Content-aware Trace Collection and I/O Deduplication for Smartphones

Bo Mao[1], Suzhen Wu[2], Hong Jiang[3], Xiao Chen[2], Weijian Yang[2]
[1]Software School, Xiamen University, China
[2]Department of Computer Science, Xiamen University, China
[3]Department of Computer Science & Engineering, University of Texas at Arlington, USA
Corresponding author: Suzhen Wu (suzhen@xmu.edu.cn)

***Abstract*—In this paper, we first introduce a trace collection tool specifically designed to capture the I/O requests with important content features in Android-based smartphones, which are critically important but rarely available in content-aware designs and optimizations such as JProbe and Netlink. Based on the analysis of the traces collected from 15 popular applications, we find that 20% to 40% of the I/O requests on the I/O critical path of the storage stack are redundant and this data redundancy is minimally shared among different applications. Based on this key observation, we propose a content-aware optimization, called APP-Dedupe, that applies data deduplication on the I/O critical path to improve both performance and efficiency by reducing write amplification and improving GC efficiency of the flash storage on Android smartphones. The evaluation results show that APP-Dedupe reduces the GC overhead by an average of 41.5%, reduces the response times by up to 15.4% and reduces the amount of write data by an average of 45.2%.**

***Index Terms*—Smartphones; I/O Deduplication; Trace Collection; Flash-based Storage**

I. Introduction

Storage is one of the key factors affecting the overall system performance of the Android-based smartphones [14], [16], [27]. The mobile applications, often referred to as "APP", in Android-based smartphones generate I/O requests that have different characteristics than those generated by non-mobile applications. The storage subsystem of smartphones usually relies on flash-based embedded Multi-Media Controller (eMMC) memory, with either a disk file system (such as Ext4) or a flash file system (such as F2FS [19]).

Generally speaking, the storage stack of current smartphones faces three challenges. First, the performance tends to degrade after repeated usages, particularly writes, due to the physical characteristics of the flash memory, also one of the reasons why smartphones slow down over time [5], [25]. Second, one of these physical characteristics of flash memory is its limited life cycles [37], [42], [31], *i.e.*, the number of times each cell can be programmed/written before it fails, and causes the flash storage to get sluggish after repeated usages, which affects the storage reliability of smartphones. Third, the cost of upgrading the flash capacity from one level to the next level, e.g., from 16GB to 32GB, amounts to nearly 100 USD for most smartphones (for example, Apple iPhones). Therefore, these challenges, pointing to the measures of performance, reliability and cost, suggest that it is important to (1) understand the mobile applications and how they interact with the flash-based eMMC and (2) optimize to reduce write traffic to the flash-based eMMC in smartphones.

Data deduplication and its applications in flash-based storage systems have been well studied in the literatures [6], [21], [23], [30], [35], [40]. Some studies have shown that by leveraging the deduplication technology to reducing the write traffic, the system performance and reliability of the storage stack of conventional, non-mobile systems can be significantly improved [3], [10], [32]. However, the unique characteristics of mobile devices and mobile applications make straightforwardly applying deduplication in Android-based smartphones both less effective and more challenging. For example, the mobile devices have much smaller memory capacity than non-mobile systems, which implies that mobile applications must be made memory efficient to attain acceptable performance. On the other hand, the user-facing nature of smartphones implies, and confirmed by our experimental observation, that only one application is usually running in the foreground while all the other opened applications are hung up in the background in the Android-based smartphones.

To make data deduplication effective and efficient in smartphones, we need to understand and gain insight into the data redundancy and unique characteristics of mobile applications by collecting and analyzing content-aware application traces. Unfortunately, due to the dataset privacy leakage risk, content-aware trace collections are rarely done in storage systems [9], let alone publicly available traces with content features, with the exception of the FIU department traces available in the SNIA trace repository [18]. To address this problem, we design a low-overhead content-aware trace collection tool, which can be used both offline and online. Using this tool, we collected traces of 15 popular mobile applications. Our workload analysis of 15 popular mobile applications reveals that an average data redundancy of 33.1% exists in mobile applications but this redundancy is minimally shared among these applications.

Therefore, we propose APP-Dedupe for Android-based smartphones to address the aforementioned challenges. Instead of treating data chunks from all application streams equally,

978-1-5386-2032-8/17/$31.00 ©2017 IEEE

APP-Dedupe organizes the hash (fingerprint) index in an application-aware way, effectively grouping the hash index of the same application (short for APP) together and dividing the whole hash index into different groups based on the application types. When an application is running in the foreground, APP-Dedupe loads the corresponding hash index of the application into the memory, thus improving the efficiency of the memory for the hash index. Moreover, it groups the data chunks of the same application together on the flash to alleviate the read amplification problem (*i.e.*, fragmentation caused by deduplication) and exploits the spatial locality to improve the read performance. The extensive trace-driven experiments conducted on our lightweight prototype implementation of APP-Dedupe show that APP-Dedupe reduces the GC overhead by an average of 41.5%, reduces the response times by up to 15.4% and reduces the amount of write data by an average of 45.2%.

