A Small Non-Volatile Write Buffer to Reduce Storage Writes in Smartphones

Mungyu Son, Sungkwang Lee, Kyungho Kim*, Sungjoo Yoo, and Sunggu Lee
Embedded System Architecture Lab, POSTECH
Samsung Electronics*

ABSTRACT

Storage write behavior in mobile devices, e.g., smartphones, is characterized by frequent overwrites of small data. In our work, we first demonstrate a small non-volatile write buffer is effective in coalescing such overwrites to reduce storage writes. We also present how to make the best use of write buffer resource the size of which is limited by the requirement of small form factor. We present two new methods, shadow tag and SQLite-aware buffer management both of which aim at identifying hot storage data to keep in the write buffer. We also investigate the storage behavior of multiple mobile applications and show that their interference can reduce the effectiveness of write buffer. In order to resolve this problem, we propose a new dynamic buffer allocation method. We did experiments with real mobile applications running on a smartphone and a Flash memory-based storage system and obtained average 56.2% and 50.2% reduction in storage writes in single and multiple application runs, respectively.

1. INTRODUCTION

Flash memory is the de-factor storage of mobile devices such as smartphones, tablet PCs, watches, etc. Compared with the conventional hard disk drive, Flash memory has advantages such as fast random read, small form factor, shock resistance, etc. However, Flash memory suffers from write-related problems, e.g., poor endurance, long latency and high power consumption. In order to address these problems, write traffics to Flash memory need to be minimized.

Our work is based on two key observations. First, mobile applications tend to generate frequent overwrites to the storage. The frequent overwrites are incurred by the characteristics of mobile applications, i.e., user interactivity and small writes. Second, mobile applications often utilize SQLite [1] to store data in the storage. SQLite incurs additional writes to mobile storage due to journaling.

In our work, we investigate the characteristics of mobile storage behavior and present new methods to reduce storage[1] writes. We aim at exploiting a small non-volatile write buffer which, in our work, consists of small (64KB) SRAM buffer with capacitor support. We achieve non-volatility, in case of sudden power failure, by writing the contents of SRAM buffer to the Flash memory with the capacitor as the power source. Especially, due to the requirements of low cost and small form factor in mobile devices, the non-volatile write buffer needs to be as small as possible (see Section 6.2 for details). In our work, we propose new management methods which make the best use of the small non-volatile write buffer.

Our contribution is as follows.

- We first demonstrate that a small non-volatile write buffer is effective in reducing storage writes with real smartphone usages.
- In order to achieve further reductions in storage writes, we present new methods of managing the small non-volatile write buffer, i.e., shadow tag to keep write history information at low cost and SQLite-aware write buffer management.
- We also present a new method of storage write reduction for multiple application runs where inter-application interference can reduce the effectiveness of write buffer thereby increasing storage writes.
- We present, as a design framework, a storage trace generation tool where SQLite traces are extracted from mobile devices. We also present a real implementation of our proposed methods on OpenSSD [2].

This paper is organized as follows. We review related work in Section 2. We give preliminaries in Section 3. We explain our motivation in Section 4. We describe the proposed method in Section 5. We report our experiments in Section 6. We conclude the paper in Section 7.

2. RELATED WORK

There have been many studies on reducing Flash memory writes. In this section, we review those related with write buffer and SQLite. Park, et al. [3] propose clean first LRU (CFLRU) to reduce write traffics to Flash memory-based storage by favoring the eviction of clean data from the main memory. Jo, et al. [4] present a method that finds a Flash block which has the most dirty pages in the main memory and evict those dirty pages to the storage in order to exploit the high performance of sequential storage writes. Kang, et al. [5] present a non-volatile write buffer consisting of capacitor-backed DRAM in the solid-state disk (SSD). This work is similar to ours in that non-volatility is realized by using capacitors. However, the size of write buffer in [5] is large (in 512MB) and there is no optimization method to better utilize the write buffer to achieve further reduction in storage writes.

