An Empirical Study of File-System Fragmentation in Mobile Storage Systems

Cheng Ji[1], Li-Pin Chang[2], Liang Shi[3*], Chao Wu[1], Qiao Li[3], and Chun Jason Xue[1]

[1]Department of Computer Science, City University of Hong Kong, Hong Kong
[2]Department of Computer Science, National Chiao-Tung University, Taiwan
[3]College of Computer Science, Chongqing University, Chongqing, China
[*]Corresponding author: shi.liang.hk@gmail.com

Abstract

Nowadays, mobile devices have become the necessities of everyday life. However, users may notice that after a long period of usage, mobile devices will start experiencing sluggish response. In this paper, by conducting an empirical study of filesystem fragmentation on several aged mobile devices, we found that: 1) Files may suffer from severe fragmentation, and database files are among the most severely fragmented files; 2) Filesystem fragmentation does affect the performance of mobile devices, and the impact varies from devices to devices. Conventional defragmentation schemes do not work well on mobile devices because they do not consider the characteristics specific to mobile storage. Two pilot solutions were then suggested to enhance file defragmentation for mobile devices.

1 Introduction

Mobile devices, including smartphones, tablets and wearable devices, have become the necessities of daily life. For mobile devices, storage performance has been identified as a critical factor of the overall device performance [7]. However, recent studies reported that the underlying flash-based storages of mobile devices are not efficiently used when the operations of file system and database are combined [9]. In addition to the inefficient usage, another factor, fragmentation in file systems, has not been well studied in the literature.

Fragmentation in file systems is highly correlated with the space management methods of file systems. As the default file system of many mobile devices, EXT4 employs extent-based allocation and delayed allocation schemes to alleviate file fragmentation. However, our experiments show that with EXT4 file systems, SQLite database files still suffer from severe fragmentation. As we shall explain later, this result is closely related to how SQLite files grow and how free space is allocated to them.

In this work, we conducted several studies on file fragmentation in mobile devices. First of all, we examined how files are fragmented in several aged smartphones with normal user usages, such as social networking, web browsing, and instant messaging. We found that files are either severely or barely fragmented, and database files are among the mostly fragmented files. For example, on an one-year-old Google Nexus 5, the file `newsfeed_db-journal` of the Facebook application is fragmented into several pieces whose average size is only 7 KB, and its fragments are widely dispersed over a range of 1.5 GB storage space.

Next, we evaluated how file fragmentation impacts I/O performance, and identified at least two reasons for I/O latency degradation: First, accessing fragmented files results in frequent block I/Os, which accumulate a large time overhead on the I/O path. Second, file fragmentation incurs highly dispersed I/O patterns, which diminish spatial localities. To enable efficient random access with limited resource, flash storages may adopt demand-based caching of the page-level mapping table. Accessing highly fragmented files imposes high overhead on the management of the mapping cache and amplifies the latencies of block reading and writing.

Conventional disk defragmentation methods are considered harmful to flash memory because they involve intensive data copy. They do not proactively avoid file fragmentation either. To the best of our knowledge, no file defragmentation schemes have been proposed for mobile devices. Finally, we suggested two pilot solutions with the consideration of the flash management overhead and file access characteristics in mobile storage systems.

This study makes following contributions: 1) We identified that file fragmentation is a serious problem in mobile devices, and SQLite database files were among the mostly fragmented files. 2) We evaluated how application performance of mobile devices can be affected by fragmentation in mobile devices; 3) We suggested two pilot solutions to optimize file defragmentation in mobile devices.

2 Background and Related Work

2.1 Fragmentation in File Systems

File systems, including EXT4, FAT, and even the log-structured file system, F2FS, suffer from fragmentation. Fragmentation in file systems is caused by the aging problem [12], and it emerges when the file system cannot find continuous free space for files. Take EXT4 as an example, which is the default file system since Android version 4.0.4, it suffers from three types of fragmentation: single file fragmentation, relevant file fragmentation, and free space fragmentation [11]. These types of fragmentation did affect the performance of hard disk drives [4]. However, there is little work targeting on how they affect the performance of mobile storage systems.

2.2 Conventional Defragmentation

Previous studies propose to restore the continuity of fragmented files through re-locating file fragments to a continuous free space. The DFS [2] file system proactivaly performs file defragmentation when severe fragmentation is detected. EXT4 employs a user-mode tool, `e4defrag` [11], to defragment files in an on-demand manner. Defragmenting Solid-State Disks (SSDs) of desktop computers was reported having little effect and even considered harmful to SSD lifetime [6]. However, applications in smartphones exhibit very unique file accessing behaviors, and the design of flash storage for smartphones is resource conservative. We observed that fragmentation is a serious problem in smartphones and it noticeably affects the performance of file accessing on flash storage.

