ARS: Reducing F2FS Fragmentation for Smartphones using Decision Trees

Lihua Yang, Fang Wang, Zhipeng Tan*, Dan Feng, Jiaxing Qian, Shiyun Tu
Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System
Engineering Research Center of Data Storage Systems and Technology, Ministry of Education of China
School of Computer Science & Technology, Huazhong University of Science & Technology
Email: {lihuayang, wangfang, tanzhipeng, dfeng, jiaxingqian, tushiyun}@hust.edu.cn, *corresponding author

***Abstract*—As we all know, file and free space fragmentation negatively affect file system performance. F2FS is a file system designed for flash memory. However, it suffers from severe fragmentation due to its out-of-place updates and the highly synchronous, multi-threaded writing behaviors of mobile applications. We observe that the running time of fragmented files is 2.36× longer than that of continuous files and that F2FS's in-place update scheme is incapable of reducing fragmentation. A fragmented file system leads to a poor user experience.**

Reserving space to prevent fragmentation is an intuitive approach. However, reserving space for all files wastes space since there are a large number of files. To deal with this dilemma, we propose an adaptive reserved space (ARS) scheme to choose some specific files to update in the reserved space. How to effectively select reserved files is critical to performance. We collect file characteristics associated with fragmentation to construct data sets and use decision trees to accurately pick reserved files. Besides, adjustable reserved space and dynamic reservation strategy are adopted. We implement ARS on a HiKey960 development platform and a commercial smartphone with slight space and file creation time overheads. Experimental results show that ARS reduces file and free space fragmentation dramatically, improves file I/O performance and reduces garbage collection overhead compared to traditional F2FS and F2FS with in-place updates. Furthermore, ARS delivers up to 1.26× transactions per second under SQLite than traditional F2FS and reduces running time of realistic workloads by up to 41.72% than F2FS with in-place updates.

I. Introduction

The number of smartphone users worldwide surpasses 3 billion in 2019 [1] and Android smartphones account for 80% of the sales. The performance of Android smartphones has attracted attentions of many users and researchers. Fragmentation accumulates easily over time in most file systems [2]. There are two types of fragmentations, file and free space fragmentation. File fragments are discrete data extents of a file and free space fragments are scattered valid blocks. They interweave with each other. If a file is written as many small separate parts, its fragments would cut off continuous free space, which increases free space fragmentation. The heavier free space fragmentation is, a new-coming file would be more fragmented. Frequently updated applications append new records to different database files in parallel, which causes severe fragmentation. The studies [2]–[5] have shown that fragmentation ages file systems and compromises flash device performance.

Flash memory has been widely used as mobile storage and has an out-of-place write mechanism due to the erase-before-write constraint. F2FS [6], designed to extract better performance on flash memory as follow a log-structured approach, is an ideal file system for smartphones. However, similar to most file systems [2], F2FS slows down by fragmentation. F2FS updates at new addresses constantly and removes the old versions later. There are a mass of scattered *holes* (small invalid or free blocks) that cannot be directly used for an upcoming file, which requires garbage collection (GC). We demonstrate that F2FS generates a lot of fragments and increases running time under a typical workload in Android: 4 KB write followed by `fsync()`, and observe that F2FS with in-place update scheme (represented by `IPU` in the rest of this paper) is incapable of reducing fragmentation and it is necessary to reduce fragmentation for F2FS in real-world smartphones.

We propose ARS, an **a**daptive **r**eserved **s**pace scheme, to improve the performance of F2FS in smartphones by reducing file and free space fragmentation. The adaptation of ARS lies in the updated reserved files, adjustable reserved space and variational reservation strategy. Different from traditional defragmentation solutions, e.g., copying fragmented data together which shortens flash memory life, ARS prevents fragmentation proactively which does not copy data. Observing the performance differences between continuous and fragmented files in motivation experiment, we take space reservation into account. However, reserving space for all files wastes space and loses advantages of logging scheme. We selectively extract some files rather than all files to reserve space based on the observation that some files are far more frequently read/written than others. There are plenty of frequently accessed files and it is difficult to manually select reserved files. Since different users use smartphones differently and a user has a personal habit of using smartphone, there are differences and similarities between traces. We filter file features and add categorization labels based on their fragmentation degree and I/O behaviors to construct data sets. We choose decision tree to efficiently select reserved files. Besides, we set the reserved space to be multiple of a segment, and update these files in-place when file system space utilization is low and reclaim unused reserved space when high.

