Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter

Sangwook Shane Hahn, *Seoul National University;* Sungjin Lee, *Daegu Gyeongbuk Institute of Science and Technology;* Cheng Ji, *City University of Hong Kong;* Li-Pin Chang, *National Chiao-Tung University;* Inhyuk Yee, *Seoul National University;* Liang Shi, *Chongqing University;* Chun Jason Xue, *City University of Hong Kong;* Jihong Kim, *Seoul National University*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahn

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter

Sangwook Shane Hahn, Sungjin Lee[†], Cheng Ji[*], Li-Pin Chang[‡],
Inhyuk Yee, Liang Shi[§], Chun Jason Xue[*], and Jihong Kim
Seoul National University, [†]Daegu Gyeongbuk Institute of Science and Technology,
[*]City University of Hong Kong, [‡]National Chiao-Tung University, [§]Chongqing University

Abstract

In this paper, we comprehensively investigate the file fragmentation problem on mobile flash storage. From our evaluation study with real Android smartphones, we observed two interesting points on file fragmentation on flash storage. First, defragmentation on mobile flash storage is essential for high I/O performance on Android smartphones because file fragmentation, which is a recurring problem (even after defragmentation), can significantly degrade I/O performance. Second, file fragmentation affects flash storage quite differently than HDDs. When files are fragmented on flash storage, the logical fragmentation and the physical fragmentation are decoupled and a performance degradation mostly comes from logical fragmentation. Motivated by our observations, we propose a novel defragger, janus defragger (janusd), which supports two defraggers, janusdL for a logical defragger and janusdP for a physical defragger. JanusdL, which takes advantage of flash storage's internal logical to physical mapping table, supports logical defragmentation without data copies. JanusdL is very effective for most fragmented files while not sacrificing the flash lifetime. JanusdP, which is useful for physically fragmented files but requires data copies, is invoked only when absolutely necessary. By adaptively selecting janusdL and janusdP, janusd achieves the effect of full file defragmentation without reducing the flash lifetime. Our experimental results show that janusd can achieve at least the same level of I/O performance improvement as e4defrag without affecting the flash lifetime, thus making janusd an attractive defragmentation solution for mobile flash storage.

1 Introduction

When a file system becomes highly fragmented, it has to allocate multiple split storage areas, i.e., extents [1], for a single file more frequently. In an HDD-based file system, accessing such a highly-fragmented file degrades the performance significantly due to the increased time-consuming seek operations. In order to mitigate the performance impact caused by file fragmentation, many file systems recommends the periodical execution of the defragmentation utility (e.g., every week) [2-6].

Unlike for HDD-based file systems, defragmentation is generally not recommended for flash-based file systems [7-13]. Since flash storage does not require seek

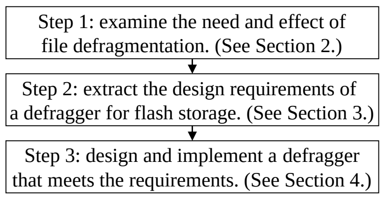

Step 1: examine the need and effect of file defragmentation. (See Section 2.)
Step 2: extract the design requirements of a defragger for flash storage. (See Section 3.)
Step 3: design and implement a defragger that meets the requirements. (See Section 4.)

Fig. 1: A summary of the key steps in our investigation.

operations, it is believed that the effect of defragmentation on the file system performance is rather negligible for flash storage. Furthermore, since a large number of files need to be copied during defragmentation, frequent defragmentation can affect the limited lifetime. However, this negative view toward flash defragmentation has been widely accepted without a proper validation study. The main goal of this paper, therefore, is to investigate the file fragmentation problem on mobile flash storage in a systematic and comprehensive fashion. Fig. 1 summarizes the key steps of our investigation study.

Since previous studies (e.g., [22]) have shown that files can be severely fragmented on mobile flash storage, in our study, we start with two key questions related to the effect of file defragmentation (step 1 in Fig. 1): 1) when fragmented files are defragmented, how much I/O performance is improved? and 2) how long does the effect of file defragmentation last? Unlike a common misconception on flash defragmentation, our evaluation study showed that I/O performance of flash storage can be significantly improved by defragmentation. For example, when fragmented files were defragmented, the average app launching time, which is an important user-perceived performance metric on smartphones, can be improved by up to 52% over the fragmented files.

Although fragmented files can degrade the I/O performance, if the effect of file defragmentation can last for long time (e.g., several months), a conventional defragmentation tool will be sufficient. However, our evaluation study indicated that file fragmentation may recur in a short cycle, around a week, even after full file defragmentation on smartphones. One main cause of recurring file fragmentation was frequent automatic app updates on smartphones. Since many popular apps tend to be updated very frequently (e.g., every 10 days [28]), the effect of file defragmentation quickly disappears.

When file defragmentation is repeatedly required, a conventional defragger such as e4defrag may not be an appropriate solution for flash storage because it requires a large amount of data copies during defragmentation, thus seriously affecting the flash lifetime. For example, if we invoke e4defrag every week as suggested from our evaluation study, it might reduce the flash lifetime by more than 10%. Therefore, in order to maintain high I/O performance in a sustainable fashion, we need a different approach to the defragmentation problem for mobile flash storage, so that the impact of file defragmentation on the flash lifetime is less adverse.

The key insight behind janus defragger (janusd) comes from our investigation on the characteristics of file fragmentation in flash storage (step 2 in Fig. 1). Our study showed that file fragmentation affects flash storage quite differently from HDDs. In HDDs, when a (logical) file is highly fragmented, its physical layout is fragmented similarly with many isolated physical fragments. That means, logical fragmentation at the file system and physical fragmentation at the storage medium level are highly *correlated*. On the other hand, in flash storage, there is no physical counterpart at the storage medium level which is strongly correlated with logical fragmentation at the file system. For example, unlike HDDs where a degree of logical fragmentation directly affects the I/O performance at the storage medium level, the I/O performance at the storage medium level in flash storage is largely decided by an average degree of the I/O parallelism during I/O operations [16-21]. As will be explained in Section 3, since the average degree of the I/O parallelism for accessing a file is not correlated with the degree of logical fragmentation of the file, file fragmentation in flash storage occurs in a *decoupled* fashion between the logical space and the physical space. (In this paper, we call that a file `foo` is *physically fragmented* when the degree of the I/O parallelism in accessing `foo` is limited.)

In order to understand the impact of decoupled fragmentation on I/O performance, we evaluated the performance impact of file fragmentation on the entire mobile I/O stack layers. As expected, because of a high degree of the I/O parallelism at the storage medium level, only a small number of (unlucky) files were stored in a severely skewed fashion, limiting their I/O parallelism levels significantly. That is, regardless of how files were logically fragmented, their I/O performance at the storage medium level did not change much. On the other hand, logically fragmented files significantly increased processing times in the block I/O layer and the device driver because of a large increase in the number of block I/O requests. Therefore, the minimum requirement for a flash defragger would be to defragment the logical space effectively. Furthermore, since flash files are fragmented in a decoupled fashion, an ideal flash defragger needs to support an *independent* physical defragger as well. The physical defragger is necessary because a logical defragger cannot even identify physically fragmented files.

Motivated by the above requirements on a defragger for mobile flash storage, we propose a novel decoupled defragger, janusd, which consists of two defraggers, janusdL for a logical defragger and janusdP for a physical defragger (step 3 in Fig. 1). JanusdL, which takes advantage of flash storage's internal logical to physical mapping table, supports logical defragmentation without reducing the flash lifetime by avoiding explicit data copies. JanusdP, which independently operates from janusdL, works like a conventional defragger with data copies. Since the I/O performance of flash storage is dominated by logical file fragmentation, janusdL works very well for most fragmented files without affecting the flash lifetime. On a rare occasion when a file is physically fragmented, janusdP is invoked to restore the degraded file performance.