3 Fragment Measurement Setup

In this section, we present the setups for our fragmentation study including the mobile platforms, measurement softwares and benchmarks.

3.1 Smartphone Platforms

Our study is based on four Android phones, including Google Nexus 5, Google Nexus 6, Huawei Honor 6, and Huawei Ascend P7. These phones were from randomly selected people to avoid potential biases. These phones had underwent at least six months of daily use of their owners. The use patterns of these phones involved common Android user activities, including web surfing, sending/receiving emails, social networking, instant messaging, and taking pictures. These activities were based on popular Android applications, including Facebook, Twitter, WeChat, Chrome, Gmail, Google Earth, and the built-in applications like Camera. Observations and experiments regarding file fragmentation were conducted on the Android `data` partition, which was formatted in EXT4. The `data` partition sizes of the selected phones were 26.8, 26.0, 11.6 and 11.8 (GB), respectively. Upon the arrivals of these phones, the file system utilizations of their `data` partitions were 93%, 57%, 44%, and 90%, respectively.

3.2 Measurement Softwares and Application Benchmarks

`e4defrag` is used for inspecting the file fragmentation and performing file defragmentation if needed. To manually execute SQL statements on a SQLite database, we use the command-line utility `sqlite3`. To study the I/O pattern of the fragmented files, we use `blktrace` to collect block-level traces for device I/O and use `MOST` [5] to identify the source file of each I/O request.

Several popular Android applications, including Facebook, Twitter, WeChat and Google Earth are used to assess the impacts of fragmentation. With these applications, we read news feed, chat with friends, and view online satellite maps for one minute, respectively.

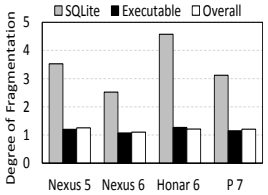
4 Characteristics of Fragmentation

In this section, we characterize fragmentation in Android devices. Degree of Fragmentation (DoF) is used to represent the degree of single file fragmentation. DoF is computed by Equation 1, where n_{ext} is the current number of extents and n_{lowest_ext} is the ideal (smallest) number of extents of the file, respectively. The larger the DoF is, the more serious the fragmentation is. In the following, fragmentation is analyzed against file types and file system utilizations.

$$DoF = n_{ext} / n_{lowest\_ext} \quad (1)$$

Fragmentation vs. File Types: We examined the fragmentation of different types of files using the smartphones described in Section 3.1. Figure 1 shows the DoF of database files (with extensions `.db`, `.db-journal`, and `.wal`), executable files (with extensions `.apk`, `.dex`, `.odex`, and `.so`) and all files. We found that the file fragmentation has two extremes: most of the executable files are barely fragmented, while the database files, especially those with the `.db` and `.db-journal` extensions, are severely fragmented.

We are particularly interested in the fragmentation of database files, because they contribute to about 70% of all block writes in Android devices [9]. We examined the single file fragmentation of database files from a selected set of Android applications on the Google Nexus 5. As shown in Figure 2(b), almost all the database files are severely fragmented. Particularly, each of the database file of Facebook is fragmented into four pieces on average (47 files have 162 fragments). Most fragmented files are those with extensions `.db` and `.db-journal`, and their fragment sizes range from tens to hundreds of kilobytes. This is because when database files are appended with new data, record by record, an aged file system can



(a)

app	Files	Fragments	Average Size
Facebook	47	162	26 KB
Twitter	14	83	55 KB
Wechat	68	237	19 KB
Google Earth	2	46	4408KB
Messenger	27	107	15 KB
QQ	52	394	20KB
Youtube	18	44	8KB
...	...	...	...
Total	483	1704	185KB

(b)

Figure 1: (a) The degree of fragmentation (DoF) of SQLite database files, executable files, and all files. (b) Fragment count of SQLite database files of a selected set of applications on an aged Google Nexus 5. The total file number, the total number of fragments, and the average fragment sizes were reported.

barely find continuous free space for them to grow, especially when multiple database files in the same directory are growing in parallel. Similar results are obtained on the other three smartphones.