We implement ARS as a module in F2FS on a HiKey960 development platform [7](denoted as Hikey960 in the rest of this paper) and a Huawei Mate10 Pro [8] smartphone and verify the effectiveness of ARS by abundant experiments.

Experimental results show that file and free space fragmentation decrease greatly with ARS, compared to both traditional F2FS and `IPU`. ARS improves the fragmented file I/O performance by up to 179.62% and reduces GC overhead by up to 78%. Under SQLite, ARS outperforms traditional F2FS and `IPU` by up to 1.26× and 1.29×, respectively. Moreover, ARS reduces running time of realistic workloads, Facebook and Twitter, by 35.38% and 24.14% respectively than `IPU` at moderate space utilization of file system. Besides, ARS is deployed with acceptable space and file creation time overheads.

The remainder of this paper is organized as follows. Section II introduces the related work. Section III describes motivations. Section IV presents the design and implementation of ARS. Section V shows evaluation results. Section VI concludes.

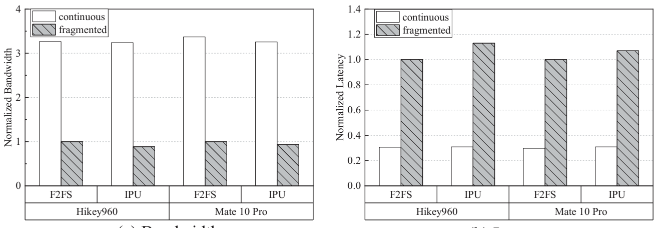
II. Related Work of Fragmentation

The fragmentation research extends from HDDs to SSDs. Moreover, the slow response caused by fragmentation of smartphones is attractive due to their universality and user interaction. The methods to reduce fragmentation can be divided into two categories. One is defragmenting into a whole piece after accumulating fragmentation. The other is preventing its occurrence before accumulating fragmentation.

The work of [9]–[11] studies the fragmentation of disks. Smith and Seltzer [9] artificially age a file system by replaying real workloads. DFS [10] dynamically relocates and clusters data blocks to place small fragmented files physical contiguously. Using `Impression` [11], a framework to generate accurate file system images, can achieve representative level of file fragmentation. However, the properties and usage characteristics of SSDs are different from disks, so defragmentation for disks is not suitable for NAND Flash based devices.

Alex et al. [4] prove that file systems tend to become fragmented, or age, whether on HDDs or SSDs. They observe aging impacts on performance quickly and dramatically but aged `BetrFS` [12] does not slow down. `BetrFS` is implemented on desktops and complicated to be deployed on smartphones since it ports TokuDB into the kernel. 65% recent file systems ignore aging [2] and `Geriatrix` induces fragmentation in both allocated files and free space. `Geriatrix` is deployed on servers. Park et al. [13] propose a defragmentation scheme that reorders the valid data blocks belonging to a victim segment on computers. However, I/O characteristics of smartphones are different from servers or desktops [14], so the schemes maybe ineffective for smartphones.

There is an urgent need to reduce file and free space fragmentation of F2FS in smartphones. Ji et al. [3] examine how severe file fragmentation is in real Android devices, especially for SQLite database files and identify that fragmentation really affects performance. It would be better to take free space fragmentation into consideration. Hahn et al. [5] respectively investigate logical fragmentation of ext4 on smartphones and physical fragmentation on mobile flash storage emulators. They propose a decoupled defragmenter, `janusd`. Its firmware module, `janusdFTL`, leverages a remapping function of flash translation layer (FTL), however, hardware vendors encapsulate FTL in smartphones so far. Liang et al. [15] show several counterintuitive observations of F2FS but do not implement a defragmentation solution. Ji et al. [16] summarize the measurement, evaluation and treatment of file fragmentation in mobile devices. Space preallocation is valid for ext4, F2FS and other file systems, however, their layouts and management mechanisms are different, so the performance gains and overheads by reducing fragmentation are varying.