[1] We refer to storage as Flash memory-based storage unless otherwise stated.

In mobile devices, due to the limited resource for write buffer imposed by the requirement of low cost and small form factor, we need optimizations as ours to make the best use of the small non-volatile write buffer.

Non-volatile write buffers or journal area can be realized by adopting fast non-volatile memory, e.g., phase-change memory (PCM). Kim, et al. [6] present PCM-based rollback journal to exploit the better write endurance of PCM than Flash memory. Kim, et al. [7] present, for mobile devices, a delta journaling based on PCM. Assuming write-ahead logging (WAL) as the journaling method, for a commit operation which writes data to the journal area, the difference between old and new data called delta is calculated. If the delta can be compressed, the compressed delta is stored in the PCM journal. Otherwise, the uncompressed data are stored in the journal area on the NAND Flash memory.

Compared with [6] and [7], there are three differences in our work. First, ours aims at reducing all the Flash memory write traffics (covering data writes as well as journal writes) while they reduce only journal writes (usually, half of write traffics). Second, they are based on emerging new memory, PCM which is still expensive, while ours is based on capacitor-backed SRAM. Third, most importantly, our work shows that a very small write buffer (of only 64KB) is effective in reducing writes to the storage of mobile device while a much larger buffer area (e.g., 16MB PCM in [7]) is assumed in the previous works.

Recently, there have been several works on reducing SQLite writes by tackling a problem called journaling of journal (JoJ) [8]. Ours is orthogonal to those JoJ solutions since they aim at reducing duplicated journal writes while ours aims at reducing any Flash writes (possibly after the JoJ solutions are applied).

3. PRELIMINARY: SQLITE IN MOBILE DEVICES

Mobile applications store data in the storage in two ways, SQLite database (DB) file and normal file. For instance, Facebook stores user data, e.g., news feed, user profile, etc. in an SQLite DB file and multimedia data in normal files. Typically, user data are hot write data while multimedia data are read-oriented. In [9], Lee, et al. report that SQLite generates average 62% (up to 95%) of write traffics in smartphone storage.

SQLite is a relational database management system (RDBMS) implemented in a library. It manages data with a B+ tree structure in a file. Each node of tree can hold one or more records called cell. The size of node is fixed to multiples of sector size, specifically, 512B to 64KB. SQLite calls this fixed-sized node a *page*. In SQLite, storage access, including journaling, is done at the granularity of page.

On each write transaction to the database, SQLite performs journaling to ensure consistency [10]. SQLite provides rollback journal (OFF, DELETE, TRUNCATE, and PERSIST) and write-ahead logging (WAL) journal. In our experiments, we implement our methods with the rollback journal, to be exact, DELETE mode. Ours can also be applied to WAL journal. Journaling is to keep a backup copy of old or new data in a journal area on the storage and utilize it in case of system crash. For each write to the database, first, the old (for rollback journal) or new (WAL journal) data are written to the journal, which is called *commit*. Then, at the end of a transaction (e.g., a set of consecutive SQLite commands), the DB page is consistently updated with the new contents, which is called *checkpoint*.[2] Note that the journal contents is discarded after the storage is checkpointed.

The journal is allocated a portion of mobile storage and a write to the journal, during the commit step described above, incurs a Flash memory write, e.g., 8KB write in case of 8KB Flash page. The journal consists of header and data pages. On each write to the journal area during the commit, the journal header needs to be updated. In case of system crash, e.g., due to sudden power failure, during the reboot step, the journal header is first checked to see if there is any valid backup copy in the journal. If there is any valid one, in order to make the storage state consistent, the backup copy is applied to the storage, e.g., a possibly inconsistent storage state returns to the consistent state of old data in case of rollback journal.

4. MOTIVATION

Figure 1 shows the statistics of storage writes to journal header and data/journal pages for four representative mobile applications. X-axis represents the number of overwrites to a Flash page and Y-axis the number of corresponding Flash pages. For instance, as the arrow indicates in Figure 1 (a), our Facebook use case has 12 journal header pages which are written five times. Figure 1 shows that many logical pages are overwritten during mobile application runs and it is common behavior across the representative applications. Note that journal header pages tend to have more overwrites than normal data and journal (data) pages.