Fragmentation vs. File System Utilizations: Even when the space utilization of file system is low or moderate, database journal files are still prone to fragmentation, regardless of any specific Android device. To verify this, we performed a factory reset on the Nexus 5 described in Section 3.1 to clear up the file system. After the reset, we re-installed all applications back and manually ran the applications for one hour. By the end of this test, the file system utilization was only 19% (previously 93%). However, we found that many database files, especially the `.db-journal` files, were still severely fragmented as they were in aged file systems. Take Facebook as example, where the SQLite library employed the default `DELETE` mode for database journaling, the DoF of Facebook's database files was 4, which was not lower than the DoF shown in Figure 1(a), and the fragmented pieces of several files (9 out of 48) were dispersed over 1 GB storage space. This is closely related to the way how database files grow, as explained in the previous paragraph. Based on our observations, the way how SQLite writes database files plays a more important role than application-level usage patterns do in terms of fragmentation production. Because most of Android apps employ SQLite for structural data management, as Figure 1 shows, the DoF of different phones appear quite similar.

5 Evaluation of Fragmentation

In this section, we investigate how file fragmentation impact the file access performance in Android devices.

5.1 Overall Problem

We performed experiments to illustrate the performance degradation caused by fragmentation on the Huawei P7. We emulated an aged file system by creating large files ($\geq$ 100 MB) and small files ($\leq$ 100 KB) alternatively

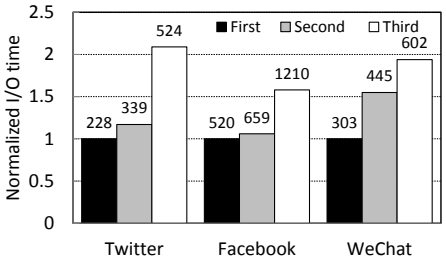


Figure 2: Normalized total I/O time in replaying file-system call traces on Ascend P7. The number above each bar is the file fragment counts at the end of the replay.

until the file system is completely full, and then randomly deleting some small files until the file system utilization dropped to 95%. Specifically, during an instance of the aging process, 86 large files of 9.4 GB and 24,000 small files of 1.8 GB were created, and about 10,000 small files were then deleted. To ensure that our experiments are repeatable, we cleaned up all previously created files and performed the same aging procedure again before each experiment. We then used Mobibench [5] to replay the file-system traces collected from Twitter, Facebook, and WeChat. Figure 2 shows that, during the trace replays, the replay time increased as the file fragmentation got worse. This observation confirms that file fragmentation affects the I/O performance in mobile devices.

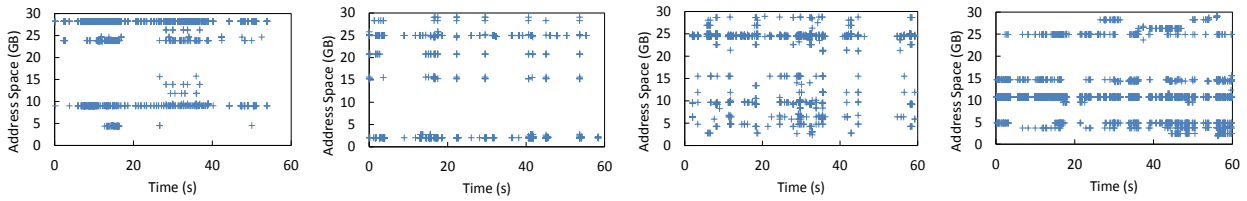
5.2 Increased Block I/O Frequency

Accessing fragmented files will result in a high I/O frequency. We used Google Earth to examine how fragmentation increases the I/O frequency, and show the impact of an increased I/O frequency on I/O performance. We aged the Google Nexus 5 in the same way as that described in Section 5.1. We then viewed maps online, and 129 MB of new map resources were downloaded and appended to the file `mirth_cache.db`. At the end of map viewing, we found that the database file was fragmented into 1605 pieces. Next, we manually performed an SQLite query to scan the entire database tables in the file. Block level trace was collected during the scan. Finally, we defragment the file and performed the same query on the defragmented file again. Results show that, after the database file is defragmented, the total block I/O count is reduced by 13%, and the elapsed time of the database table scan is reduced by 10%. The results confirm that, accessing fragmented files introduces a high overhead on the I/O path of mobile storage system.

5.3 Dispersed Block I/O Pattern

5.3.1 Characteristics of Dispersed I/O

Since database files are among the mostly fragmented files and they contribute to the majority of all block writes, we are interested in the block write patterns associated with the database files. We ran the benchmarks



(a) Facebook (b) Twitter (c) WeChat (d) Google Earth

Figure 3: I/O pattern of writes on fragmented database files.