(a) Bandwidth (b) Latency

Fig. 1: Bandwidth/latency of different files.

III. Motivations

We investigate whether fragmentation accumulation is easy and how serious the impact of fragmentation on F2FS is in real-world mobile devices. In particular, F2FS provides in-place update scheme and we also explore whether `IPU` is effective in alleviating fragmentation. The parameter *ipu_policy* in F2FS controls five policies of in-place updates: F2FS_IPU_FORCE, F2FS_IPU_SSR, F2FS_IPU_UTIL, F2FS_IPU_SSRUTIL and default F2FS_IPU_FSYNC. The most fragmented files are SQLite database files [3]. While a user uses an application, the application performs tasks in multi-threaded mode in the background, e.g., the Facebook creates 18 concurrent threads that write different SQLite files in parallel [16]. The sequential read performance drops significantly as fragmentation increases [15]. So we compare the file extents and sequential read performance of continuous with fragmented files in pristine F2FS on Hikey960 and Huawei Mate10 Pro. We deploy the experiment under a typical workload in Android: 4 KB write followed by `fsync()` where individual random writes are synchronized to storage. We use fio [17] to construct the continuous file written sequentially while the fragmented file written concurrently. Regardless of the traditional F2FS or `IPU`, the number of extent for a continuous file is 1 while that for a fragmented file is about 32,767. As shown in Fig. 1, the sequential read bandwidth of fragmented file in traditional F2FS declines significantly, by 69.39% and 70.35% on Hikey960 and Mate10 Pro, respectively, and the running time is 2.27× and 2.36× longer accordingly. The performance of `IPU` also declines by 72.75% and 71.19% on Hikey960 and Mate10 Pro, respectively. The experimental results on Hikey960 and Mate10 Pro demonstrate the same conclusion. Only writing 8 files randomly at the same time accumulates heavy fragmentation and sequential read performance drops dramatically, so fragmentation accumulation is easy and detrimental.

We want to understand the impact of fragmentation when a file is split into fragments of different sizes. We set different fio block sizes to obtain file fragments of different sizes

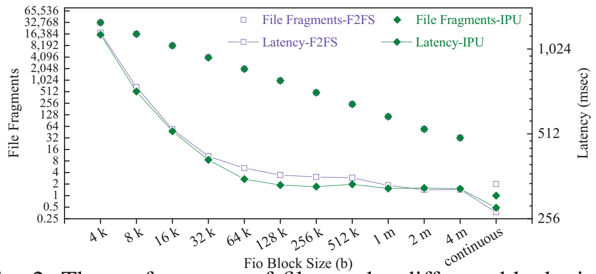


Fig. 2: The performance of files under different block sizes.

while randomly writing a file of 128 MB. The sequential read performance is shown in Fig. 2. The number of file fragments and the sequential read latency decrease as the fragment size increases. The negative impact of fragmentation is very serious when a file fragment is small than 64 KB and is slight when a file fragment is larger than 1 MB. We observe that `IPU` does little to mitigate fragmentation in the case of multi-threaded synchronization from Fig. 1, and `IPU` can slightly reduce fragmentation for a single file from Fig. 2.

Although the performance of smartphone system is influenced by complicated factors, fragmentation is the key factor contributing to the I/O throughput degradation and longer response time [3]. Since FTL and flash memory are transparent to users, we optimize the logical fragmentation without considering the physical fragmentation of device in this paper. More importantly, a performance degradation mostly comes from logical fragmentation [5].