In order to validate the effectiveness of the proposed janusd technique, we have implemented janusd on an emulated mobile flash storage, `simeMMC` and `simUFS`. (`SimeMMC` and `simUFS`, which are based on an extended Samsung 843T SSD which supports host-level FTLs, are configured to effectively simulate the bandwidth of eMMC and UFS devices [14, 15], respectively.) Our experimental results show that janusd significantly improves the I/O performance of mobile flash storage. For example, janusd can reduce the app launching time by up to 53%, achieving an equivalent I/O performance improvement as e4defrag. However, janusd requires a less than 1% of data copies over e4defrag, thus making it an attractive defragmentation solution for flash storage. Furthermore, janusdL alone achieves about 97% of the janusd's performance level for most files.

The remainder of this paper is organized as follows. In Section 2, we report our key findings through our evaluation study of real-world file fragmentation on Android smartphones. Section 3 describes decoupled fragmentation in flash storage and explains needs for both logical and physical defraggers. A detailed description of janusd is given in Section 4. Experimental results follow in Section 5, and related work is summarized in Section 6. Finally, Section 7 concludes with future work.

2 File Fragmentation: User Study

In this section, we empirically investigate how file I/O performance is affected by file fragmentation on flash storage using 14 smartphones in use. In particular, we examine how quickly file fragmentation occurs again after defragmentation and how much I/O performance is affected by different defragmentation intervals.

2.1 Evaluation Study Setup

For our study, we collected 14 used Android smartphones. In order to avoid possible bias, we have se-

Table 1: File system utilizations of 14 smartphones.

50-59%	60-69%	70-79%	80-89%	90-99%
S5, GP	**S3**, G5	**N5**	**N6**, T2, T5, Z1, Z3	**S6**, I2, T3, T4

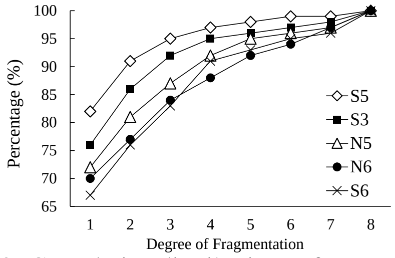


Fig. 2: Cumulative distributions of DoF values.

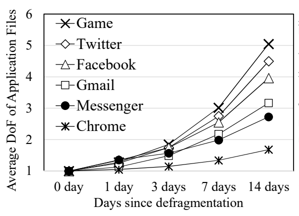
lected these smartphones from five different manufacturers with at least six month's real use. 14 users, like most other smartphone users, heavily used popular Android applications such as `Chrome, Messenger, Gmail, Facebook, Twitter` and `Game`. Table 1 divides 14 smartphones[1] into 5 categories based on the file system utilization. (In the rest of this section, we report the evaluation results on five representative smartphones, S5, S3, N5, N6 and S6, which were chosen from each utilization category.) We inspected file fragmentation on the `data` partition only because the `data` partitions occupied most of the total storage space available and most I/O operations occur in the `data` partition.

For our study, we used the degree $DoF(x)$ of fragmentation of a file `x`, which is defined as the ratio of the number of extents allocated to the file `x` to the ideal (i.e., smallest) number of extents needed for the file `x`. For example, if an 1-GB file `foo` in Ext4 were allocated to 24 extents, DoF(`foo`) would be 3 (i.e., 24/8), because `foo` would have required at least 8 extents even when `foo` was contiguously allocated. (A single extent can cover up to 128 MB in Ext4.) The large DoF value means that the file is highly fragmented.

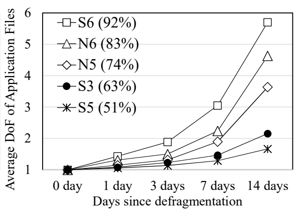
2.2 Degree of File Fragmentation Analysis

We first examined DoF values of files in the `data` partition of the five smartphones using `e4defrag`, and Fig. 2 shows cumulative distributions of DoF values on the five smartphones. As reported in other investigations such as [22], our inspected smartphones exhibited similar characteristics on file fragmentation. Fragmented files accounted for between 14% and 33% of all files. In particular, on N5, 717 files among its 2,704 files were fragmented. Furthermore, 476 files were fragmented with their DoF values larger than 2. When the file system space was highly utilized, the number of fragmented files tends to be large. For example, on S6, having the highest file system utilization, 33% of its files were fragmented.

[1]14 phones include Nexus 5 (N5), 6 (N6), Galaxy S3 (S3), S5 (S5), S6 (S6), Note 2 (T2), Note 3 (T3), Note 4 (T4), Note 5 (T5), Xperia Z1 (Z1), Z3 (Z3), Optimus G Pro (GP), G5 (G5) and Vega Iron 2 (I2).



(a) Six applications on N6.



(b) `Twitter` on five smartphones.

Fig. 3: The average DoF value of application files.

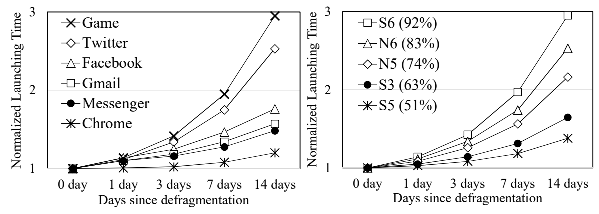
2.3 File Fragmentation Recurrence

Since our target smartphones have never been defragmented before, the results shown in Fig. 2 are interesting but somewhat expected. A more critical question for our study was to find out how soon file fragmentation recurs after full file defragmentation. If the recurrence interval of file fragmentation were quite large (say, several months), an existing defragmentation would be sufficient for mobile flash storage as well.

In order to understand file fragmentation recurrence (as well as others), after defragmenting all the files using `e4defrag`, we collected a daily snapshot of each smartphone for the subsequent two-week interval using a custom data collection app. Our snapshot data include DoF values of files and app launching times, Fig. 3(a) shows the changes in the average DoF values of the files associated with six popular applications, `Chrome`, `Messenger`, `Gmail`, `Facebook`, `Twitter` and `Game`, on N6. As shown in Fig. 3(a), file fragmentation recurred quickly after the full file system defragmentation. For most applications on N6, file fragmentation occurs again in a week since the full defragmentation. Fig. 3(b) shows the changes in the average DoF values of the files associated with `Twitter` on the five smartphones with different file system utilizations. The recurrence interval of file fragmentation was proportional to the file system utilization. For example, on the seventh day after the full file system defragmentation, the average DoF value of the `Twitter` files reached 1.86 and 3.04 for 70% and 90% of file system utilization, respectively. Even though only the DoF values of `Twitter` files are presented here, we had similar observations on the files of the other applications [42].

Our observation strongly suggests that file fragmentation is a recurring problem in smartphones, especially when the file system utilization is high[2]. In the following subsections, we shall show that file fragmentation negatively impact on user experience, but regular file defragmentation is harmful to flash storage lifetime. The proposed `janusd` technique is novel in that these two conflicting phenomena are resolved in a satisfactory fashion.

[2]One of the reasons for a short recurrence interval is frequent app updates which automatically invoked in background when a smartphone is connected to a Wi-Fi environment. Since popular apps such as `Twitter` are reported to be updated, on average, every 7 days [29], when the file system utilization is high, newly installed apps are very likely to experience severe file fragmentation.



(a) Six applications on N6. (b) `Twitter` on five smartphones.

Fig. 4: Changes in app launching times.

2.4 Impact on User Experience

File fragmentation can negatively impact on the smartphone user experience due to degraded I/O performance. For example, the launching of an application involves reading a set of files, including executables, libraries, and data files. This procedure creates a user-perceived latency because the user has to wait until all the required files have been loaded from flash storage. We define the launching time of an application to be the time interval between the time when the application icon is touched and the time when all graphical user interface components are displayed for the next user interaction.