The contributions of this paper are threefold. First, we design a low-overhead content-aware trace collection tool that captures the I/O requests in the storage stack in Android-based smartphones. Second, we collect the traces with content features from 15 popular applications and perform in-depth I/O analysis. We find that 20% to 40% of the I/O requests on the I/O critical path of the storage stack are redundant and this data redundancy is minimally shared among different applications. To the best of our knowledge, currently no such study exists for the Android smartphones. Third, we propose APP-Dedupe to improve the storage efficiency of Android smartphones. The trace collection tool, the 15 traces and the image of the prototype system are available for academic purposes.

The rest of this paper is organized as follows. Section 2 presents the trace collection tool and workload characteristics in Android-based smartphones. The design and implementation of APP-Dedupe is presented in Section 3. Section 4 describes the performance results through the extensive evaluations on the APP-Dedupe prototype. The conclusion is given in Section 5.

II. Trace Collection and Analysis

In this section, we first present the design of a content-aware trace collection tool in Android-based smartphones. Then we analyze the workload characteristics of the traces collected by this tool to motivate our App-Dedupe study.

A. Content-aware trace collection

While trace collections on enterprise storage systems have been well studied, there is limited effort on trace collection in mobile systems. In particular, currently only MOST [12] and BIOtracer [41] are designed for I/O trace collections in Android-based smartphones and they only capture the I/O requests behaviors (e.g., size, read/write patterns, etc.) without including any content values or features. Yet, it is the content features of the traces that enable one to analyze the data redundancy characteristics of I/O accesses in Android-based smartphones. For this reason, we design a content-aware trace collection tool, called *MobileCT*. MobileCT collects the

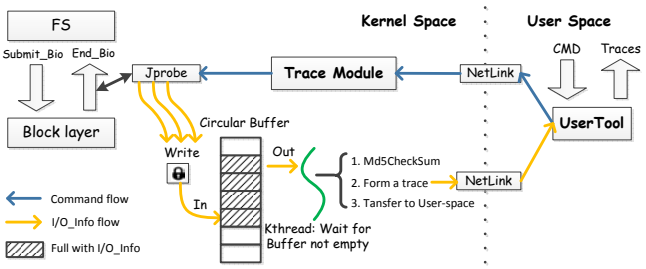

Fig. 1. Trace collection workflow in MobileCT.

traces that contain both the basic I/O request information and the content features as follows. First, it captures the basic information of I/O request via the *bio* structure, including time, R/W, offset and size, while copying the data of the request for the subsequent hash computing of the content. Second, it splits the data into 4KB chunks and calculate the MD5 fingerprints for each of the chunks. Finally, the basic information and hash values of the I/O requests are recorded in the trace file and transferred to the user level.

There are two major challenges facing content-aware trace collection in Android-based smartphones, namely, how to anonymize the contents to protect the privacy of the users [30] and how to minimize the interference between the trace capturing operations and the user I/O requests to reduce collection overhead [41]. For the first privacy challenge, similar to the FIU traces [18], we also use the MD5 fingerprints to represent the content feature for the deduplication research without leakage of the location or personal information [36]. For the overhead challenge, since the processing resources in Android-based smartphones are limited and hash computing for chunk fingerprints can be resource demanding, the aforementioned interference, if not avoided or minimized, can adversely affect the application performance and/or the accuracy of the traces. To address this challenge, MobileCT uses a circular buffer to temporarily store the write data and delay the subsequent MD5 computing of the data chunks to avoid the interference and contention on the CPU resources. Figure 1 shows the trace collection workflow in MobileCT. JProbe is a servlet for inspecting the bio→end_io() function and Netlink_sock is used for the data transfer between the user space and the kernel space.

B. Workload characteristics

The traces presented in this paper are collected from 15 applications on the Google Nexus 5 smartphone (running Android 5.0.1 with Linux Kernel 3.4). We compare the chunk fingerprints of the 15 data sets using the chunk-level deduplication with 4KB chunk size. The trace characteristics are summarized in Table I, which shows that the data redundancy of the mobile applications is between 20% to 40%, with an average of 33.1%. Of particular interests are the findings that the IOPS is less than 10 for all the 15 applications and the write requests dominate in the mobile applications. These

findings of workload characteristics are consistent with the previous studies of mobile applications [12], [41].

TABLE I
THE KEY CHARACTERISTICS OF THE 15 MOBILE APPLICATIONS.