IV. Design and Implementation of ARS

Inspired by the motivation experiment that continuous files perform better than fragmented files, we make the data of a file as continuous as possible. What's more, we observe that some files are far more frequently written than others. We classify files into written-many and written-once and analyze the trace properties of Facebook and Twitter. The details of Facebook and Twitter traces provided by Mobibench are described in [14]. Facebook trace completes 14,429 writes and Twitter trace completes 6,029 writes. Their respective top 19.8%, 25% of files are written-many as each of them is written more than 100 times. Gathering together the data of a file proactively reduces file fragmentation and the chance of its fragments splitting other files. Fig. 3 illustrates the differences among traditional F2FS, reserving space for all files and our proposed scheme. The incoming data stream describes the sequence of operations: write `A1`, `A2`, `B1` and `B2`; write `C`, update `A2` with `A2'`, write `B3`; write `A3`, update `B2` with `B2'`; write `D1`, `D2`. Traditional F2FS updates a new data block out-of-place, the second scheme reserves space for all files, and ARS updates some files in-place and append writes others. Files `A` and `B` are fragmented in traditional F2FS while they are continuous in the second and ARS scheme after updating `A2`, `B2` (see ③ in Fig.3). The second scheme also reserves space for file `C`. Considering file `D` coming, traditional F2FS requires GC and the second scheme needs to reclaim unused space of file `C` and redistribute it to `D`, so both schemes take longer, however, file `D` is just right written in ARS quickly(④). Locating distributed data of files `A` and `B` costs more in next accesses for traditional F2FS. The second scheme is more likely to waste space and cause free space fragmentation. Our proposal tries to make better tradeoff between fragmentation and free space. There are three important issues when designing ARS: (1) the solution to effectively choose reserved files, (2) the amount of reserve space that marked as three continuous squares in Fig. 3 and (3) the reservation and recycling strategies.

A. Reserved Files

We collect a trace of using smartphone half an hour, including using WeChat, Weibo and some system applications. There are 822,532 writes and 148,423 reads. We speculate these written-many files like files `A` and `B` in Fig. 3 are good reserved objects while written-once files like file `C` are not since written-once files are free of fragmentation. A reserved object is a file that suitable for reserving space. We observe that most of the `.db`, `.db-wal` and `.db-journal` files are written-many, which is in line with that the SQLite files are among the most severely fragmented files [3]. However, we cannot judge a file whether to reserve based on its file type, because we find some `.png` files are written-once but others are written-many counterintuitively. Besides, a user uses a variety of applications over time, which generates a lot of data. It is unfeasible to manually identify the reserved files. There are similarities and differences between these massive traces. Machine learning algorithms accomplish efficient analysis for diverse data and adapt to dynamic workloads. We want a low-cost, high-accuracy machine learning algorithm to select reserved files efficiently.

It is challenging to filter file features to construct a data set for machine learning. There are many features in the file system that can be used to classify files. **File type** is related to fragmentation given the fact that most database files are fragmented, but is inaccurate as mentioned. **File size** continue to grow over time. A larger file is more likely to be fragmented while there are severe free space fragmentation but it may be written-once. **File write time interval** is the time interval between the first and last write of a file in a trace. The greater the file write time interval is, the more likely the file is to be shredded by others. **Write request times** is a usable feature because a file becomes fragmented as write requests increase. Mobibench provides system calls captured by `strace` and the write operation object in the replay file is described by `fd`, so we can't obtain the file type directly. Finally, we choose **file size**, **write time interval** and **write request times** that are easy to get, precise and associated with generating fragmentation as features. Then we preprocess these data features to obtain a data set. We don't need to discretize the continuous-valued feature data and we refer to file type, access behaviors and fragmentation degree to tag a file.

We predict reserved files based on historical information. Whether to reserve space for a file is a two-category problem. The most commonly used classification algorithms are k-Nearest Neighbor (KNN), Logic Regression, Decision Tree, AdaBoost and Random Forest. Since the feature data are continuous, Decision Tree adopt the algorithm CART. We

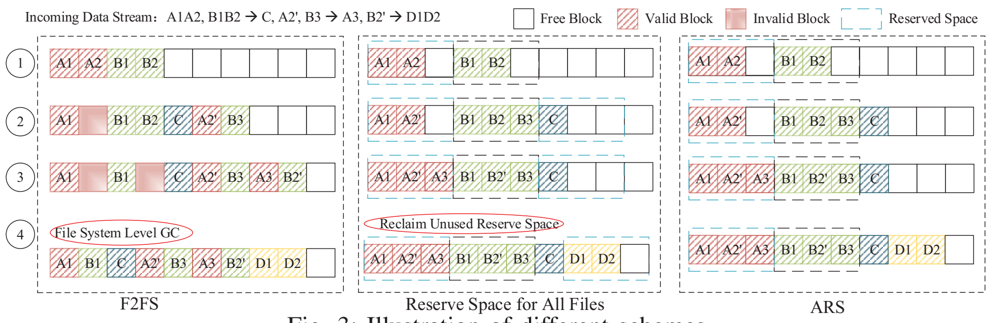

Incoming Data Stream: A1A2, B1B2 → C, A2', B3 → A3, B2' → D1D2
Free Block
Valid Block
Invalid Block
Reserved Space
1
2
3
4
File System Level GC
Reclaim Unused Reserve Space
F2FS
Reserve Space for All Files
ARS

Fig. 3: Illustration of different schemes.