Fig. 4(a) shows the launching time of the six popular applications on N6 and Fig. 4(b) depicts the launching time of `Twitter` on five smartphones with different file system utilizations. The launching time noticeably degraded as the day count increased, especially with the high file system utilization. For example, compared to the launching time right after the full file system defragmentation, the launching time of `Twitter` on the seventh day was already 1.6 times longer when the file system utilization was 70%, and the launching time was amplified to two times longer when the file system utilization was 90%. This result indicates that the recurring file fragmentation can highly impact the quality of user experience in a short period of time.

2.5 Impact on Flash Memory Lifetime

Because file fragmentation is a recurring problem, regular file defragmentation might be necessary to maintain satisfiable user experience. In fact, weekly file defragmentation is recommended by many defragmentation tools [25, 26]. However, conventional file defragmentation is based on data copies, which increases the wear in flash memory. We performed full file system defragmentation with different frequencies, including a daily basis and a weekly basis, under the emulated application update behaviors. Fig. 5 shows the total write traffic contributed by file defragmentation measured by the built-in Linux block I/O tracing tool blktrace. Surprisingly, the amount of data copies during file defragmentation was fairly large. For example, defragmenting files on the third day involved 1.8 GB of data copies under a 70% file system utilization, and this number increased to 5.76 GB if the file system utilization was 90%. If file defragmentation was performed in a weekly manner, the amount of data copies reached up to 9.53 GB.

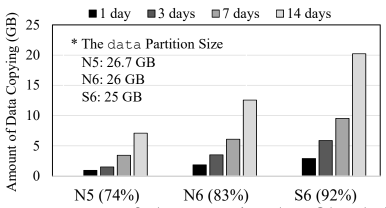


Fig. 5: The amount of data copies by file defragmentation with different defragmentation periods.

The extra data copies negatively impacts on flash memory lifetime. This problem is further exaggerated by the deteriorated flash endurance due to the introduction of multilevel cells. Specifically, the program-erase cycle (PE cycle) limit of TLC NAND is as low as 300 PE cycles. The `data` partition of the S6 is 25 GB, and weekly file defragmentation costs every flash block (9.53 GB/week $\times$ 4 weeks)/25 GB$\approx$1.5 extra PE cycles per month. In the typical smartphone life cycle of two years, weekly file defragmentation introduces 36 extra PE cycles to every block, and thus the flash lifetime is degraded by more than 10%. This significant lifetime reduction highly discourages the use of conventional copy-based file defragmentation tools on flash storage.

3 File Fragmentation: Under The Hood

In order to develop a flash-aware file defragmentation tool which does not have a negative effect on the flash lifetime, we performed a detailed characterization study of file fragmentation on flash storage.

3.1 Decoupled Fragmentation on Flash

Since flash storage works quite differently from HDDs at the storage medium level, before our study, we redefined the concept of physical fragmentation for flash storage.

Since flash storage is composed of a group of parallel I/O units (e.g., multiple flash memory channels/planes) and each I/O unit can support random access, a conventional definition of physical data sequentiality on hard drives does not make much sense to flash storage. In order to better reflect the effect of file fragmentation on I/O performance in flash storage, we associate two metrics, $DoF^L(\mathtt{x})$ and $DoF^P(\mathtt{x})$, for a file `x`, where $DoF^L(\mathtt{x})$ and $DoF^P(\mathtt{x})$ represent the degrees of logical fragmentation and physical fragmentation, respectively. For the logical DoF value, $DoF^L(\mathtt{x})$, of a file `x`, we use $DoF(\mathtt{x})$ as defined in Section 2.1. Since the I/O performance at the flash device level is largely determined by a degree of the I/O parallelism while accessing the file `x`, not the number of split extents as in HDDs, we define the physical DoF value, $DoF^P(\mathtt{x})$, of a file `x` as (1 - $DoP(\mathtt{x})$). $DoP(\mathtt{x})$, which indicates the effective degree of the I/O parallelism for accessing the file `x`, is computed as the ratio of the average degree of the I/O parallelism for accessing the file `x` sequentially to the maximum degree of the I/O parallelism supported by a flash storage sys-

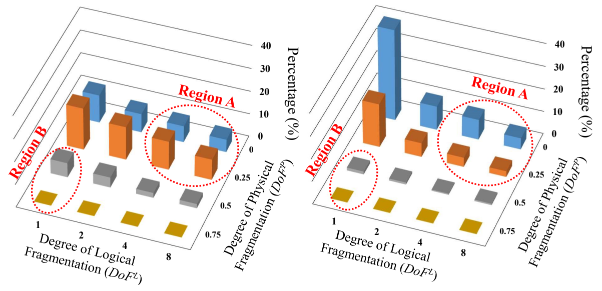


Fig. 6: A snapshot distribution of files classified based on their DoF^L values and DoF^P values.

tem. When a flash storage system can support up to M I/O operations at the same time, if, on average, n operations were supported in parallel while accessing `foo`, DoP(`foo`) is n/M. Therefore, DoF^P(`x`) becomes 0 when the file `x` was accessed under the maximum I/O parallelism. As the effective degree of the I/O parallelism drops, DoF^P(`x`) approaches (1- $1/M$).

In order to understand how logical fragmentation and physical fragmentation interact with each other in flash storage, we measured how DoF^L and DoF^P values change from the Ext4 file system after aging Ext4 with simulated one-year and one-week workloads. Since we need to collect DoF^P values, we used a mobile flash storage emulator (see Section 5).

Fig. 6 shows the distributions of DoF^L and DoF^P values after aging Ext4 with simulated one-year and one-week workloads, respectively. The results indicate that logical and physical fragmentation are highly decoupled. For example, the files in Region A suffered from high degrees of logical fragmentation but their degrees of physical fragmentation were quite low. On the other hand, surprisingly, there were still a few files in Region B that were barely fragmented at the logical space but suffered from high degrees of physical fragmentation.

Decoupled logical and physical fragmentation is mainly attributed to the high degree of the I/O parallelism available in flash storage as well as the extra indirection layer in flash storage for logical to physical mapping. Logical fragmentation and physical fragmentation impose different impacts on I/O performance. Specifically, logical fragmentation amplifies the overhead in the system software I/O stack due to the increased I/O frequency, while physical fragmentation degrades the I/O parallelism in flash storage. Defragmentation only at the logical or physical level may not produce the optimal I/O performance. For example, even though a file has been defragmented at the file system level, it dost not guarantee that the file is accessed through the maximum I/O parallelism inside of flash storage.

Conventional defragmentation tools cannot perform physical defragmentation for flash storage because the host does not have direct access to flash channels. In addition, these tools are not aware of the existing indirection layer inside of flash storage, which is useful to modify the logical layout of files without physical data copies. We believe that the firmware of flash storage must be adequately involved during the defragmentation process. As shown in Fig. 6, the majority of file fragmentation is affiliated with logical fragmentation. While it is possible to perform copyless defragmentation for logically fragmented files (the files in Region A), data copies are still necessary to re-distribute data among flash channels for physical defragmentation. Fortunately, as shown in Fig. 6(a) and 6(b), the files with $DoF^P \geq 0.5$ contribute to no more than 20% of all files. In other words, physical defragmentation will be performed only for absolutely needed cases to prevent the extra data copies which will reduce the flash memory lifetime.

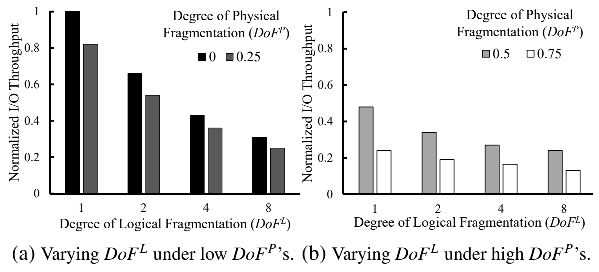
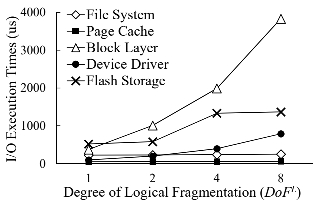


Fig. 7: I/O throughput under varying DoF^L and DoF^P.