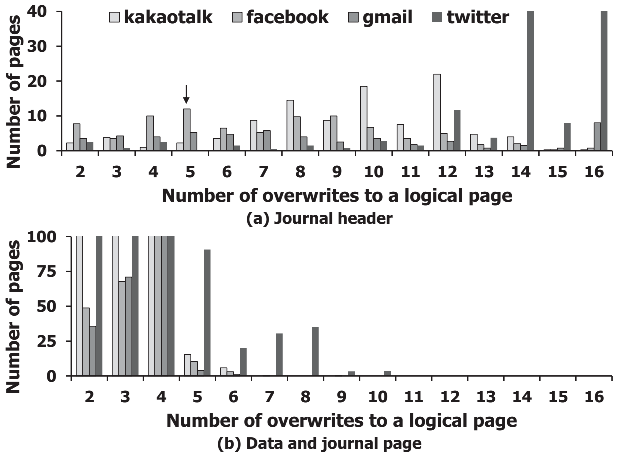

Figure 1 Characteristics of storage overwrites.

[2] In case of rollback journal, the new data are written to the DB page. Thus, the checkpoint is to verify the completion of new data write and to discard the journal contents, i.e., old data from the journal. In case of WAL journal, the journal has the new data. Thus, the checkpoint is to copy the journal contents to the DB page and discard the journal contents.

We investigated the write behavior of mobile applications. Our findings are as follows. Mobile applications frequently write a small amount of data using SQLite. SQLite manages the data with a B+ tree structure consisting of database pages as mentioned in Section 3. Figure 2 illustrates an INSERT operation to a database page (a leaf node of the B+ tree) for a messenger application called Kakaotalk. When a new message is entered (left in the figure), it is inserted into an existing database page (right in the figure) as far as the page has available free space. In this case, when entering a short message (in bold), only tens of bytes are newly inserted into the database page.

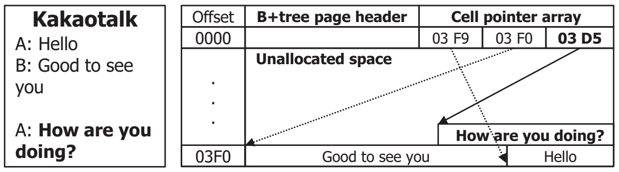

Figure 2 INSERT operation example.

Most of mobile applications have such small writes in common. Figure 3 shows data difference per page write. For instance, in case of Facebook, 69.6% of overwritten pages have maximum 64 bytes of difference between two consecutive writes.

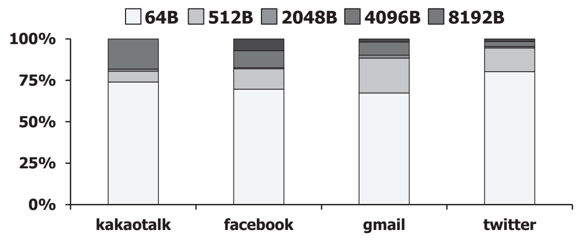

Figure 3 Data difference statistics in storage writes.

Note that each write to a database page incurs multiple writes to the journal area, that is, journal header and journal data pages as well as the database page. To be specific, for each database command such as INSERT and UPDATE, a commit incurs writes to the journal data and header files. Then, during the checkpointing, the database file is written in case of WAL journal[3], and the journal file is invalidated and the journal header is again updated, i.e., written. Thus, the journal header page tends to receive more (2~4 times more writes per page on average as shown in Figure 1) writes than the journal data and database pages.

In summary, the SQLite database pages tend to be overwritten, which motivates us to utilize the write buffer. Especially, the journal header pages are hot pages. Thus, given a small write buffer, our strategy is to identify as many hot pages as possible at a low cost and, especially, exploit the information of journal header page to keep journal header pages in the write buffer as long as possible.