TABLE I: Performance of different classifiers.

Applications	Algorithm	Precision	Recall	Accuracy
Facebook	KNN	0.878	0.762	0.76087
	Logic Regression	0.768	0.574	0.573913
	Decision Tree	0.834	0.878	0.913043
	AdaBoost	0.935	0.93	0.930435
	Random Forest	0.918	0.903	0.904348
Twitter	KNN	0.896	0.694	0.693333
	Logic Regression	0.819	0.773	0.773333
	Decision Tree	0.916	0.885	0.926667
	AdaBoost	0.931	0.92	0.92
	Random Forest	0.942	0.926	0.926667

TABLE II: Normalized computational time of three classifiers.

Applications	Decision Trees	AdaBoost	Random Forest
Facebook	1	3.56	3.04
Twitter	1	4.48	4.28

repeat a process which randomly divides a data set 10 times and output the average results. 75% of the data are training sets. The performance of different classifiers is shown in Table I. Decision Tree, AdaBoost and Random Forest achieve good classification accuracy for both Facebook and Twitter traces. We need to choose one with low cost. Their normalized computational time overheads are shown in Table II. AdaBoost and Random Forest have higher computational overheads than Decision Tree. Although accuracy of AdaBoost on Facebook is slightly higher than Decision Tree by 2%, AdaBoost has a much higher computational overhead by 3.56× as it trains 10 classifiers. Similarly, Random Forest takes more time by up to 4.28×. The training of Decision Tree can be completed within 1 second, so the overhead of profiling and prediction is acceptable. The Decision Tree is easy to understand and achieves good classification accuracy with low computational cost. So we apply Decision Trees to diverse traces and obtain reserved files efficiently. We update the classifier offline. Considering user behaviors and application update frequency [18], we train the classifier when users use the phone least, e.g., 4:00 am, every 10 days. It takes a short time to update the reserved files due to the efficiency of CART. More importantly, we find that some reserved files are the same after updating.

B. Reserved Space

Finding appropriate reserved space is also challenging. If the space is too large, it would waste space. Nevertheless, if the space is too small, it couldn't keep all data of a file. As shown in Fig. 2, when a file fragment is larger than 1 MB, the negative effect of fragmentation is small. Since F2FS performs GC in unit of section (1 section consists of 1 segment here) whose size is 2 MB, aligning reserved space with a GC unit makes the segments of a reserved file less likely to be selected as a victim section. Assuming that the reserved space is 128 KB, the probability of it being selected as victim section is larger than 2 MB whose entire segment is valid blocks. ARS aims to the least GC overhead since the data in the reserved space belonging to a file may be invalid at the same time. So we decide that the initial reserved space is 2 MB. When a reserved file grows more than 2 MB, we preallocate another segment.

C. Reservation Strategy

When free space is deficient, it is better to stop reserving to avoid space being repeatedly occupied and recycled and triggering GC. F2FS_IPU_UTIL is one of the `IPU` policies as mentioned in Section III and its default file system space utilization threshold is 70%. We empirically adopt dynamic update policy for reserved files according to file system space utilization. When the file system space utilization is lower than 70%, reserving space for all target files; when it is between 70% and 80%, updating the old reserved files in-place but stop preallocating space for a new-coming reserved file; when it is higher than 80%, stop reserving and reclaiming unused reserved space. It is easy to reclaim unused reserved space by modifying the reserved flag and freeing up space. Using stepped thresholds is to prevent system performance from bumping.

We only make some changes at the file system level to deploy ARS. So the applications don't need to make complicated changes, nor does the flash memory. We implement ARS as a module in 657 lines of C code based on the traditional F2FS in AOSP (Android Open Source Project) 9.0 [19] and the reads and writes of unreserved files have no changes.