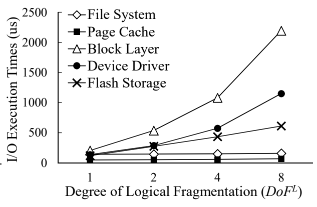
3.2 Need for Logical Defragmentation

To measure the significance of logical and physical fragmentation in terms of performance impact, we measured the throughput of reading a file `foo` under different values of DoF^L(`foo`) and DoF^P(`foo`). In order to control DoF^L values in our study, we made a simple utility which repeatedly splits a given file `foo` until DoF^L(`foo`) reaches the target DoF^L number. The performance measurement was conducted on the mobile flash storage emulator so that the degree of physical fragmentation DoF^P(`foo`) can also be carefully controlled. Based on the majority of the distribution in Fig. 6, the DoF^L value was between 1 and 8, while the DoF^P value was between 0 and 0.25. Fig. 7(a) shows that, when there was no physical fragmentation (DoF^P = 0), a high degree of logical fragmentation (DoF^L = 8) significantly degraded the I/O throughput by 75% compared to the case without any logical fragmentation (DoF^L = 1). On the other hand, increasing DoF^P(`foo`) from 0 to 0.25 only slightly degraded the throughput, no more than 20% for each DoF^L value. This observation suggests that logical fragmentation should be managed in a more aggressive manner than physical fragmentation.

In order to understand how logical fragmentation affects the overhead in the system software I/O stack, we built a fully integrated storage I/O profiler, IOPro, for quantitative evaluations. IOPro can profile the complete Android I/O stack from the application level to the de-



(a) Execution time changes on N6



(b) Execution time changes on S6

Fig. 8: Execution time changes under varying DoF^L.

vice driver level. The key feature of IOPro is that all I/O activities can be seamlessly linked together via their corresponding file information throughout the entire Android I/O stack. Using this tool, we can easily measure times spent in different I/O stack layers. For each measurement run, IOPro measured execution times spent in the Ext4 file system, the page cache, the block layer, the device driver and the mobile flash storage , respectively, on each of our inspected smartphones. For brevity's sake, we only include the measurement data for N6 and S6 in this section, which represent smartphones with eMMC devices and with UFS devices, respectively.

In order to evaluate the effect of logical fragmentation, we measured I/O execution times while varying DoF^L from 1 (no fragmentation) to 8 (heavy fragmentation). For all the measurements, we ran a simple synthetic I/O workload which reads a 512-KB file. The 512-KB file was pre-split into multiple fragments by our fragmentation utility so that the target DoF^L can be satisfied. Figs. 8(a) and 8(b) show how different I/O stack layers were affected under varying DoF^L values on N6 and S6, respectively. The times spent for the block layer, the device driver, and the flash storage device have increased as with the increasing DoF^L values. On the other hand, the times spent in the file system and page cache layers are barely affected. (In the block layer and the device driver, the increased number of block I/O requests in accessing the fragmented file directly affected the overhead of the I/O scheduler, handshaking and interrupt handling [36-41].) In mobile flash storage, although the same I/O layers were affected as in HDDs by the increased number of block I/O requests, the relative impact on these I/O layers were quite different from that in HDDs. As shown in Figs. 8, the block layer is dominantly affected by the number of block I/O requests over the flash storage device. In HDDs, the impact on the HDD device would have been very dominant, making the impact on the rest of I/O layers negligible.

3.3 Need for Physical Defragmentation

As previously shown in Fig. 6, most of the files have small DoF^P values (≤ 0.25). This is because, with the rich I/O parallelism inside of flash storage, it is very unlikely that a file suffers from extremely low I/O parallelism. For example, suppose that data are allocated among eight channels of equal availability, the probability that a 64-KB file composed of eight 8-KB flash

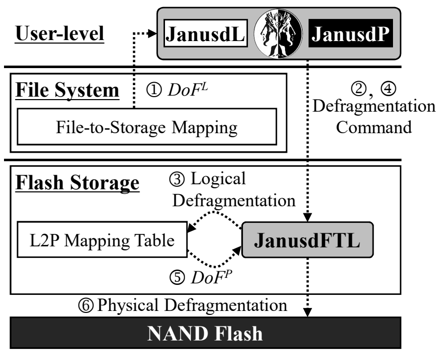


Fig. 9: An overall architecture of janusd.

pages is entirely allocated to one single channel would be 0.00004%. This probability further reduces if the file size is larger than 64 KB. On the other hand, the probability that the 64-KB file is allocated to 6 or more channels would around 80%.

Although it is a rare case that a file has a very high DoF^P value, the overall performance may still be adversely affected if a physically fragmented file is frequently accessed. Fig. 7(b) shows that, a high degree of physical fragmentation (i.e., ≥ 0.5) severely degraded the I/O throughput even when the degree of logical fragmentation was low. For example, even if a file was not fragmented at all in the logical space (DoF^L=1), if the file had a DoF^P value of 0.5, the I/O throughput became only 48% of that with DoF^P=0. Because logical and physical fragmentation is decoupled on flash storage, in such a rare case of high physical fragmentation, it is not sufficient to perform logical defragmentation only, and physical defragmentation is necessary to redistribute data among channels at a cost of flash lifetime.

4 Design and Implementation of Janusd

Our analysis in Sections 2 and 3 strongly indicates that file system fragmentation causes serious performance degradation even in flash storage, badly affecting the quality of user experiences in mobile systems. Moreover, unlike in HDDs, logical and physical fragmentation in flash storage must be handled in different manners.

Janusd is designed to effectively cope with the problems arising from logical and physical fragmentation at a low cost. Fig. 9 shows an organization of janusd with two defraggers, janusdL and janusdP, which are implemented as a user-level tool like e4defrag. Once the janusdL or janusdP is run by end users, it collects information of files to decide whether or not to trigger logical or physical defragmentation. To perform logical/physical fragmentation, special supports from the flash storage side are required. Those supportive functions are implemented as a firmware module, called janusdFTL, which is an extension of the existing FTL algorithm. Janusd is designed with a minimal change to the existing system. Thus, it is unnecessary to change the underlying file system and OS kernel, except for the addition of a device driver for communication between the user-level tool and janusdFTL.

JanusdL is responsible for resolving logical fragmentation of files. JanusdL selects a list of fragmented files based on DoF^L of files (see ① in Fig. 9). Instead of physically moving files' data to another location, it sends a defragmentation command to janusdFTL (②) so that the logical-to-physical mapping table inside of flash storage (③) will be updated. This design enables us to resolve logical fragmentation without any physical data copies (see Section 4.1). JanusdP does not change logical layouts of files. Instead, it is in charge of resolving physical fragmentation for better exploitation of multiple channels in flash storage by re-distributing data among channels. JanusdP notifies janusdFTL of a list of frequently accessed files (④), and janusdFTL calculates the DoF^P values of the files (⑤) based on the physical data allocation inside of flash storage. Because data copies have negative impact on flash memory lifetime (see Section 4.2), among the frequently accessed files, janusdFTL performs physical defragmentation only on the files with high DoF^P values (⑥).

For the janusdL/P and janusdFTL to communicate with each other, new custom interfaces must be added. Table 2 summarizes a set of new custom interfaces, which can be implemented using user-defined command facilities of SATA and NVMe. Detailed descriptions of janusdL/P will be given in the following subsections.

4.1 JanusdL: Logical Defragmentation

Because janusdL inherits most of the features and algorithms from e4dfrag, the implementation of janusdL is done with slight modifications of e4dfrag.