APPs	*Redundancy*	*IOPS*	*Size (MB)*	*Write Ratio*
Qiu	40.5%	1.6	55.0	87.9%
58City	32.2%	2.6	153.3	69.4%
Baidu Tieba	39.5%	4.3	213.5	88.6%
Game2048	31.7%	1.7	63.2	89.2%
Meitu	31.3%	5.6	131.6	41.9%
Moji Weather	32.4%	2.6	82.1	63.4%
Opera Browser	33.1%	3.0	72.7	57.5%
Fruit Cool	36.6%	6.0	159.9	47.1%
Sohu News	32.2%	2.0	138.2	84.2%
Tencent	34.1%	1.5	82.6	95.4%
Pea Pod	35.4%	2.1	237.0	67.8%
Wechat	30.0%	4.6	345.5	90.4%
Weibo	29.0%	1.4	97.0	88.5%
Xiami Music	27.3%	7.4	122.9	39.4%
Youdao Dict	31.9%	9.2	136.5	35.2%

The most interesting finding, as shown in Table II, is that the amount of data redundancy shared between any two different mobile applications, or the percentage of shared redundancy, is minimal. The percentage of shared redundancy, the percentage entry in the table, is the percentage ratio of the number of common redundant data chunks between the two applications (row and column) to the total number of data chunks of the row application. From Table II, we can see that the redundant data chunks shared by any two different applications are less than 5% for most cases , which implies that the amount of redundant data shared by different applications is negligible. The reason is that different applications usually have different data contents and data formats. The results are consistent with the previous studies on non-mobile applications [7], [8], [33] and indicate that there is very little overlap between hash indexes of any two different applications. The significance of this finding is that it makes it possible to effectively group the hash index of the same application together and partition the whole hash index into multiple small segments according to the application types. This in turn helps optimize hash index locality for optimal cashing efficiency.

C. Motivation

The storage subsystem affects the performance of the mobile applications in Android-based smartphones [4], [14], [22]. Recent studies have shown that the overlap between the file system journaling (such as EXT4) and the database journaling (such as SQLite) activities, also referred to as journal of journaling[17], [20], [29], is the root cause of the inefficiency for the flash storage in smartphones. Existing solutions to this problem to either reduce the journaling overhead in the SQLite database, such as reducing the SQLite journaling I/Os through multi-version B-tree [17], minimize the synchronization overhead, such as WALDIO [20], or optimize the file systems, such as MobiFS [28] and F2FS [19]. However, none of them exploits the content redundancy characteristics in the mobile storage subsystems, which has the potential to improve not only application performance but also system reliability and efficiency (space and energy).

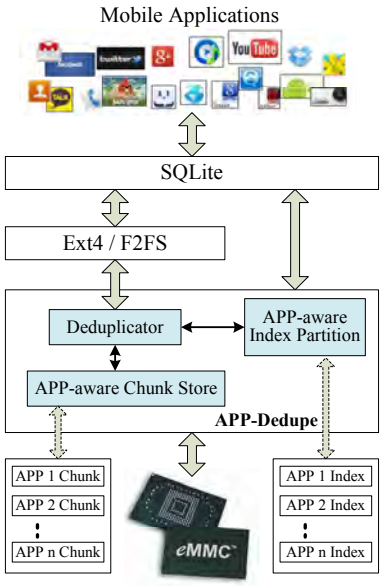

Fig. 2. System architecture of APP-Dedupe.

In this paper, we revisit the flash performance issue in Smartphones from a fresh perspective based on the content feature analysis of the mobile applications. Recent studies in the literature indicate that reducing the amount of I/Os on the critical path can be much more effective and efficient than optimizing I/Os in storage systems [24], [15]. I/O deduplication on the I/O critical path can reduce the I/O traffic at the cost of computing and memory overhead. Motivated by the importance of the mobile storage space, performance and reliability, combined with the observations from the workload studies, we propose APP-Dedupe to improve the storage efficiency of Android-based smartphones.

III. THE DESIGN OF APP-DEDUPE

In this section, we first present an architecture overview of APP-Dedupe, which is then followed by detailed descriptions of the main functional components of APP-Dedupe.

A. Architecture overview

Figure 2 shows a system architecture overview of our proposed APP-Dedupe in the context of the storage subsystem in the Android-based smartphones. APP-Dedupe sits below the file system and the SQLite database and thus can be easily incorporated into any existing platforms to accelerate their storage subsystem performance. Moreover, APP-Dedupe is independent of the upper file system, which makes APP-Dedupe amenable to be deployed in a variety of environments, including the newly proposed MobiFS and F2FS.

APP-Dedupe has three main functional components: *Deduplicator*, *APP-aware Index Partition*, and *APP-aware Chunk Store*. The Deduplicator module is responsible for splitting the incoming write data into data chunks, calculating the hash

TABLE II
THE PERCENTAGE OF SHARING OF DATA REDUNDANCY BETWEEN ANY TWO DIFFERENT APPLICATIONS. NOTE: DUE TO THE SPACE LIMIT, ONLY THE FIRST WORD OF AN APPLICATION'S NAME IS PRESENTED.