V. Evaluation

In order to accurately understand the performance implication of ARS, we implement these schemes on Hikey960 and Huawei Mate10 Pro. We compare the performance of ARS with traditional F2FS and `IPU`, and let `ARS+IPU` be the performance of ARS works with F2FS in-place updates. The F2FS_IPU_FSYNC policy is adopted.

A. File and Free Space Fragmentation

At first, we repeat the motivation experiment as shown in Fig. 1. The performance of ARS are similar to `ARS+IPU`, which indicates that ARS is compatible with the in-place update scheme provided by F2FS. An extent is a piece of

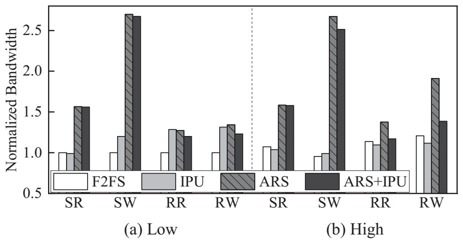


Fig. 4: The impact of ARS on I/O throughput.

data with consecutive addresses which we use to measure file fragmentation. The fragmented file in traditional F2FS has 32,768 extents while in ARS has 1 extent, so ARS significantly eliminates file fragmentation. The throughput of fragmented file in ARS performs 46.93% and 88.97% better than that in traditional F2FS on Hikey960 and Mate10 Pro, respectively. ARS also shortens the latency by 32.27% and 47.5% on Hikey960 and Mate10 Pro, respectively. The degree of performance increase of Hikey960 and Mate10 Pro is different due to hardware differences, but the upward trend is the same. For the sake of narrative clarity, we don't report the performance of Huawei Mate10 Pro in subsequent evaluations. Although generating fragmented files is identical, traditional F2FS writes file data one by one while ARS writes them in reserved place which reduces file fragmentation and prevents from cutting off other files. `IPU` has limited effect on reducing fragmentation because mobile systems are subject to costly random writes with frequent `fsync` and ARS changes file layout while `IPU` does little.

After a lot of create, update, delete or truncate operations, there is severe free space fragmentation at high space utilization of file system where big non-SQLite files also have file fragmentation. We explore whether ARS is effective in reducing file and free space fragmentation at high space utilization. We design a fragmentation benchmark: (1) we set $file\ size = 2 \times 10^{-18} * the\ integer\ power\ of\ 2\ closest\ to\ the\ data\ partition\ size$, (2) fill files until the file system space utilization reach 90%, (3) delete files by intervals, (4) set file size that are half of the file in (1). Repeat (2)-(4) until file size is 4 KB. There are frequently applications updates, accumulated text information, and deletion of out-of-date pictures and videos in long-term use of smartphones. We use this fragmentation benchmark to simulate rapid age. We follow four system space states in [16]: `Low` (the pristine state), `Moderate` (less than 80%), `High` (between 80% and 95%) and `Full` (higher than 95%). After quickly aging the file system using a fragmentation benchmark, the file system utilization is 84%, which is at `High`.

Fig. 4 shows the I/O performance of file reads and writes with respect to the baseline, the traditional F2FS at `Low`. We set up 11 reserved files. ARS gives 56.55%, 169.76%, 27.13% and 34.28% more promotion than traditional F2FS at `Low` of sequential read, sequential write, random read, and random write, respectively. The corresponding increases are 47.65%, 179.62%, 20.8% and 58.12% at `High`. The number of a fragmented file extents at `High` in traditional F2FS and `IPU` is around 32,768 and the extents number in ARS and `ARS+IPU` is 3 and 5, respectively. While a fragmented file

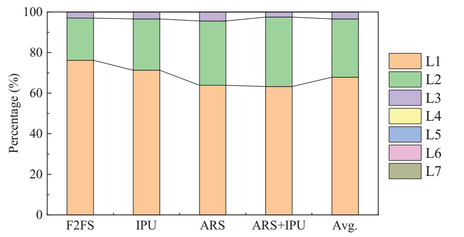


Fig. 5: A breakdown of different levels of free space fragments.

has severe fragmentation in traditional F2FS and `IPU`, its fragmentation is far reduced in ARS at `High`. So the overheads of I/Os in ARS are smaller than that in traditional F2FS and `IPU`. Fragmentations decrease the I/O performance because they affect merge operations of I/O scheduler and cause more Android I/O stack overheads. Besides, the improvements result from the highly synchronous, multi-threaded writing behaviors of applications.