Logical Defragmentation: When janusdL is invoked, it first searches for fragmented files using file-to-storage mapping. JanusdL calls the `FIBMAP` command of the Linux VFS to obtain a list of logical block addresses (LBAs) where the data of a given file is stored, and then it calculates the values of DoF^L of the file accordingly. With a list of files for logical defragmentation, the following process repeats for each of the files: JanusdL first looks for free and continuous LBAs as the destination where the file fragments can be moved to. These destination LBAs are obtained using the existing free-space allocation feature in e4dfrag. With the LBAs of the file fragments (source LBAs) and the destination LBAs, janusdL sends a `defrag` command, shown in Table 2, containing pairs of source-destination LBAs to janusdFTL in flash storage. Upon receiving `defrag` command, janusdFTL updates its logical-to-physical (L2P) mapping table so that the destination LBAs will refer to the physical pages referred to by the source LBAs. After completion of the command, janusdL revises the pointers in the inode of the defragmented file so that host applications can access the file through continuous LBAs.

Table 2: Custom interfaces for janusd.

Command	Description
defrag(list `src_LBA`, list `dst_LBA`)	Change `src_LBA` in logical-to-physical mapping table to `dst_LBA`.
flush()	Flush buffered defrag log to flash from DRAM.
check()	Check whether commit completion flag is saved at defrag log or not.
discard()	Delete the uncommitted log entries in defrag log.

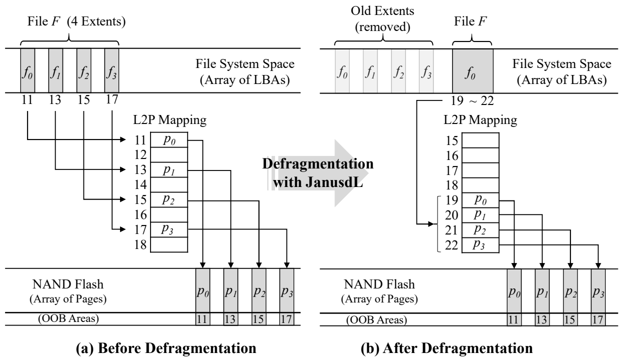


Fig. 10: An example of defragmentation in janusd.

Fig. 10 illustrates an example of how janusdL performs logical defragmentation. We assume that a target file F for defragmentation is fragmented into four extents f_0, f_1, f_2, and f_3, and they are stored in LBAs 11, 13, 15, and 17 (source LBAs), respectively. JanusdL sends a defrag command to map the extents to new LBAs 19 to 22 (destination LBAs). JanusdFTL first locates a list of physical pages that are mapped to the source LBAs. In this example, the file extents f_0, f_1, f_2, and f_3 at the source LBAs 11, 13, 15, and 17 are mapped to physical pages p_0, p_1, p_2, and p_3, respectively. JanusdFTL then updates the mapping entries of the destination LBAs 19 to 22 so that they refer to the physical pages p_0 to p_3, respectively. Finally, the L2P mapping entries of the source LBAs are unset, and janusdFTL sends an acknowledgment to the host to finish the defrag command. After this, `janusdL` revises the inode of the file to access the new extent f_0 through the new LBAs 19 to 22.

Power Failure Recovery: JanusdL may introduce inconsistency between L2P and P2L mapping information in the event of unexpected power failures. When new data is being written to a page, the FTL stores a corresponding LBA in an OOB area of that page for reverse P2L mapping. Even after a power failure occurs and an L2P mapping table (in DRAM) is lost, the FTL is able to recover a complete L2P mapping table by scanning all of the OOB areas in NAND flash. Unfortunately, when an L2P mapping table gets updated by janusd, corresponding LBAs in OOB areas cannot be updated in sync with the changes of L2P mapping because of NAND flash's erase-before-write constraint. In Fig. 10, for example, the LBA referring to the page p_0 was changed from 11 to 19, but the page p_0 still stores the old LBA (i.e., 11) in its OOB area. Suppose that the L2P mapping table is

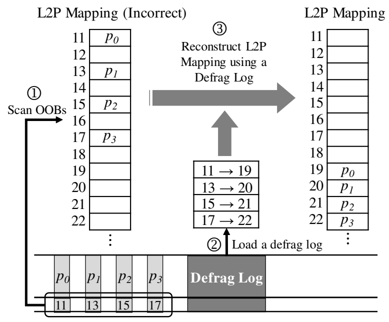
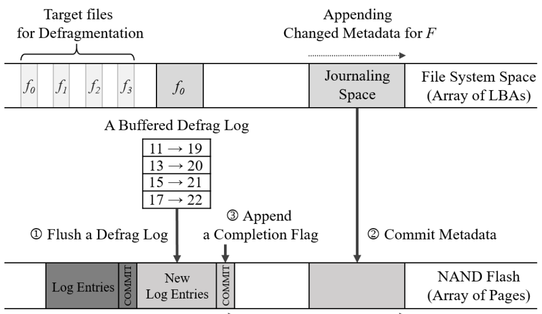


Fig. 11: A power failure recovery of janusdL.

lost due to a power failure. The FTL will rebuild the L2P mapping table by scanning OOB areas. Based on the old P2L information in OOB areas, the page p_0 is referred to by LBA 11. However, at the file-system level, the new extent f_0 is at LBAs 19 to 22 because the inode of the file has been changed. As a result, when applications attempt to access f_0, the file-system sends wrong LBAs (e.g., 19) and the FTL returns invalid data or reports an error.

JanusdL addresses the inconsistency problem by logging all of the history of remapped LBAs in a special log, called a *defrag log*. A defrag log is an ordered collection of entries, each of which is a pair of a source LBA and a destination LBA plus a length. This information can easily be extracted from defrag commands. For example, a defrag log entry for f_0 is (11, 19, 1), where 11 is a source LBA, 19 is a destination LBA, and 1 is a length. Fig. 11 shows an example of how the mapping table is reconstructed after an unexpected power failure. When a flash storage device is rebooted, the FTL scans OOB areas of all pages and builds the L2P mapping table as usual. Then, it checks the defrag log to see if any L2P entries have been remapped for defragmentation and updates the mapping table accordingly.

To prevent frequent writes to flash, janusdFTL keeps defrag log entries in DRAM temporally and flushes them to flash at proper timings. This buffering, however, potentially causes another inconsistency problem – if a power failure occurs before the buffer is flushed to flash, the inconsistency between L2P and P2L mapping occurs. This problem can be solved by using a commit protocol combined with `fsck`. Fig. 12 illustrates how the commit protocol guarantees atomicity of defragmentation. Once all target files are moved and defragmentation is ready to finish, janusdL explicitly (1) flushes the buffered defrag log to flash by transmitting `flush` command in Table 2, (2) writes all file-system's metadata to a journaling area, and (3) appends a *commit completion flag* to the end of the defrag log. On system rebooting, `fsck` modified for janusdL first checks if the latest commit completion flag was written successfully by sending `check` command in Table 2. If not, the system was improperly shut down due to a system failure. Using `discard` command in Table 2, the modified `fsck` asks janusdFTL to discard uncommitted log entries in the defrag log and to rebuild



Target files for Defragmentation
Appending Changed Metadata for F
Journaling Space
File System Space (Array of LBAs)
A Buffered Defrag Log
11 → 19
13 → 20
15 → 21
17 → 22
① Flush a Defrag Log
③ Append a Completion Flag
② Commit Metadata
Log Entries
COMMIT
New Log Entries
COMMIT
NAND Flash (Array of Pages)

Fig. 12: A synchronization of file-system's metadata and defrag-log commits.

an L2P mapping table only with committed ones. In the file system level, at the same time, the modified `fsck` rollbacks all the changes made to files by janusdL and reverts the files to their last consistent states.