APPs	*Qiu*	*58City*	*Baidu*	*Game2048*	*Meitu*	*Moji*	*Opera*	*Fruit*	*Sohu*	*Tencent*	*Pea*	*Wechat*	*Weibo*	*Xiami*	*Youdao*
Qiu	**40.5%**	0.2%	1.0%	0.1%	0.3%	0.0%	0.1%	0.4%	0.2%	0.0%	0.1%	0.5%	1.5%	0.2%	0.3%
58City	0.3%	**32.2%**	3.3%	1.3%	3.6%	3.2%	4.3%	3.9%	4.9%	1.2%	3.9%	2.4%	1.6%	4.2%	3.5%
Baidu	0.4%	6.5%	**39.5%**	0.2%	2.4%	5.5%	7.1%	4.4%	3.9%	0.0%	3.5%	3.2%	3.6%	3.6%	6.4%
Game2048	0.3%	0.7%	1.0%	**31.7%**	1.0%	2.0%	2.9%	1.5%	0.7%	2.0%	1.4%	0.3%	1.5%	1.0%	2.4%
Meitu	0.3%	3.1%	2.2%	1.3%	**31.3%**	2.2%	5.9%	6.4%	1.5%	1.1%	2.3%	1.3%	1.6%	4.4%	3.1%
Moji	0.1%	2.7%	4.7%	2.8%	2.3%	**32.4%**	6.5%	5.5%	3.5%	0.8%	3.5%	1.4%	1.5%	3.8%	3.8%
Opera	0.2%	1.5%	2.7%	3.4%	3.7%	5.7%	**33.1%**	4.8%	3.0%	1.7%	1.9%	2.1%	1.1%	1.4%	4.2%
Fruit	0.3%	1.1%	2.5%	3.2%	3.8%	6.4%	4.7%	**36.6%**	2.6%	0.5%	2.2%	2.6%	2.0%	1.7%	2.7%
Sohu	0.4%	5.9%	3.5%	1.3%	1.6%	5.3%	4.5%	3.8%	**32.2%**	1.1%	2.5%	3.3%	4.6%	2.5%	4.1%
Tencent	0.0%	0.8%	0.0%	2.7%	1.8%	0.8%	2.1%	0.6%	0.8%	**34.1%**	0.5%	0.2%	0.6%	2.2%	2.8%
Pea	0.2%	6.1%	2.9%	3.9%	4.5%	7.3%	4.5%	5.0%	3.6%	1.2%	**35.4%**	2.1%	1.1%	6.2%	3.9%
Wechat	1.3%	3.5%	4.8%	0.1%	0.7%	4.2%	3.5%	5.5%	4.3%	0.2%	1.8%	**30.0%**	5.0%	1.9%	3.0%
Weibo	0.4%	0.4%	1.5%	0.0%	0.6%	0.4%	0.2%	0.9%	1.4%	0.2%	0.1%	2.1%	**29.0%**	0.4%	0.5%
Xiami	0.2%	5.0%	2.6%	3.3%	5.6%	6.5%	3.6%	3.0%	2.9%	9.3%	6.1%	2.1%	1.3%	**27.3%**	3.6%
Youdao	0.2%	1.7%	2.7%	5.8%	3.0%	4.6%	5.4%	3.7%	3.2%	2.1%	5.2%	2.1%	1.2%	1.4%	**31.9%**

value of each data chunk and identifying whether a data chunk is a duplicate. The APP-aware Index Partition (short for AIP) module divides the whole hash index into small subsets based on the application types. When an application is in the active state, the corresponding hash index subset is loaded from the back-end eMMC storage into the memory. AIP swaps out the cached hash index to the back-end eMMC storage when the corresponding application is hung out in the background. The APP Chunk Store (short for ACS) module groups the data chunks of the same application together to alleviate the data fragmentation problem [13] by fully leveraging the spatial locality of the user accesses.

B. APP-aware Index Partition (AIP)

Figure 3 illustrates the write workflow in APP-Dedupe. There are two key data structures used to deduplicate and redirect the I/O requests, and identify the popular hash index entries, namely, *Map_table* and *App_index_table* , as shown in Figure 3. While Map_table keeps all the information of the deduplicated write requests whose write data are already stored on the back-end eMMC storage, App_index_table maintains the fingerprints of the data chunks according to the specific application. The mapping between the items in the Map_table and the items in the APP_index_table is *many-to-1*. It means that an LBA (Logical Block Address) can only be linked to a unique and distinctive physical data block, *i.e.*, PBA (Physical Block Address), but multiple LBAs can be linked to the same PBA.

In order to reduce the memory and processing overhead of storing and querying the large hash index, AIP only swaps the corresponding index subset in the memory when the application is in the active state. App_index_table is organized in an LRU form and maintains the frequency of write requests to each data chunk (PBA) by using the *Count* variable (initialized to "1"), as shown in Figure 3. When a write request hits App_index_table, the count value of the corresponding index entry in App_index_table is increased by 1, which captures the temporal locality and frequency of write requests to this PBA. The Count variable is also used to prevent the referenced data blocks from being modified or deleted.