[3] In case of rollback journal, no action is taken here since the database file was already written with the new data.

5. WRITE BUFFER FOR MOBILE STORAGE

In this section, we describe the management of small write buffer to minimize storage writes. The issue of realizing non-volatility will be discussed in Section 6.

5.1 Small Write Buffer Organization

Figure 4 shows the organization of the small non-volatile write buffer. It is located in the controller of Flash storage, e.g., the controller of eMMC storage device. It consists of three parts: write data buffer, journal header buffer, and shadow tag. The write data buffer contains recently written data and the associated addresses. In our experiments, we use 8-entry write data buffer. Thus, it stores the capacity of 8 Flash pages (each 8KB), total 64KB. We set the configuration of our non-volatile write buffer through an extensive sensitivity analysis (refer to Section 6). Note that only the write data buffer stores write data while the other journal header buffer and shadow tag keep only address information.

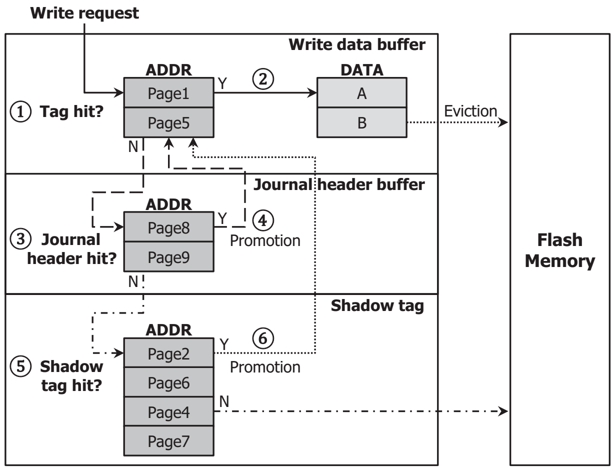

Figure 4 Small non-volatile write buffer organization.

The journal header buffer is to exploit the write behavior on journal header described in Section 4. It contains the addresses of journal header page (8KB page) provided by the SQLite library code running on the mobile application processor of the smartphone. The information of journal header page can be transferred from the library code to the mobile storage via TRIM command [11]. The shadow tag contains past write history. It consists of addresses of recently written Flash pages (the data of which are not stored in the write data buffer). When a Flash write hits the shadow tag, the write is considered to be hot.

Assume that a write request arrives at the controller (step 1 in Figure 4). First, the addresses in the write data buffer are compared with the new one. If there is a match, i.e., a hit, then the data are written to the data part (step 2). The write data buffer has a replacement policy. We use least recently used (LRU) policy after a sensitivity analysis with other candidates, LFU, RRIP, NRU [12][13][14]. Thus, the hit entry is promoted to the most recently used (MRU), i.e., the highest priority position in the priority stack. If there is a miss, then the journal header buffer is looked up (step 3). If there is a hit in the journal header buffer, the write data buffer is revisited in order to insert the journal header data to the write data buffer (step 4). When inserting a new data entry to the write data buffer, a victim entry is selected by a replacement policy. Note that, even in case of hit, the journal header buffer continues to keep the associated address of journal header while the corresponding data and address are written to the write data buffer. The address of journal header buffer is evicted only when a new address of journal header is inserted to the journal header buffer (by the hint of SQLite library). We also apply LRU replacement policy to the journal header buffer (32 entries in our experiments).

If the write gives a miss to the journal header buffer, then the shadow tag is consulted for a hit (step 5). In case of hit, the hit entry is promoted to the write data buffer (step 6). To be exact, the corresponding entry is removed from the shadow tag while the associated address and data are written to the write data buffer, if needed, evicting a victim. In case of a miss to the shadow tag, the new write address is inserted into the shadow tag evicting a victim under LRU replacement policy. In this case, the write data are written to the Flash memory.