Defrag Log Management: The FTL conducts internal page movements for garbage collection and wear leveling. If these page movements involve a page whose LBA is previously remapped, the defrag log must be updated. When a page is moved by garbage collection or wear leveling, janusdFTL writes the page according to its most recent P2L mapping information. The update of L2P mapping is required when a page is overwritten with new data as well. For both cases, since the P2L page mapping has been rewritten to flash, the corresponding old log entry should be removed.

Fig. 13 illustrates how janusdFTL manages the defrag log during garbage collection. Suppose that the flash block where valid pages p_0, p_1, p_2 and p_3 are stored is selected as a victim so that those pages are moved to four free pages p_4, p_5, p_6 and p_7, respectively. Accordingly, the L2P mapping table is updated to refer to new page locations. While moving valid pages, janusdFTL updates P2L mapping in OOBs if they are previously remapped by the defrag remapper. For example, 11 in p_0 is changed to 19 in p_4. After this, the entries of the moved pages are deleted from the defrag log. For example, entries (11, 19, 1), ..., (17, 22, 1) are now unnecessary. However, because of the overwrite restriction, janusdFTL has to append log entries to the defrag log, (11, Ø, 1), ..., (17, Ø, 1), to mark the old entries of LBAs 11 to 17 deleted. By this design, the defrag log may have multiple entries for the same LPAs, for example, (11, 19, 1) and (11, Ø, 1). To ignore old entries when the defrag log is scanned, janusdFTL writes a unique version number together.

As astute readers may notice, the defrag log would grow very large over time. To prevent this, janusdFTL sets a limit on the defrag log size. Once the size limit is reached, janusdFTL performs compaction – it selects flash blocks containing part of the defrag log, filters out obsolete entries, and writes only valid entries to the defrag log. (11, 19, 1) and (11, Ø, 1) are examples of obsolete entries – since L2P is equivalent to P2L, there is no need to keep them in the defrag log. The maximum

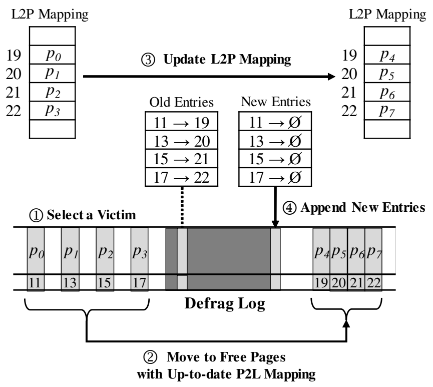


Fig. 13: Updating defrag log during garbage collection.

size of the defrag log is currently set to 10 MB, which is large enough to hold several millions of entries. Thanks to its huge size, almost all of the log entries become obsolete before being selected for compaction, and thus compaction involves few entry copy operations.

4.2 JanusdP: Physical Defragmentation

Different from janusdL, janusdP involves data copies for physical defragmentation. To minimize the negative impact of data copies on flash lifetime, janusdP performs physical defragmentation only on selected files that meet the following criteria: 1) they must be frequently accessed and 2) they must have high dragees of physical fragmentation (i.e., high DoF^P values).

To measure read frequencies of files, we implement a daemon program that keeps track of the total count of read accesses of files using the `inotify` feature provided by the Linux kernel. The read counts of files are stored in a separate file, and the janusdP utility reads the file to determine a list of 50 most frequently read files. JanusdP and janusdL use the same command to communicate with janusdFTL. To notify janusdFTL of physical defragmentation on a file, janusdP stores all the LBAs associated with the file as the source LBAs of a defrag command, but fills all the destination LBAs of the command with a null value -1. In this way, janusdFTL can easily distinguish a command for logical defragmentation from a command for physical defragmentation.

After janusdFTL receives a command for physical defragmentation, it first calculates the DoF^P value for the source LBAs stored in the defrag command. Recall that the DoF^P value associated with a set of LBAs is 0 if the LBAs can be accessed through the maximum I/O parallelism inside of flash storage. We employ 0.5 as an empirical threshold of DoF^P for janusdFTL to conduct physical defragmentation on the source LBAs. If the DoF^P of the LBAs is higher than or equal to 0.5, janusdFTL re-distributes the data (mapped to the source LBAs) among channels for the best I/O parallelism of future accesses. If the DoF^P of the LBAs is lower than 0.5, janusdFTL does nothing because the benefit of physical defragmentation would be marginal.

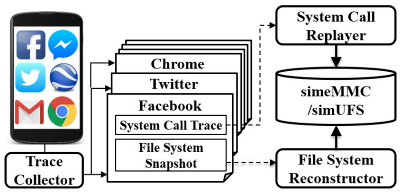


Fig. 14: An overview of our evaluation platform.

5 Experimental Results

In order to objectively understand the performance implication of janusd, we implement a comprehensive evaluation platform in the Linux operating system that supports three useful features, including (1) file-system snapshot/replication, (2) trace collection/replay, along with (3) mobile storage emulation. This evaluation platform makes it possible for us to conduct a set of the evaluations in an easy and convenient manner without modifying various smartphone platforms.

Fig. 14 illustrates our evaluation platform. The snapshot/replication tool allows us to take a storage snapshot of a smartphone and to replicate the same one in local flash storage for experiments. The trace collection/replay tool helps us to collect system-call events (e.g., `read()` and `write()`) from various applications running on real-world smartphones, and it replays them on the local storage. Those features enable us to repeat exactly the same I/O workloads on the same storage setup while varying defragmentation policies.

It is impossible to modify mobile storage devices like eMMC and UFS. Thus, we build two emulated mobile flash devices, called `simeMMC` and `simUFS`, using a customizable SSD device based on Samsung's 843T SSD [27]. 843T SSD supports extended SATA interfaces that allow a host system to directly control channels using NAND-specific I/O primitives (e.g., a page read/write and block erasure). Based on those interfaces, we implement a complete page-level FTL in a block layer of the Linux kernel (ver. 3.10). eMMC and UFS have similar channel architectures as conventional SSDs, except that they have smaller numbers of channels due to limited power budgets. We emulate I/O throughputs of eMMC and UFS by limiting the number of available channels of the 843T SSD to 4 and 8 for `simeMMC` and `simUFS`, respectively. To simulate a smaller I/O queue depth of mobile storage, we also intentionally increase end-to-end I/O latencies between the host and the flash device. As a result, both `simeMMC` and `simUFS` can accurately simulate I/O performance of eMMC and UFS devices over various request sizes.

As mentioned in Section 4, we implement janusdL/P as a user-level tool using e4defrag. The number of code lines newly added to e4defrag is about 400. janusdFTL is implemented as an extended module of the page-level FTL in the block layer. The custom interfaces between janusdL/janusdP and janusdFTL listed in Table 2 are implemented using the ioctl facility of the Linux.

5.1 Usage Scenario of Smartphone

We collect I/O activities of six popular applications running on N6. Table 3 summarizes the usage scenarios of each application. Each scenario starts with launching an application and runs specific tasks described in Table 3 for 10 minutes. The file system utilization is about 83%.

In order to perform evaluations under realistic environments, we create a six-month usage scenario of a smartphone. Based on a statistical study reporting that average daily time spent with a smartphone is 220 minutes [30], we simulate a daily usage scenario of a smartphone by repeating the six scenarios for 220 minutes. In a similar way, we finally create a six-month usage scenario by repeating the daily usage scenario 180 times. The applications are updated every 10 days based on the analysis of the update cycle of Android applications [28].

5.2 I/O Performance Analysis

While executing the six-month usage scenario, we compare the effect of six different defragmentation policies on performance: baseline, janusd, janusdL, e4defrag_1w, e4defrag_2w and e4defrag_4w. (Note that e4defrag_*n*w indicates when we invoke e4defrag with every *n* weeks.) For a fair comparison, before the execution of the scenario with a specific policy, the file system is initialized with the snapshot/replication tool mentioned in Section 5.1. Baseline does not perform file defragmentation. For janusd and janusdL, we execute janusd and janusdL every week. In the case of e4defrag, we invoke e4defrag with three different cycles, 1 week, 2 weeks and 4 weeks.