(described in Section 3.2) on the aged Google Nexus 5, and collected the block I/O traces. We extracted the block I/Os associated with database files only. Figure 3 shows that the block writes on the fragmented database files were widely dispersed over the entire address space.

5.3.2 Impacts of Dispersed I/O

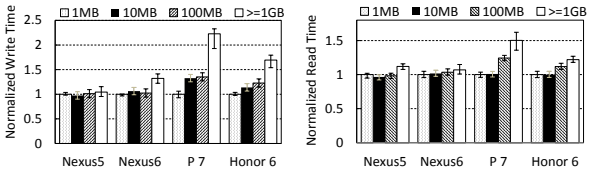
As fragmented database files contribute to random block writes, we are interested in how bad the performance of random writes will be on different smartphones. For comparison, on each of the phones listed in Section 3.1, we performed 10,000 synchronous file writes on an 1 GB file that had been defragmented beforehand. The synchronous file writes were randomly distributed with a region in the file, and the region size varied from 1 MB to 1 GB.

Figure 4(a) shows that, on the P7 and Honor 6, random writes in large regions are significantly slower than in small regions. As the write region is enlarged, the average write latency is increased by 130% and 69% on the P7 and Honor 6, respectively. On the Nexus 5 and Nexus 6, the write latency is less sensitive to the write region size, and noticeable degradation appeared only when the write region was as large as 1GB.

We further performed a similar test to evaluate the performance of dispersed reads. Interestingly, Figure 4(b) also indicates that random reads are unfavorable to flash storage. As pointed out in [3], random writes should be confined to a small area of disk space; otherwise, dispersed random writes can cause serious write performance degradation due to the amplified garbage collection overhead. However, as flash reads do not incur garbage collection, the results in Figures 4(a) and 4(b) suggest that dispersed I/O patterns impose some overheads on flash management other than garbage collection.

5.3.3 Mapping Cache Management Overhead

To deploy page-mapping FTL with a limited mapping structure size, demand-based page-level mapping scheme [1] is proposed to selectively cache an active portion of the entire mapping table. Map caching is a popular design option for flash storage in mobile devices [8], including eMMCs in smartphones. Demand-based map caching exploits spatial locality by prefetching multiple mapping entries of a set of continuous logical pages



(a) Random sync writes (b) Random reads

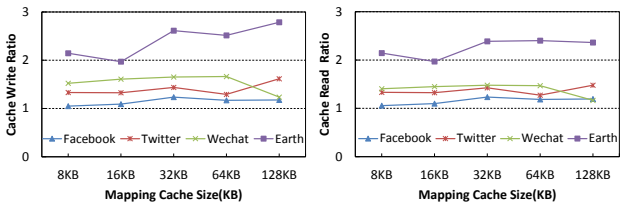
Figure 4: Normalized elapsed time when changing the size of I/O region. I/O size: 4 KB, I/O number: 10000, direct IO mode is used in read.

[10]. The poor performance of the dispersed I/Os may be related to the management overhead of the mapping cache. If the access pattern appears random, the prefetching mechanism not only has little effect but also imposes a high pressure on the mapping cache.

Cache Simulation: To verify our theory, we implemented a page-mapping FTL simulator with a demand-based mapping cache. The simulated flash page size is 4KB, which consists of 1024 mapping entries. The unit for cache fetching and evicting is one page, and the cache replacement policy is the LRU algorithm. The simulation settings are based on the mostly common features that we learned from industry. We believe that our simulation results can reflect the performance characteristics of real eMMC mapping cache designs.

Our simulation involves two sets of block I/O traces from the same applications. The first set of traces were those collected in Section 5.3.1, and they were collected under severe file fragmentation. We produce the second set of traces based on the first set of traces, as follows: First, we defragmented the database files (which were severely fragmented) and recorded the block migration history during the defragmentation. Second, based on the block migration history, we converted the old block traces into a set of new traces. The old (dispersed) and new (defragmented) trace sets were both replayed.

We measured the increases in translation page reads and writes due to file fragmentation. Cache Write Ratio is the ratio of the total number of translation page writes with fragmentation to that without fragmentation. Cache Read Ratio is defined accordingly. Figure 5 shows that Cache Write (and Read) Ratios are larger than 1 under all applications, and therefore the mapping cache incurs a much higher overhead under the dispersed I/O patterns than it does under the defragmented patterns.