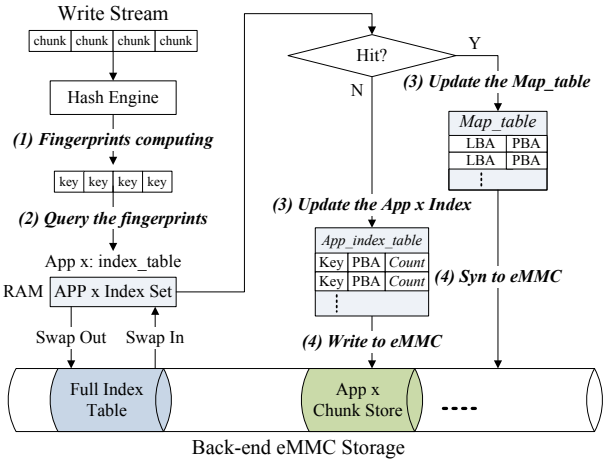

Fig. 3. The write workflow in APP-Dedupe.

When a write request arrives, APP-Dedupe splits the write data into multiple fixed-size data chunks and calculates their fingerprints. If a fingerprint hits App_index_table, meaning that the corresponding data chunk is a duplicate, the corresponding Count value in App_index_table is incremented. APP-Dedupe only updates Map_table for the duplicate data chunks, and synchronizes Map_table to the back-end eMMC storage periodically. Otherwise, a new hash index entry is inserted into App_index_table and the data chunk is directly written to the back-end eMMC storage.

C. APP-aware Chunk Store (ACS)

Our experimental observation indicates that the amount of data redundancy shared by different applications is negligible and most data redundancy exists among the data chunks of the same application. To alleviate read performance degradation problem caused by the data fragmentation associated with data deduplication, the ACS module stores the data chunks of the same application in the same container. Moreover, by exploiting the semantic information of the file access correlation, the ACS module effectively groups the data chunks of these files together, thus allowing the subsequent read requests

to fetch them in a single I/O request. In this way, the data fragmentation problem, which can degrade read performance (read amplification), is alleviated by concentrating the read accesses to a single container, thus improving the restore/read performance.

When a read request arrives at the block layer, having missed the upper-level cache, APP-Dedupe first checks whether the read request hits Map_table. If the read request misses Map_table, the read request is directly submitted to the block device layer. Otherwise, the address of the request will be replaced by one or more addresses according to Map_table, depending on whether the read request fully hits Map_table or not. If fully hit (*i.e.*, all LBAs of the constituent data chunks of the request are found), the read request will be replaced by the new address in Map_table. Otherwise, partially hit (*i.e.*, not all LBAs of the constituent data chunks of the request are found), the read request will be split into multiple new read requests based on the locations of the constituent data chunks of the original read request. Then, the newly generated read request(s) is (are) submitted to the block layer. After the completion of these read requests, the read data is reconstructed and returned to the upper layer.

IV. Performance Evaluations

In this section, we first describe the experimental setup and methodology. Then we evaluate the performance of APP-Dedupe through both benchmark-driven and trace-driven evaluations.

A. Experimental setup and methodology

We implement a prototype of APP-Dedupe on the Google Nexus 5 smartphone, with Qualcomm MSM8974 Quadcore 2.3 GHz, 2 GByte DRAM, 16 GByte eMMC storage, and running Android 5.0.1 with Linux Kernel 3.4. The system and application software is configured with the default settings without data deduplication as the baseline system. We use both the benchmark workload, *i.e.*, the Monkey tool, and the trace replay workload to evaluate the effectiveness of APP-dedupe. The Monkey tool is a program that runs on the smartphones to generate for mobile applications pseudo-random streams of user events such as clicks, touches, and gestures, as well as a number of system-level events [26]. In our evaluation it runs to generate pseudo-random streams of user- and system-level events for the 15 applications listed in Table I. Moreover, A1 SD Bench is used to test the I/O read and write throughput [2]. The trace replay evaluations are driven by the 15 collected traces that are shown in Table I. In order to evaluate the internal GC activities within eMMC, we also incorporate the deduplication functionality into the SSD-based DiskSim simulator [1]. The reserved free space is set to be 15% and the greedy cleaning policy is used in the simulation experiments.

B. Performance results

During the benchmark evaluations driven by the Monkey tool, we use the *iostat* command to monitor the CPU utilization and I/O statistics and the *dumpsys* command to monitor the memory usage. Figure 4 shows the memory usage and CPU utilization. It indicates that, compared with the baseline system, APP-Dedupe incurs very little memory overhead, by no more than 2.5% and with an average of 1.6%. The reason is that APP-Dedupe only loads a subset of the hash index into the memory when the corresponding APP is running in the foreground. This memory overhead is expected to further diminish given the trend of increasing memory capacity in smartphones from one generation to the next. Similarly, the processing overhead, measured in CPU utilization, is also minimal, by no more than 2%. The reason is that the CPU resource in smartphones is usually idle during data transmission [27]. For example, a recent study has revealed that Android smartphones spend a significant portion of their CPU active time (up to 58%) waiting for storage I/Os to complete [27]. The very low memory and processing overheads measured here, combined with the fact that user interactions with smartphones are much less intensive than those with the server and enterprise environment, as evidenced in Table I and previous studies [12], [41], make the I/O deduplication a feasible solution in Android based smartphones.