5.2 Write Buffer Management for Multiple Apps

Recent smartphones start to support multiple application (App) runs. In such a case, inter-App interference can degrade the effectiveness of write buffer thereby increasing storage writes. Figure 5 illustrates the negative impact of such an interference. As the metric of interference, we use a ratio, $R_{m/s}$ which is the ratio of the amount of storage writes in multiple application runs to the sum of storage writes that each application has in a single application run. Thus, if $R_{m/s}$ is larger than 1, the multiple application run produces more storage writes than single application runs. Thus, the write buffer is less effective in the multiple application run. For instance, as shown in Figure 5, the multiple application run of Kakaotalk and Twitter gives 1.6 times storage writes compared with the sum of storage writes of their solo runs.

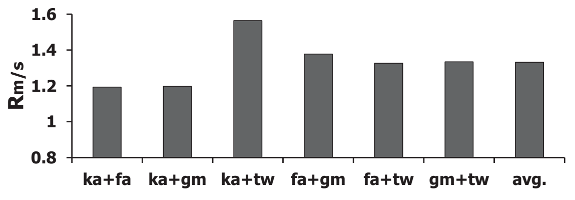

Figure 5 Inter-application interference.

In the experiments of Figure 5, both applications share the write data buffer without considering their storage write behavior. If the write data buffer resource is allocated to each application in proportion to their demand, the write data buffer can be more efficiently utilized thereby reducing total storage writes. In order to address this problem, we propose a dynamic write buffer allocation which measures the demand of each application and judiciously allocates buffer resource to each application. The proposed method adjusts the insertion point (in the priority stack) to the write data buffer on an application basis. As explained in Section 5.1, the write data buffer basically adopts LRU replacement policy where a newly inserted entry is given the highest priority, i.e., MRU position. This policy is effective for single-application cases. For multiple application cases, we propose a new policy called dynamic insertion point (DIP).

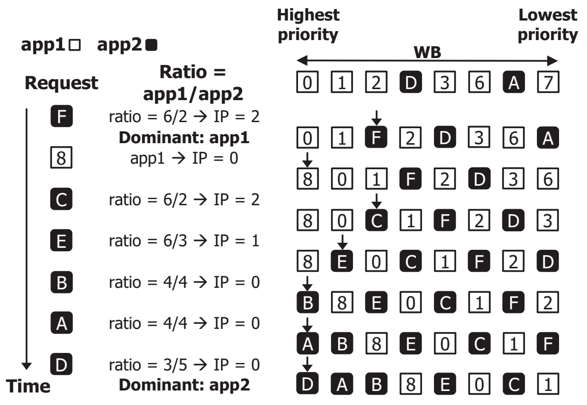

Figure 6 An example operation of DIP for two applications.

Figure 6 illustrates a simple example for 8-entry write data buffer shared between two applications with the DIP policy (in short, DIP). In this figure, we assume a dual-application run, applications 1 and 2. DIP runs as follows. First, periodically, it checks the per-App occupation of write data buffer and shadow tag. Then, we adjust the insertion point of dominant App to the highest priority position while that of less dominant App to a lower priority position. In Figure 6, when application 2 makes a write request to page F, application 1 is occupying more resource of write data buffer than application 2. Thus, application 2, a less dominant one, has an insertion point (IP) at a lower priority position (see details later in this section). In this case, IP is assumed to be calculated to be 2 (the 3rd highest position). As shown in the figure, when the dominant App, application 1 makes a write request to page 8, the data are inserted into the highest priority position. Thus, the write data of dominant App, application 1 can stay in the write data buffer longer than that of application 2, which will enable us to achieve more write coalescing from the application having more write data than the other application. Note that, due to the characteristics of frequent overwrites as shown in Figure 1, the dominant application tends to give more reductions in storage writes.