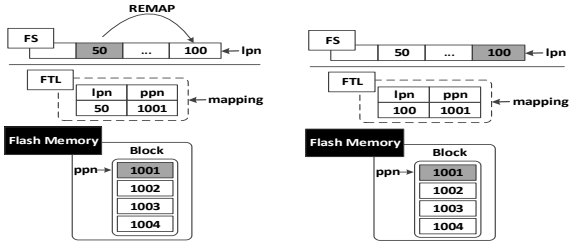
(a) Cache Write Ratio (b) Cache Read Ratio

Figure 5: Amplifications of translation page writes and translation page reads.

6 Pilot Solutions

Conventional defragmentation methods are based on data copying and may unnecessarily wear flash memory in mobile storage. They do not proactively avoid fragmentation either. We suggest two pilot solutions to the fragmentation problem in mobile storage.

Defragmenting Flash Storage: We suggest to exploit the mapping scheme in the FTL to speed up the defragmentation process without actual data copying.



(a) Before defragmentation (b) After defragmentation

Figure 6: Defragmentation based on FTL re-mapping.

A new block command `REMAP` is introduced. As shown in Figure 6, by adjusting the FTL mapping information, the logical addresses of two pieces of data are exchanged for file defragmentation. The implementation of `REMAP` is similar to that of `TRIM` in EXT4. The defragment tool (i.e., `e4defrag`) makes `ioctl()` requests to inform the file system of moving all data from a fragmented file to a contiguous donor file. The file system then prepares corresponding `REMAP` commands, which will be flushed to the flash storage on transaction committing. Upon receiving a `REMAP` command, the FTL adjusts the mapping information accordingly. During this process, file system metadata does not employ `REMAP` and they must undergo data copy operations.

Proactive Fragmentation Avoidance: Based on the accessing behaviors of files, we suggest to proactively avoid fragmentation by allocating sufficient continuous space for files to grow. To minimize the impact on the current space allocation method in file systems, we suggest to perform fragmentation avoidance on database files only, because they are most likely to be fragmented and their sizes are usually small. Applications can pre-allocate free space for SQLite database files via the posix_fallocate() system call. For example, most `.db-journal` files are smaller than 100KB, which can be a good choice of the pre-allocation size.

7 Conclusion

In this study, we examined how severe file fragmentation is in real Android devices. We found that SQLite database files are among the most severely fragmented files, and also identified that fragmentation really affects mobile device performance: frequent block I/Os increase the time overhead on the I/O path, and dispersed I/O patterns impose a high pressure on the mapping-cache management. Two pilot solutions were suggested to enhance conventional disk defragmentation methods by considering the characteristics of mobile storages.

8 Acknowledgments

We are grateful to our shepherd Beomseok Nam and the anonymous reviewers for their insightful feedbacks. This work is supported by the Fundamental Research Funds for the Central Universities (106112016CDJZR185512 and 106112014CDJZR185502), NSFC (61402059 and 61572411), and National 863 Programs 2015AA015304.

References

[1] A. Gupta, Y. K., and Urgaonkar, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. of ASPLOS* (2009), ACM.

[2] Ahn, W. H., Kim, K., Choi, Y., and Park, D. DFS: A defragmented file system. In *Proc. of MASCOTS* (2002), IEEE.

[3] Bouganim, L., Jónsson, B., and Bonnet, P. uflip: Understanding flash io patterns. *arXiv preprint arXiv:0909.1780* (2009).

[4] de Nijs, G., Biesheuvel, A., Denissen, A., and Lambert, N. The effects of filesystem fragmentation. In *Proc. 2006 Linux Symposium* (2006), vol. 1, pp. 193–208.

[5] Jeong, S., Lee, K., Hwang, J., Lee, S., and Won, Y. Framework for analyzing android i/o stack behavior: from generating the workload to analyzing the trace. *Future Internet* (2013).

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

[7] Kim, H., Agrawal, N., and Ungureanu, C. Revisiting storage for smartphones. In *Proc. of FAST* (2012), pp. 273–285.

[8] Kim, K. Map cache design in mobile storage (SK hynix). http://dcslab.hanyang.ac.kr/nvramos14/presentation/s9.pdf, NVRAMOS (2014).

[9] Lee, K., and Won, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proc. of EMSOFT* (2012).

[10] Qin, Z., Wang, Y., Liu, D., and Shao, Z. A two-level caching mechanism for demand-based page-level address mapping in nand flash memory storage systems. In *RTAS* (2011).

[11] Sato, T. ext4 online defragmentation. In *Proceedings of the Linux Symposium* (2007), vol. 2, pp. 179–86.

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