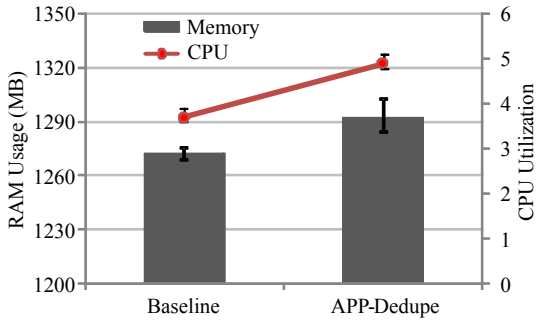

Fig. 4. The memory usage and CPU utilization under the benchmark evaluation driven by the Monkey tool.

In the benchmark evaluation, we also compare the total amounts of written data for different schemes, as shown in Figure 5. Note that "APP-Dedupe generated" means the total amount of data generated in the APP-Dedupe system and "APP-Dedupe Written" means the total amount of data written to the back-end eMMC storage in the APP-Dedupe system. First, APP-Dedupe generates more data than the baseline system because the deduplication operations incur extra metadata overhead. Second, APP-Dedupe reduces the amount of write data to the back-end eMMC storage by an average of 45.2%. The significant reduction in write data leads directly to a notably improved storage efficiency and reduced cost.

Figure 6 shows the system throughput normalized to the baseline system under different access patterns driven by the A1 SD Bench. APP-Dedupe improves the sequential write throughput by 11.5% but degrades the sequential read throughput by 30.0%. The reason for the degraded read performance stems from the fact that deduplication causes the data to be scattered across the flash memory, *i.e.*, the known data fragmentation problem [39]. Though APP-Dedupe uses

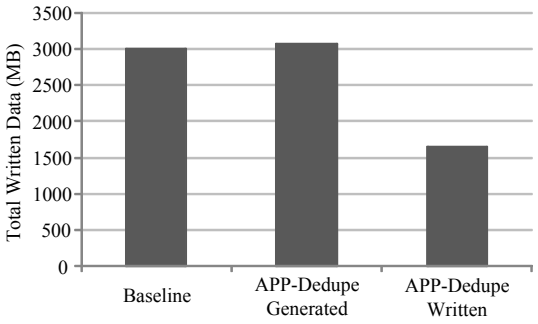

Fig. 5. The total amount of data written to the back-end eMMC storage driven by the Monkey tool.

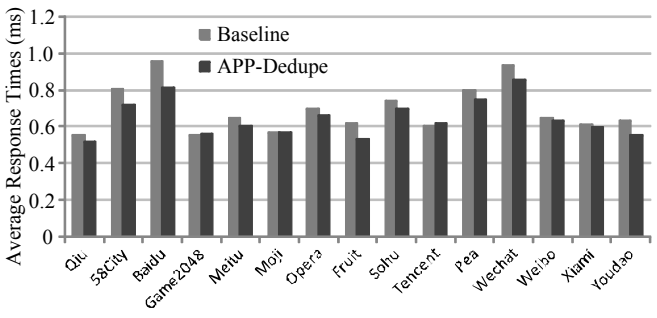

Fig. 7. Average response times in the evaluation driven by the 15 traces.

APP-aware chunk store to group the related chunks together, accesses to the deduplicated sequential read data are no longer sequential, unlike the system without deduplication. Moreover, since the bandwidth of eMMC in smartphones is much lower than that of the enterprise SSDs, the performance gap between random read and sequential read for eMMC is significant. The increased write throughput comes from the reduced write data. In contrast, both the baseline system and the APP-Dedupe system perform similarly in random accesses. The reason is that eMMC's random access performance is much lower than its sequential access performance, which overshadows both the overhead and performance improvement of deduplication for the random accesses.

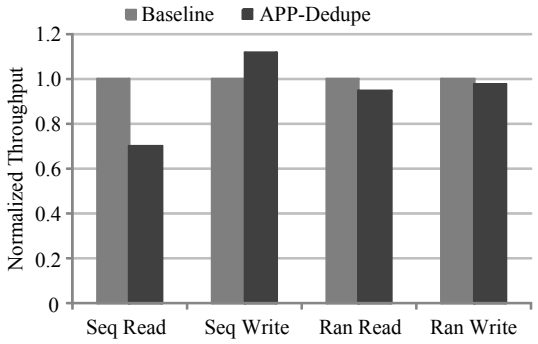

Fig. 6. The system throughput normalized to the baseline system under different access patterns driven by the A1 SD Bench.