In order to determine the lower priority (LP) position for the less dominant application, first, we use a minimum level called minimum insertion point (mIP) which is set to the half position in the priority stack after a sensitivity analysis (see Section 6). mIP is required to avoid premature eviction for the data of the less dominant application. In addition, the LP position needs to be a function of the relative amount of write data of dominant application to that of less dominant one. Figure 7 shows the function used to determine the LP position in our experiments. We obtain this through an extensive sensitivity analysis. X-axis represents the ratio of the amount of write data of dominant application to that of less dominant one. The figure shows that as the ratio increases, i.e., the dominant application issues more write data than the less dominant one, the LP position moves to the minimum insertion point, mIP (=3 since the size of write data buffer is

8). Recall the example in Figure 6. As application 2 writes more data, the ratio (=app1/app2) decreases thereby increasing the insertion point of application 2. In our implementation, we use the number of entries in the write data buffer and shadow tag as the estimate of the amount of write data.

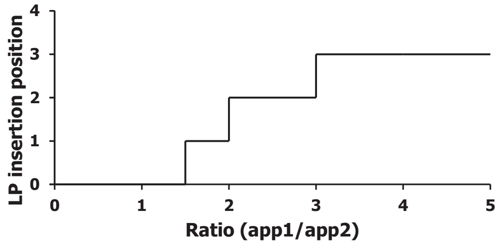

Figure 7 The LP insertion position.

6. EXPERIMENTS

6.1 Experimental Setup

Mobile applications are interactive ones. Thus, for repeatable experiments, we needed to obtain storage write traces generated by real mobile applications running on a smartphone. To do that, we developed a storage trace generation tool where SQLite traces are extracted from mobile devices. To be specific, we modified the source code of SQLite, which is publicly available, in order to insert trace generation functions. The tool runs as follows. When an application starts to run SQLite library, the tool stores the initial copy of target database file as a part of trace. On each SQLite command execution to the database file, our tool generates an entry in a command trace file. In addition, it stores, in a separate data trace file, write data bound for the database file. After running an application, we obtain three files, the initial copy of database file, the SQLite command trace file and the data write trace file.

We used four representative mobile applications, Facebook, Gmail, Twitter, and Kakaotalk. For each application, we prepared eight (four long and four short) traces using our trace generation tool. For instance, in the case of Kakaotalk, we obtained the traces while users perform 1:1 chat with emoticons, pictures, video, long news, game invitations, and group chats. The long (short) traces range between 2000 and 38000 (200 and 400) page writes. For the experiments of multiple application runs, we focused on dual application cases since the current smartphones mostly run only two active applications.

We implemented our methods on a real Flash storage prototype, OpenSSD [2]. To be specific, we designed a software implementation of our methods (~1000 lines in C) which runs on the controller of OpenSSD. In order to allocate the resource of write data buffer (64KB), we borrowed a portion of SATA write buffer. We also developed a trace player running on the host (laptop in our experiments). The trace player takes as input the three trace files, stores the initial copy of database file on the OpenSSD, and replays storage writes respecting the SQLite command and write data in the trace files. We implemented a profiling function on the OpenSSD which gathers statistics, e.g., number of writes to write data buffer, hit ratio of write buffer, the amount of data difference between consecutive writes to a page, etc.

6.2 Non-Volatility in Write Buffer

There are two possibilities in the implementation of non-volatile write buffer: host and storage side implementations. We expect a host side implementation can be economically difficult in reality since the smartphone itself needs to be equipped with capacitors for the storage. We expect a storage side implementation can be more practical. Table 1 shows the parameters used in our calculation of capacitance required to write 64KB data from SRAM to Flash memory. Assuming a capacitor implementation inside of eMMC package, the capacitor having the required capacitance, 411μF can occupy only 4% of the package volume [15].

Table 1 Parameters used in calculation of required capacitance.

Parameter	Value
SRAM buffer size	64 KB
Page program time [16]	220 μs
Program current [16]	25~35 mA
Supply voltage	2.7~3.3 V
Energy	738 μJ
Required capacitance for NV-write buffer	411 μF

6.3 Evaluation of Storage Write Reductions

In the single application case, we compare the baseline (where no write buffer is used) with three solutions of ours: WB (write data buffer only), WB+SH (=WB + shadow tag), and WB+SH+JH (=WB+SH + journal header buffer). We use a configuration of 8-entry write data buffer, 32-entry shadow tag, and 32-entry journal header buffer which is obtained through extensive sensitivity analyses.