We extend `FS` (Fragment Size) [16] to evaluate free space fragmentation. We investigate the percentage of free space fragments of different sizes. The thresholds are 4 KB, 8 KB, 16 KB, 32 KB, 64 KB and 128 KB, which respectively correspond to L1 to L7, e.g., L2: $4\ KB < fragment \leq 8\ KB$. L1 starts with 4 KB because we observe there are many 4 KB free space fragments, and 64% of the I/O operations involve data with size less than 4 KB [14]. The number of fragments above L6 is small and a free space fragment larger than 128 KB is large enough for an unreserved file as shown in Fig.2, so `FS` is divided into 7 levels. The distribution of free space fragmentation is plotted in Fig. 5. The L1 fragments contribute to the largest fraction of free space fragmentation in all schemes, ranging from 63% to 77%. The free space is fragmented into small pieces at `High`. The number of free space fragments larger than 16 KB is too small to be displayed in graph. Compare to traditional F2FS, `IPU` reduces the L1 free space fragments, but ARS has the least L1 and the most L3 free space fragments of all schemes. GC of F2FS has a negative impact on performance which is exaggerated by free space fragmentation and we monitor the GC overhead at `High` by *cat stat*. ARS gives 23.15%, 7.76%, 78% and 38.36% more reductions than traditional F2FS for GC count, move blocks, read I/Os and write I/Os, respectively. ARS gives 13.29%, 0.13%, 67.43% and 40.02% more reductions than `IPU` accordingly. File and free space fragmentation influence each other. ARS reduces file fragmentation dramatically and also alleviates free space fragmentation, so ARS gains I/O performance improvements and GC overhead reductions.

B. SQLite

Fig. 6 gives SQLite performance measured by transactions per second (TPS). We measure three types of transactions that comprise 1,000 records with 2 threads under the default `TRUNCATE` journal mode. We run SQLite to get data set and use the Decision Tree to determine reserved files. The fragmentation benchmark fills file system up to 70% for reserved files to work. Then the file system space utilization is at `Moderate`. `IPU` performs worse than traditional F2FS. Since the performance of SQLite operations is proportional to

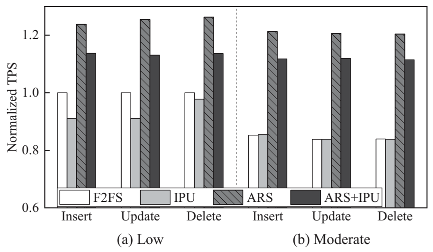


Fig. 6: SQLite performance of different schemes.

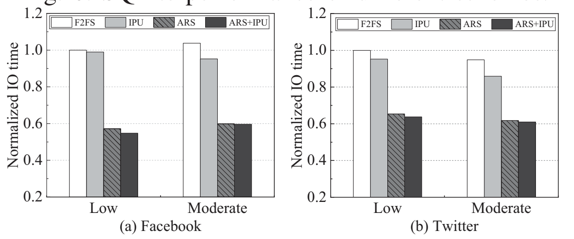


Fig. 7: Running time to complete Facebook and Twitter I/Os.

the performance of `write()` followed by `fsync()`, ARS outperforms traditional F2FS by up to 1.26×.

C. Running Time of Applications

We repeat the decision tree learning process as on SQLite for Facebook and Twitter traces supplied by Mobibench. The normalized running time to complete these traces is depicted in Fig. 7. Since ARS changes files layouts, which gathers reserved files data and reduces file fragmentation, it decreases the running time by 41.72%, 35.38% (for Facebook) and 29.83%, 24.14% (for Twitter) compare to `IPU` at `Low` and `Moderate`, respectively.

D. Space and Time Overhead

There is almost no reserved space to waste since the reserved files grow constantly. The last reserved segment may waste space, but ARS reclaims unused space in time. Furthermore, reserved objects are just a few files, and only the data of reserved files are updated in-place but there is no change in inode management, we believe that ARS is consistent.