Figure 7 shows the average response times in the evaluation driven by the 15 traces, indicating that APP-Dedupe reduces the average response times of the baseline system by up to 15.4% with an average of 6.2%. The reasons are twofold. First, APP-Dedupe removes a large portion of the redundant write requests, thus significantly reducing the request delays. Second, by reducing the write traffic to the flash-based eMMC, the internal GC activities are also reduced, thus improving both the read and write performances. On the other hand, it is interesting to notice that APP-Dedupe increases the average response times of the Game2048 and Tencent applications by 1.1% and 2.3%. The reason is that, while these two applications have relatively very small amount of write data and low IOPS to begin with, making the amounts of reduced write requests and GC activities very small and limiting the benefits of deduplication. These limited benefits from deduplication are more than offset by the hash computing overhead incurred by deduplication.

Figure 8 shows the total GC counts within the eMMC device in the evaluation driven by the 15 traces, inicating that APP-Dedupe reduces the GC counts by up to 58.7% with an average of 41.5%. The reason is that the GC frequency in flash memory is highly correlated to the amount of data written to it, *i.e.*, the more the write data, the higher the GC frequency. The large fraction of write data reduced by APP-Dedupe leads directly to reduced GC activities [34], [38]. Moreover, by reducing the GC counts within flash-based eMMC, APP-Dedupe also improves the reliability and enhances the lifespan of the smartphone storage system [3].

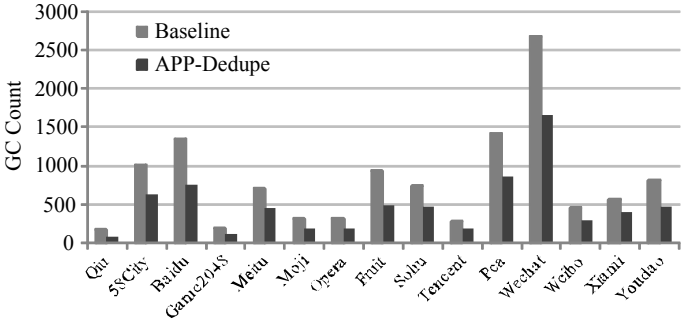

Fig. 8. The total GC counts within the eMMC device in the evaluation driven by the 15 traces.

V. Conclusion

Performance of the storage subsystem in smartphones plays an important role in the application performance. This paper proposes APP-Dedupe to detect and eliminate the I/O redundancy. The extensive benchmark-driven and trace-driven experiments conducted on our lightweight prototype implementation of APP-Dedupe show that APP-Dedupe reduces the GC overhead by an average of 41.5%, reduces the response times by up to 15.4% and saves the storage capacity by an average of 45.2%.

It is worth noting that our application of deduplication to the smartphone storage is still preliminary and an ongoing research topic. As such, some research issues remain

to be addressed as our future work. First, the issue of power consumption is critically important in smartphones [11], [27]. Deduplication is effective in reducing the I/O traffic and GC activities and thus has a potential to improve the storage energy efficiency. On the other hand, the hash computing consumes extra processing power. As a result, another important objective of APP-Dedupe is to improve the energy efficiency, in addition to the performance improvement and capacity saving. As a direction of future work, we will investigate the power consumption issue associated with deduplication in smartphones. Second, from our experimental results, we observe that different applications have different data characteristics. Depending on such characteristics, deduplication may not always improve the performance. Thus, deduplication for the mobile applications should be dynamically enabled or disabled. We will investigate how to dynamically apply deduplication on the smartphone storage at runtime to improve the flexibility of APP-Dedupe.

VI. Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 61472336 and No. 61402385, the US NSF under Grant No. NSF-CNS-1116606 and NSF-CNS-1016609. This work is also supported by Key Laboratory of Information Storage System, Ministry of Education of China and sponsored by Huawei Innovation Research Program.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (ATC'08)*, Jun. 2008.

[2] A1 SD Bench SD Card Benchmarking App. http://a1dev.com/sd-bench/.

[3] F. Chen, T. Luo, and X. Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, Feb. 2011.

[4] J. Courville and F. Chen. Understanding Storage I/O Behaviors of Mobile Applications. In *Proceedings of the IEEE 32nd Symposium on Mass Storage Systems and Technologies (MSST'16)*, May. 2016.

[5] Why do Samsung phones slow down after time of usage? https://www.quora.com/why-do-samsung-phones-slow-down-after-some-time-of-usage.

[6] F. Douglis, A. Duggal, P. Shilane, T. Wong, S. Yan, and F. Botelho. The Logic of Physical Garbage Collection in Deduplicating Storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Feb. 2017.

[7] A. El-Shimi, R. Kalach, A. Kumar, A. Oltean, J. Li, and S. Sengupta. Primary Data Deduplication - Large Scale Study and System Design. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*, Jun. 2012.