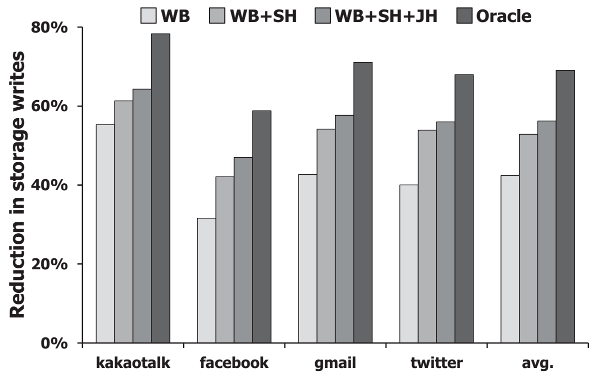

Figure 8 Reduction in storage writes.

Figure 8 shows the reduction in storage writes with respect to the baseline. Each bar represents the average of storage write reductions obtained from four long traces. WB, WB+SH, and WB+SH+JH give average 42.4%, 52.9%, and 56.2% reductions, respectively. Oracle represents the ideal case where the write buffer can give maximum reductions in storage writes by exploiting future write behavior. Large reductions in storage writes result from the write behavior of mobile applications that data and journal pages tend to have overwrites and journal header pages have more significant overwrites as explained in Section 3.

Figure 9 compares average storage write reductions between short and long traces. As the figure shows, our proposed methods give similar reductions in storage writes across long and short traces.

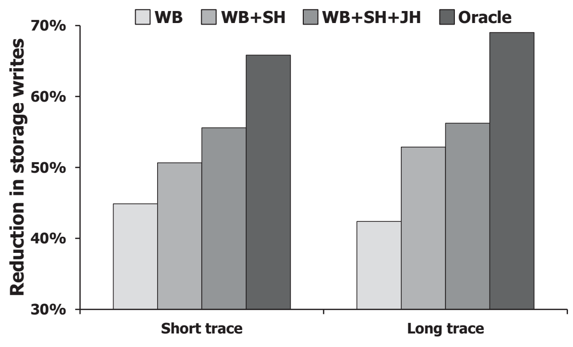

Figure 9 Effects of trace size on storage write reduction.

Figure 10 shows the reductions in storage writes in multiple application runs. In this case, we compare the baseline (without the write buffer), existing buffer management methods, and our dynamic insertion point (DIP) method (with 8-entry write data buffer and 32-entry shadow tag). In the figure, HWB represents a method of dividing in half the write data buffer between two applications. HI is another way to equally allocate write data buffer between two applications by setting the insertion point of each application to the half position of priority stack. DPIPP is the dynamic cache partitioning method used in multi-core caches [17][18]. LRU is our solution used for single application case. As the figure shows, our method, DIP gives 6.2%, 3%, 7.7%, and 2% more reductions in storage writes than HWB, HI, DPIPP and LRU, respectively. By applying the journal header buffer, ours gives 4.6% more reduction than the best of existing method, LRU. In Figure 10, DPIPP gives inferior results to the others. It is because DPIPP fails to keep hot data for long enough time in the small write data buffer.

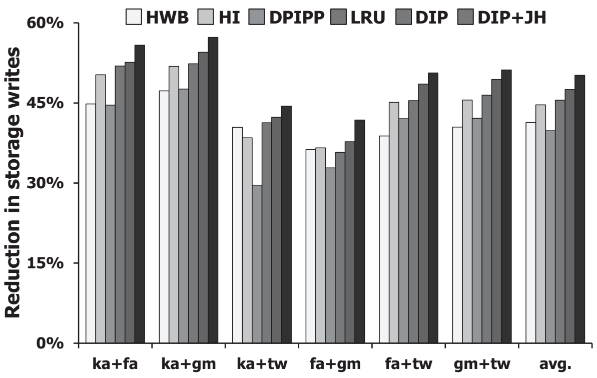

Figure 10 Cases of multiple application run.