Fig. 15 shows that janusd achieves a consistent I/O throughput similar to or slightly better than e4defrag_1w ((a) Chrome 58 MB/s and (b) Game 66 MB/s). An interesting observation here is that the I/O throughput drops sharply even after one week without defragmentation. This indicates that frequent invocations of defragmentation are desirable to maintain high and consistent performance. In particular, janusd works better than janusdL and e4defrag_1w, offering the performance very close to the clean file system. Compared with janusdL and e4defrag_1w that perform only logical defragmentation, janusd conducts physical defragmentation that physically distributes fragmented pieces of files across different channels, improving I/O parallelism of file access. Fig. 16 shows I/O throughputs of the rest of the applications not shown in Fig. 15. On average, janusd improves the I/O throughput by 57% and 76% over baseline for simeMMC and simUFS, respectively. As expected, as the larger the values of DoF^L, the higher the I/O throughputs improved by janusd.

Table 3: A summary of benchmark scenarios.

Scenario	DoF^L	Scenario Description
Chrome	1.34	Launching app → Viewing webpages
Messenger	1.99	Launching app → Viewing chat records
Gmail	2.18	Launching app → Viewing emails
Facebook	2.55	Launching app → Viewing online news
Twitter	2.75	Launching app → Viewing online news
Game	3.02	Launching Lineage 2 → Playing game

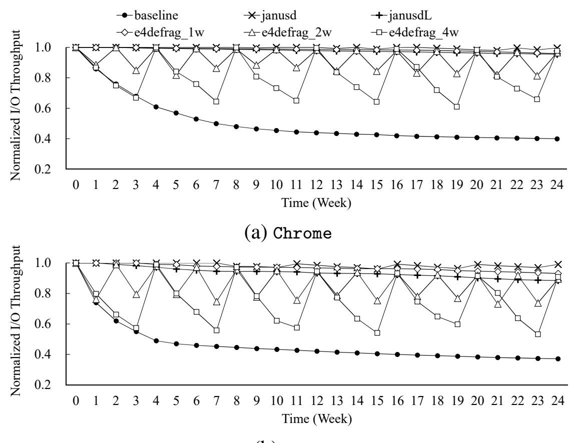


Fig. 15: Changes of I/O throughput over 6 months.

In order to analyze the impact of janusd on the quality of user experiences, we measure app launching times of the usage scenarios. We replay system call traces that are issued while an app is being launched, and then measure the reductions of I/O elapsed times spent by flash storage. Fig. 17 shows that janusd reduces the app launching times by up to 29% and 36% for simeMMC and simUFS over baseline, respectively. Our results confirm that janusd is effective in improving the quality of user experiences in smartphones.

Finally, Figs. 16 and 17 show that the performance improvement by janusd is more significant in a faster storage device like simUFS than a slower one, simeMMC. As observed in Section 3.2, the heavy fragmentation of files increases the number of small I/O requests to flash storage, which results in the increase of I/O stack overheads. SimUFS is more badly affected by the increased software I/O overheads – because of a fast storage access time, the handling of I/O requests at the software I/O stack level accounts for a larger proportion of the total I/O elapsed time. Janusd translates a large number of small I/Os to a fewer large ones, alleviating a performance penalty caused by I/O stack overheads. As a result, simUFS gets more benefits over simeMMC from the reduction of I/O stack overheads.

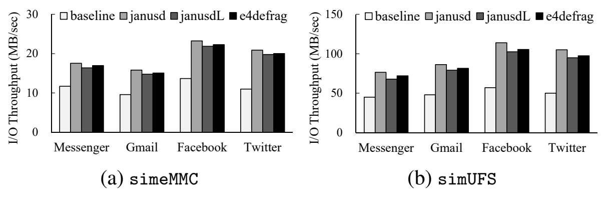
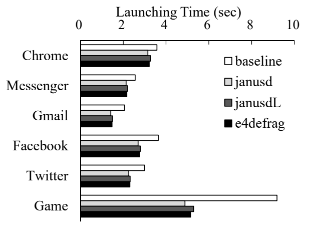
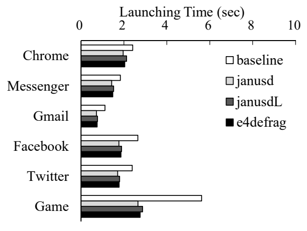


Fig. 16: The impact of janusd on the I/O throughput.



(a) `simeMMC`



(b) `simUFS`

Fig. 17: The impact of janusd on the app launching time.

5.3 Lifetime Analysis

JanusdP has to physically move data. By performing physical defragmentation only on files that are physically fragmented and heavily read, janusdP minimizes its negative effect on flash lifetime. Table 4 shows that physical defragmentation by janusdP involves only a small amount of data copies, 364 MB, which is negligible compared to e4defrag_4w that copies data of 217 GB. Even though a smaller number of files are defragmented, its impact on performance is more significant than e4defrag as illustrated in Fig. 16. This is because janusdP optimally relocates files in multiple channels by taking into account the physical layout of flash storage.

Finally, we measure the amount of extra data movements needed for the maintenance of a defrag log in NAND flash. As mentioned in Section 4.2, we limit the size of a defrag log to 10 MB, and if its size exceeds the limit, janusd triggers compaction to reduce the log size. Since janusdL does not make data copies, the amount of data copies of janusdL in Table 4 indicates the amount data coped during the defrag log compaction. 219-MB data copies by janusdL is negligible over e4defrag_1w that involves 156-GB data copies for defragmentation.

6 Related Work

File Defragmentation: Recent interests in file defragmentation on flash storage were largely motivated by high-performance I/O support in flash storage. As flash storage gets faster, SW I/O stack overheads are emerging as a new I/O performance bottleneck, and flash fragmentation is reevaluated as a potential I/O bottleneck for flash storage. For example, Ji *et al.* showed that file fragmentation negatively affected the performance of mobile applications through an empirical study using several used smartphones [22]. In particular, they confirmed that redundant I/Os caused by fragmented files account for a nontrivial fraction of the total I/O time, degrading the overall I/O performance. More recently, Park *et al.* presented that file defragmentation on a log-structured file system reduced the frequency of I/O requests to a flash storage system, thereby improving the overall read performance [35]. While existing studies just discovered fragmentation problems [22-24] or presented a file-system-specific solution [35], our work, which is based on a detailed characterization study of flash file fragmentation, proposes a general scheme that can solve the fragmentation problem in flash storage, regardless of application types or system platforms.

Table 4: Impact of janusd on the amount of data copies.

e4defrag_1w	e4defrag_2w	e4defrag_4w	janusdL	janusdP
156 GB	182 GB	217 GB	0.219 GB	0.364 GB

Remapping Optimization in Flash: There are several studies proposed to improve flash storage performance by enhancing the remapping function of the FTL [31-34]. For example, Choi *et al.* presented a remapping technique that avoided double writing in journaling file systems [31]. Kang *et al.* proposed a transactional FTL for SQLite databases, which remapped a logical address from a physical location to a new physical location [32].

Our work is similar to the aforementioned studies in that it leverages an FTL's remapping function to offer better I/O performance. The above studies, however, did not take into account of the fragmentation problem in flash storage, and thus their remapping schemes could not effectively deal with fragmented files. Consequently, those studies are not applicable to resolve fragmentation.

7 Conclusions

We have presented a complete treatment for file fragmentation on mobile flash storage. From a systematic evaluation study, we showed that 1) file fragmentation is a recurring problem with a short recurrence interval and 2) the impact of file defragmentation on I/O performance is significant. By exploiting the decoupled fragmentation characteristics of flash storage, we proposed a novel flash-aware decoupled defragger, janusd, with two separate defraggers, janusdL and janusdP. JanusdL supports logical defragmentation without data copies by remapping the LBAs of the logically fragmented files with the FTL's mapping table. By saving a complete history of remapped LBA pairs in the defrag log, janusdL can safely recover from sudden power failures. On the other hand, janusdP, which is rarely invoked, improves the degree of the I/O parallelism of files which are severely limited in their available I/O parallelism. Our evaluation results showed that janusd can improve the I/O throughput by 57% and 76% on average in the Ext4 file systems on `simeMMC` and `simUFS`, respectively.