[8] Y. Fu, H. Jiang, N. Xiao, L. Tian, and F. Liu. AA-Dedupe: An Application-Aware Source Deduplication Approach for Cloud Backup Services the Personal Computing Environment. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (Cluster'11)*, Sep. 2011.

[9] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, I. Toledo, and A. Zuck. SDGen: Mimicking Datasets for Content Generation in Storage Benchmarks. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Feb. 2015.

[10] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, Feb. 2011.

[11] J. Huang, A. Badam, R. Chandra, and E. Nightingale. WearDrive: Fast and Energy-Efficient Storage for Wearables. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, Jun. 2015.

[12] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13)*, Jun. 2013.

[13] C. Ji, L. Chang, L. Shi, C. Wu, Q. Li, and C. Xue. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*, Jun. 2016.

[14] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, Feb. 2012.

[15] H. Kim, D. Shin, Y. Jeong, and K. Kim. SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Feb. 2017.

[16] S. Kim, J. Jeong, and J. Lee. Efficient Memory Deduplication for Mobile Smart Devices. *IEEE Transactions on Consumer Electronics*, 60(2):276–284, 2014.

[17] W. Kim, B. Nam, D. Park, and Y. Won. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Feb. 2014.

[18] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, Feb. 2010.

[19] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'13)*, Feb. 2015.

[20] W. Lee, K. Lee, H. Son, W. Kim, B. Nam, and Y. Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomal. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, Jun. 2015.

[21] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace. Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Feb. 2014.

[22] H. Luo and H. Jiang. Fast Transaction Logging for Smartphones. In *Proceedings of the IEEE 32nd Symposium on Mass Storage Systems and Technologies (MSST'16)*, May. 2016.

[23] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok. Using Hints to Improve Inline Block-Layer Deduplication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, Feb. 2016.

[24] B. Mao, H. Jiang, S. Wu, and L. Tian. POD: Performance Oriented I/O Deduplication for Primary Storage Systems in the Cloud. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*, May 2014.

[25] B. Mao and S. Wu. Exploiting Request Characteristics and Internal Parallelism to Improve SSD Performance. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD'15)*, Oct. 2015.

[26] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[27] D. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang. Reducing Smartphone Application Delay through Read/Write Isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, May 2015.

[28] J. Ren, M. Liang, Y. Wu, and T. Moscibroda. Memory-Centric Data Storage for Mobile Systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, Jun. 2015.

[29] K. Shen, S. Park, and M. Zhu. Journaling of Journal is (almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Feb. 2014.

[30] P. Shilane, R. Chitloor, and U. Jonnala. 99 Deduplication Problems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*, Jun. 2016.

[31] H. Wang, P. Huang, S. He, K. Zhou, C. Li, and X. He. A Novel I/O Scheduler for SSD with Improved Performance and Lifetime. In *Proceedings of the 29th International Conference on Massive Storage Systems and Technology (MSST'13)*, May 2013.

[32] G. Wu and X. He. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys'12)*, Apr. 2012.

[33] S. Wu, X. Chen, and B. Mao. Exploiting the Data Redundancy Locality to Improve the Performance of Deduplication-based Storage Systems. In *Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS'16)*, Dec. 2016.

[34] S. Wu, Y. Lin, B. Mao, and H. Jiang. GCaR: Garbage Collection aware Cache Management with Improved Performance for Flash-based SSDs. In *Proceedings of the 30th International Conference on Supercomputing (ICS'16)*, Jun. 2016.

[35] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[36] H. Xu, Y. Zhou, C. Gao, Y. Kang, and M. Lyu. SpyAware: Investigating the Privacy Leakage Signatures in APP Execution Traces. In *26th IEEE International Symposium on Software Reliability Engineering (ISSRE'15)*, Nov. 2015.

[37] G. Yadgar, E. Yaakobi, and A. Schuster. Write once, get 50% Free: Saving SSD Erase Costs Using WOM Codes. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Feb. 2015.

[38] S. Yan, H. Li, M. Hao, H. Tong, S. Sundararaman, A. Chien, and H. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Feb. 2017.

[39] P. Zhang, P. Huang, X. He, H. Wang, L. Yan, and K. Zhou. RMD: A Resemblance and Mergence Based Approach for High Performance Deduplication. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP'16)*, Sep. 2016.

[40] S. Zhen, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok. A Long-Term User-Centric Analysis of Deduplication Patterns. In *Proceedings of the IEEE 32nd Symposium on Mass Storage Systems and Technologies (MSST'16)*, May 2016.

[41] D. Zhou, W. Pan, W. Wang, and T. Xie. I/O Characteristics of Smartphone Applications and Their Implications for eMMC Design. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC'15)*, Oct. 2015.

[42] K. Zhou, S. Hu, P. Huang, and Y. Zhao. LX-SSD: Enhancing the Lifespan of NAND Flash-based Memory via Recycling Invalid Pages. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST'17)*, May 2017.