As mentioned previously, we obtained the configuration of 8-page write data buffer, 32-entry shadow tag, 32-entry journal header buffer, and LRU policy (for single application cases) via extensive sensitivity analyses where we varied the size of write data buffer (1~12 pages), shadow tag size (2~32 entries), minimum insertion point (8 cases, application 1's IP is set to 0 and application 2's IP ranges 0 to 7), and promotion policy from shadow tag to write data buffer (1, 2, and 3 hits). Replacement policy LRU gives 8.5%, 1.2%, and 0.7% more reduction in storage writes than LFU, NRU, and RRIP, respectively. The area overhead of our proposed method is small. It costs 65KB (64KB write data buffer and 1KB for 32-entry shadow tag and 32-entry journal header buffer).

7. CONCLUSION

In this paper, we first show that smartphones tend to have frequent overwrites to the storage due to the mobile usage behavior, e.g., short messages, and the database structure managed by SQLite. We demonstrate that, due to the frequent overwrites, a small non-volatile write buffer is effective in reducing storage writes in smartphones. For further reductions in storage writes, we present a new method to identify hot data, shadow tag and journal header buffer. For storage write reduction for multiple application runs, we present a novel method called dynamic insertion point. Experiments with real mobile applications and OpenSSD system give average 56.2% and 50.2% reduction in storage writes in single and multiple application runs, respectively.

8. REFERENCES

[1] Data storage, storage options, http://developer.android.com/guide/topics/data/index.html.
[2] OpenSSD Project. http://goo.gl/J0Ts5, 2011.
[3] S. Park, et al., "CFLRU: A Replacement Algorithm for Flash Memory," Proc. CASES, pp 234-241, 2006.
[4] H. Jo, et al., "FAB: Flash-Aware Buffer Management Policy for Portable Media Players," IEEE Trans. Consumer Electronics, vol. 52, no. 2, pp. 485-493, 2006.
[5] W. Kang, and S. Lee, "Durable Write Cache in Flash Memory SSD for Relational and NoSQL databases," Proc. SIGMOD, pp 529-540, 2014.
[6] D. Kim, et al., "Improving the storage performance of smartphones through journaling in non-volatile memory," IEEE Trans. Consumer Electronics, vol. 59, no. 3, pp 556-561, 2013.
[7] J. Kim, C. Min, and Y. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," IEEE Trans. Consumer Electronics, vol. 60, no. 2, pp 217-224, 2014.
[8] W. Kim, and B. Nam "Resolving journaling of journal anomaly in android I/O: multi-version B-tree with lazy split," Proc. FAST, pp 273–285, 2014.
[9] K. Lee, and Y. Won, "Smart layers and dumb result: IO characterization of an android-based smartphone," Proc. EMSOFT, pp 23-32, 2012.
[10] Atomic commit, http://www.sqlite.org/atomiccommit.html.
[11] F. Shu, "Data Set Management Commands Proposal for ATA8-ACS2," T13 Technical Committee, 2007.
[12] A. Dan, and D. Towsley, "An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes," Proc. SIGMETRICS, pp 143-152, 1990.
[13] D. Lee, et al., "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," IEEE Trans. Computers, vol. 50, no. 12, pp. 1352–1360, 2001.
[14] A. Jaleel, K. Theobald, and S. Steely, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," Proc. ISCA, pp 60-71. 2010.
[15] Sandisk iNAND eMMC 4.41 specification, JESD84-A441, Datasheet.
[16] Micron Flash memory chip SLC specification, MT29F4G08AAA, Datasheet.
[17] Y. Xie, and G. Loh, "PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches," Proc. ISCA, pp 174-183, 2009.
[18] M. Qureshi, and Y. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," Proc. MICRO, pp 423-432, 2006.