Our work can be extended in several directions. For example, janusdL can be easily extended to support different types of spatial locality of a file system such as free-space defragmentation. It would be also possible to support *defrag-on-write* that triggers logical defragmentation right after calling write() because the overhead of janusdL is negligible (i.e., $<$ 1 ms) over the cost of write() itself. Defrag-on-writes would realize a fragmentation-free file system, guaranteeing no performance degradation from fragmented files.

8 Acknowledgments

We would like to thank Ji-Yong Shin, our shepherd, and anonymous referees for valuable comments that greatly improved our paper. This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2015M3C4A7065645), Ministry of Science and Technology of Taiwan (MOST 104-2221-E-009-011-MY3) and China National 863 Program 2015AA015304. The ICT at Seoul National University provided research facilities for this study. (*Corresponding Author: Jihong Kim*)

References

[1] MANTHUR, A., CAO, M., AND BHATTACHARYA, S. The New ext4 File System: Current Status and Future Plans. In *Proceedings of Linux Symposium* (2007).

[2] E4defrag - Online Defragmenter for Ext4 File System. `http://manpages.ubuntu.com/manpages/trusty/man8/e4defrag.8.html`

[3] Condusiv Diskeeper. `http://www.condusiv.com/products/diskeeper/`

[4] Auslogics Disk Defrag. `http://auslogics.com/en/software/disk-defrag/`

[5] Defraggler. `http://www.piriform.com/defraggler`

[6] Smart Defrag. `http://www.iobit.com/en/iobitsmartdefrag.php?a`

[7] Samsung SSD Performance Enhancement & Maintenance. `http://www.samsung.com/semiconductor/minisite/ssd/support/faqs-03.html`

[8] Frequently Asked Questions for Intel Solid State Drives. `http://www.intel.com/content/www/us/en/support/software/000006110.html`

[9] Crucial SSD and HDD Support & Maintenance. `http://www.crucial.com/usa/en/support-system-maintenance-defragment-hard-drive`

[10] KEHRER, O. O&O Defrag and Solid State Drives. `http://www.oo-software.com/en/docs/whitepaper/ood_ssd.pdf`

[11] LIND, A. Auslogics: How to Defrag Disk Drives The Right Way. `http://www.auslogics.com/en/articles/how-to-defrag/`

[12] Windows 8 TRIM SSD Instead of Defragmentation. `http://www.eightforums.com/tutorials/8615-optimize-drives-defrag-hdd-trim-ssd-windows-8-a.html`

[13] Windows 10 TRIM SSD Instead of Defragmentation. `http://www.tenforums.com/tutorials/8933-optimize-defrag-drives-windows-10-a.html`

[14] Embedded MultiMediaCard (e.MMC). `http://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc`

[15] Universal Flash Storage (UFS). `http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs`

[16] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference* (2008).

[17] KANG, J.-U., KIM, J.-S., PARK, C., PARK, H., AND LEE, J. A Multi-channel Architecture for High-performance NAND Flash-based Storage System. *Journal of Systems Architecture: the EUROMICRO Journal* (2007).

[18] PARK, S.-H., HA, S.-H., BANG, K., AND CHUNG, E.-Y. Design and Analysis of Flash Translation Layers for Multi-channel NAND Flash-based Storage devices. *IEEE Transactions on Consumer Electronics* (2009).

[19] HU, Y., JIANG, H., FANG, D., TIAN, L., AND LUO, H. Performance Impact and Interplay of SSD Parallelism Through Advanced Commands, Allocation Strategy and Data Granularity. In *Proceedings of the ACM International Conference on Supercomputing* (2011), pp. 96–107.

[20] JUNG, M., AND KANDEMIR, M. T. An Evaluation of Different Page Allocation Strategies on High-Speed SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (2012).

[21] JUNG, M., WILSON III, E. H., AND KANDEMIR, M. T. Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks. In *Proceedings of the International Symposium on Computer Architecture* (2012), pp. 404–415.

[22] Ji, C., Chang, L., Shi, L., Wu, C., Li, Q., and Xue, C. J. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (2016).

[23] Conway, A., Bakshi, A., Jiao, Y., Zhan, Y., Bender, M. A., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Farach-Colton, M. File Systems Fated for Senescence? Nonsense, Says Science!. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2017).

[24] Kinsella, J. The Impact of Disk Fragmentation. `http://condusiv.com/disk-defrag/fragmentation-impact/`

[25] Kessler, M. Maintaining Windows 2000 Peak Performance Through Defragmentation. `https://msdn.microsoft.com/en-us/library/bb742585.aspx`

[26] Sinofsky, S. Disk Defragmentation Background and Engineering the Windows 7 Improvements. `https://blogs.msdn.microsoft.com/e7/2009/01/25/disk-defragmentation-background-and-engineering-the-windows-7-improvements/`

[27] SAMSUNG 843T Data Center Series. `http://memorysolution.de/mso_upload/out/all/SM843T_Specification_v1.0.pdf`

[28] Kumar, U. Understanding Android's Application Update Cycles. `https://www.nowsecure.com/blog/2015/06/08/understanding-android-s-application-update-cycles/`

[29] Twitter Version History. `https://www.apk4fun.com/history/2699/`

[30] Hechtel, E. How Smartphones and Mobile Internet Have Changed Our Lives. `https://testobject.com/blog/2016/01/smartphones-mobile-internet-changed-our-life.html`

[31] Choi, H.-J., Lim, S.-H., and Park, K.-H. JFTL: A Flash Translation Layer Based on A Journal Remapping for Flash Memory. *ACM Transactions on Storage* (2009).

[32] Kang, W.-H., Lee, S.-W., Moon, B., Oh, G.-H., and Min, C. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013), pp. 97–108.

[33] Zhang, Y., Arulraj, L. P., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2012).

[34] Oh, G., Seo, C., Mayuram, R., Kee, Y., and Lee, S. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *Proceedings of the International Conference on Management of Data* (2016), pp. 343–354.

[35] Park, J., Kang, D.-H., and Eom, Y.-I. File Defragmentation Scheme for A Log-structured File System. In *Proceedings of the ACM SIGOPS Asia-Pacific Workshop on Systems* (2016), pp. 19.

[36] Traeger, A. An Introduction to Linux Block I/O. `http://researcher.ibm.com/researcher/files/il-AVISHAY/01-block_io-v1.3.pdf`

[37] I/O Schedulers. `http://www.makelinux.net/books/lkd2/ch13lev1sec5`

[38] T10, Technical Committee of the InterNational Committee on Information Technology Standards. SCSI TEST UNIT READY Command. `http://www.t10.org/ftp/t10/document.06/06-022r0.pdf`

[39] T10, Technical Committee of the InterNational Committee on Information Technology Standards. SCSI Block Commands - 3 (SBC-3). `http://www.t10.org/ftp/t10/document.05/05-344r0.pdf`

[40] Anderson, D. C., Chase, J. S., Gadde, S., Gallatin, A. J., and Yocum, K. G. Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet. In *Proceedings of the USENIX Annual Technical Conference* (1998).

[41] Ahmad, I., Gulati, A., and Mashtizadeh, A. vIC: Interrupt Coalescing for Virtual Machine Storage Device I/O. In *Proceedings of the USENIX Annual Technical Conference* (2011).

[42] Hahn, S.S. Impact of File Fragmentation on Android Smartphones. `http://cares.snu.ac.kr/?view=publications&menuN=34#2017_Technical_Report`