We compare the average file creation time among different schemes because ARS increase operations when creating a file. Compare with traditional F2FS, ARS spends 3.6%, 1.5% more creation time for measuring *synthetic workloads* and *SQLite* respectively, but 5.1% less creation time for measuring *applications*. Although we exhibit the throughput in Fig.4, the running time is significantly shortened, e.g., the random read latency of ARS is shorter than traditional F2FS by 1,136 millisecond. The reduction in running time by reducing fragmentation is larger than the increment of creation time.

In summary, whether the file system space utilization is `Low`, `Moderate`, or `High`, ARS effectively reduces file and free space fragmentation for better performance. Since our system is not full while measuring the performance of individual files, the effect of high space utilization on fragmentation is not as severe as expected. ARS is easy to be ported to other log-structured file systems as a module and is superior than other schemes since ARS does not replicate data, reduces GC and selectively reserves space.

VI. Conclusion

In this paper, we first demonstrate that smartphones generate lots of fragments whether in traditional or providing in-place updates F2FS. We propose an adaptive reserved space (ARS) scheme which uses decision trees to classify reserved files and reserves space for them. The experimental results show that ARS is effective in reducing file and free space fragmentation, improving I/O performance and reducing GC overheads. ARS also outperforms traditional F2FS and F2FS with in-place updates under SQLite, Facebook and Twitter workloads.

Acknowledgment

This work was supported in part by National Defense Preliminary Research Project (31511010202), National Key R&D Program of China NO.2018YFB10033005, NSFC No.61832020, 61821003, 61772216, Hubei Province Technical Innovation Special Project (2017AAA129), Wuhan Application Basic Research Project (2017010201010103), Fundamental Research Funds for the Central Universities.

References

[1] T. S. Portal, "Number of smartphone users worldwide from 2016 to 2021 (in billions)." https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, 2018.

[2] S. Kadekodi, V. Nagarajan, G. R. Ganger, and G. A. Gibson, "Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems," in *USENIX Annual Technical Conference*, pp. 691–703, 2018.

[3] C. Ji, L.-P. Chang, L. Shi, *et al.*, "An empirical study of file-system fragmentation in mobile storage systems," in *HotStorage*, 2016.

[4] A. Conway, A. Bakshi, Y. Jiao, *et al.*, "File systems fated for senescence? nonsense, says science!," in *FAST*, pp. 45–58, 2017.

[5] S. S. Hahn, S. Lee, C. Ji, *et al.*, "Improving file system performance of mobile storage systems using a decoupled defragmenter," in *USENIX Annual Technical Conference*, pp. 759–771, 2017.

[6] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *FAST*, pp. 273–286, 2015.

[7] 96Boards, "Hikey 960." https://www.96boards.org/product/hikey960/, 2018.

[8] H. T. Co., "HUAWEI Mate 10 Pro." https://consumer.huawei.com/en/phones/mate10-pro/, 2018.

[9] K. A. Smith and M. I. Seltzer, "File system aging increasing the relevance of file system benchmarks," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, pp. 203–213, ACM, 1997.

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

[11] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Generating realistic impressions for file-system benchmarking," *ACM Transactions on Storage (TOS)*, vol. 5, no. 4, p. 16, 2009.

[12] W. Jannen, J. Yuan, Y. Zhan, *et al.*, "Betrfs: A right-optimized write-optimized file system," in *FAST*, pp. 301–315, 2015.

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

[14] S. Jeong, K. Lee, S. Lee, *et al.*, "I/O stack optimization for smartphones," in *USENIX Annual Technical Conference*, pp. 309–320, 2013.

[15] Y. Liang, C. Fu, Y. Du, *et al.*, "An empirical study of F2FS on mobile devices," in *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–9, 2017.

[16] C. Ji, L.-P. Chang, S. S. Hahn, *et al.*, "File fragmentation in mobile devices: Measurement, evaluation, and treatment," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2062–2076, 2018.

[17] S. Media, "Fio." http://freshmeat.sourceforge.net/projects/fio, 2015.

[18] U. Kumar, "Understanding Android's application update cycles." https://www.nowsecure.com/blog/2015/06/08/understanding-android-s-application-update-cycles/, 2015.

[19] Google, "About the Android open source project." https://source.android.com/, 2018.