 Linux Symposium*, vol. 1, pp. 193–208, Citeseer, 2006.

[17] A. Conway, A. Bakshi, Y. Jiao, W. Jannen, Y. Zhan, J. Yuan, M. A. Bender, R. Johnson, B. C. Kuszmaul, D. E. Porter, *et al.*, "File systems fated for senescence? nonsense, says science!," in *Proceedings of FAST*, pp. 45–58, 2017.

[18] O. Kehrer, "O&O defrag and solid state drives." http://www.oo-software.com/en/docs/whitepaper/ood_ssd.pdf, 2014.

[19] W. H. Ahn, K. Kim, Y. Choi, and D. Park, "DFS: A de-fragmented file system," in *Proceedings of MASCOTS*, pp. 71–80, 2002.

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

[21] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2, pp. 21–33, 2007.

[22] "How many apps do smartphone owners use?." https://www.emarketer.com/Article/How-Many-Apps-Do-Smartphone-Owners-Use/1013309, 2015.

[23] "UI automator." https://google.github.io/android-testing-support-library/docs/uiautomator/.

[24] E. Hechtel, "How smartphones and mobile internet have changed our lives.." https://saucelabs.com/blog/how-smartphones-and-mobile-internet-have-changed-our-lives, 2016.

[25] K. Purdy, "Insufficient storage available' is one of android's greatest annoyances." https://www.itworld.com/article/2833377/mobile/insufficient-storage-available-is-one-of-android-s-greatest-annoyances-here-s-how-to-fix-it.html, 2014.

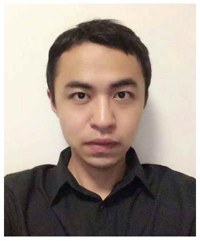
[26] A. Chen, "Why the sd market continues to focus on capacity, security and reliability." https://www.sdcard.org/press/thoughtleadership, 2016.

[27] "The darwin kernel (mirror)." https://github.com/apple/darwin-xnu.

[28] "NSPersistentStore." https://developer.apple.com/documentation/coredata/nspersistentstore.

[29] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won, "Framework for analyzing android i/o stack behavior: from generating the workload to analyzing the trace," *Future Internet*, pp. 591–610, 2013.

[30] G. R. Ganger and M. F. Kaashoek, "Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files.," in *USENIX Annual Technical Conference*, pp. 1–17, 1997.
[31] "The cat command." http://www.linfo.org/cat.html.
[32] "sqlite3 command line shell for sqlite." https://www.sqlite.org/cli.html.
[33] "Block I/O Layer Tracing: blktrace." http://linux.die.net/man/8/blktrace.
[34] L. Bouganim, B. Jónsson, and P. Bonnet, "uflip: Understanding flash io patterns," *arXiv preprint arXiv:0909.1780*, 2009.
[35] K. Kim, "Map cache design in mobile storage (SK hynix)." http://dcslab.hanyang.ac.kr/nvramos14/presentation/s9.pdf, NVRAMOS (2014).
[36] W. Jeong, H. Cho, Y. Lee, J. Lee, S. Yoon, J. Hwang, and D. Lee, "Improving flash storage performance by caching address mapping table in host memory," in *Proceedings of HotStorage*, USENIX Association, 2017.
[37] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, pp. 125–131, IEEE, 2009.
[38] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.
[39] U. Kumar, "Understanding androids application update cycles." https: //www.nowsecure.com/blog/2015/06/08/understanding-android-s-applicationupdate-cycles/, 2015.
[40] H.-Y. Chang, C.-C. Ho, Y.-H. Chang, Y.-M. Chang, and T.-W. Kuo, "How to enable software isolation and boost system performance with sub-block erase over 3d flash memory," in *Proceedings of CODES+ISSS*, pp. 1–10, ACM, 2016.



Cheng Ji received the B.E. degree from the School of Computer Science and Communication Engineering, Jiangsu University in 2011, and M.E. degree from School of Computer Science and Technology, University of Science and Technology of China in 2014, respectively. He is currently working toward the Ph.D degree in the Department of Computer Science, City University of Hong Kong. His research interests include embedded systems, non-volatile memory, and operating systems.

Li-Pin Chang received a M.S.E and a Ph.D. degree in Computer Science and Information Engineering from National Taiwan University, Taiwan, in 1997 and 2003, respectively. He is currently a Professor at Department of Computer Science, National Chiao Tung University, Taiwan. His research interest involves operating systems, storage systems, non-volatile memory, and mobile devices.

Sangwook Shane Hahn received the BS degree in computer science from Korea Advanced Institute of Science and Technology, in 2011, and the MS degree in computer science and engineering from Seoul National University, Korea, in 2013. He is currently working toward the PhD degree in computer science and engineering at Seoul National University. His research interests include storage systems, operating systems, and embedded software.

Sungjin Lee is an assistant professor at DGIST in South Korea. He earned the PhD and MS degrees in computer science and engineering from the Seoul National University in 2013 and 2007, respectively, and received the BE degree in electrical engineering from the Korea University in 2005. Before joining DGIST, he was a postdoctoral associate in the Computation Structures Group (CSG) at MIT CSAIL in Boston, MA. His current research interests include storage systems, operating systems, and system software.

Riwei Pan received B.E degree in Information Engineering from South China Normal University in 2016. He is now a research assistant in Department of Computer Science, City University of Hong Kong. His research interests include operating system and embedded system.

Liang Shi received the B.S. degrees in Computer Science from Xi'an University of Post & Telecommunication, Xi'an, Shanxi, China, in July, 2008, Ph.D. degree from University of Science and Technology of China, Hefei, China, in June, 2013. He is now the associate professor in the College of Computer Science at the Chongqing University. His research interests include flash memory, embedded systems, and emerging non-volatile memory technology.

Jihong Kim received the BS degree in computer science and statistics from Seoul National University (SNU), Korea, in 1986, and the MS and PhD degrees in computer science and engineering from the University of Washington, Seattle, in 1988 and 1995, respectively. Before joining SNU in 1997, he was a technical staff member in the DSPS R&D Center, Texas Instruments, Dallas, Texas. He is currently a professor in the Department of Computer Science and Engineering, Seoul National University. His research interests include embedded software, low-power systems, computer architecture, and storage systems. He is a member of the IEEE.

Chun Jason Xue received BS degree in Computer Science and Engineering from University of Texas at Arlington in May 1997, and MS and PhD degree in Computer Science from University of Texas at Dallas, in Dec 2002 and May 2007, respectively. He is now an Associate Professor in the Department of Computer Science at the City University of Hong Kong. His research interests include memory and parallelism optimization for embedded systems, software/hardware co-design, real-time systems and computer security.