Received October 13, 2021, accepted October 29, 2021, date of publication November 8, 2021, date of current version November 12, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3125779

Characterizing File Accesses in Android Applications and Caching Implications

SOOJUNG LIM AND HYOKYUNG BAHN, (Member, IEEE)
Department of Computer Engineering, Ewha University, Seoul 120-750, South Korea

Corresponding author: Hyokyung Bahn (bahn@ewha.ac.kr)

This work was supported in part by the ICT Research and Development Program of MSIP/IITP through the Developing System Software Technologies for Emerging New Memory That Adaptively Learn Workload Characteristics under Grant 2019-0-00074, and in part by the Development of Data Improvement and Dataset Correction Technology Based on Data Quality Assessment under Grant 2020-0-00121.

ABSTRACT In this paper, we explore Android applications' file access characteristics, and find out that smartphone file accesses are different from traditional desktop applications in terms of the following aspects. 1) There exist a limited number of hot blocks, which are accessed consistently during the entire execution of an application. 2) Block accesses in Android are highly biased such that the top 20% blocks account for 80% of total accesses. 3) Hot blocks of the top 100 rankings are mostly involved in SQLite. 4) Unlike desktop applications, file accesses in Android applications are write-intensive. 5) In predicting future file accesses in Android applications, frequency is a better estimator than temporal locality. 6) The effect of traditional buffer cache is limited in Android as file I/O in Android has a lot of synchronous writes, which incurs immediate storage flushing. Based on these analyses, this paper presents the implication of buffer cache management in Android. Specifically, we add a small non-volatile write buffer and present how this write buffer can be managed efficiently. Experimental results show that the proposed scheme improves the storage write traffic by an average of 21.7% and a maximum of 48.1% compared to the conventional buffer cache system.

INDEX TERMS Android, file access, application, smartphone, buffer cache, non-volatile memory.

I. INTRODUCTION

Due to recent advances in mobile platform technologies and the explosion of applications based on it, smartphones have become an essential computing device in our daily life [1]–[3]. People are increasingly working with their smartphones, and various types of applications including social media, location-based services, and multimedia streaming services emerge every day [3]–[5]. Currently, the hardware specification of smartphones has reached a certain level close to that of desktops [2]. For example, Google Pixel 5 consists of 2.4 GHz Octa-core CPU, 8GB DRAM, and 128GB Flash storage, which is sufficient to concurrently execute traditional desktop applications [6]. Such trends are expected to continue as desktop applications tend to extend their execution platform to smartphones.

Studies on smartphone performances reported that the storage subsystem is the performance bottleneck of smartphones rather than processor cores or wireless networks [7]. In order to improve the storage performances, buffer cache is widely used in various computing environments. Buffer cache stores requested file blocks read from storage in a certain part of memory area, which allows fast accesses of the same file blocks in subsequent requests [8]–[10].

As traditional buffer cache uses volatile memory (i.e., DRAM), modified file blocks should be flushed to storage immediately to prevent inconsistency or loss of data upon system crashes. To do so, file systems usually perform journaling or flush operations that reflect the modifications to storage periodically [11], [12]. In case of Android, applications also perform journaling or logging operations by making use of SQLite [11], [13]. Though such storage writes improve the reliability of file data, they degrade the effectiveness of buffer cache significantly due to frequent storage accesses even when the cache space is sufficient [12]. That is, read operations can benefit from the buffer cache, but even with large cache space, write operations cannot be absorbed by the buffer cache, resulting in heavy storage traffic.

Non-volatile memory technologies have caught interest as an attempt to reduce the number of storage writes in buffer cache [14], [15]. Non-volatile memory such as PRAM (phase change random access memory) and STT-MRAM

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

(spin-transfer torque magnetic random access memory) is a byte-addressable medium like DRAM but it is non-volatile and thus cached data are safe from power failure situations although they are not flushed to storage [16]–[18].

Figure 1 compares the traditional buffer cache architecture consisting of volatile memory and a new architecture by adding non-volatile memory as write cache or write buffer. Though volatile and non-volatile hybrid buffer cache has already been studied [14], [19], we show that previous solutions are not efficient for Android buffer cache because of some distinct file access characteristics in smartphone applications. Specifically, we explore the file access characteristics of Android applications, and find out that smartphone file accesses are different from desktop or server systems in terms of the following aspects.

- There exist a limited number of hot file blocks in Android applications, which are accessed consistently during the entire execution of an application.
- File block accesses in Android are highly biased such that the top 20% blocks account for 80% of total accesses. This is different from file access in desktops where the top 50-60% blocks account for 80% of all accesses.
- We investigate the identities of hot file blocks in Android applications, and find out that the top 100 blocks in the popularity ranking are mostly involved in SQLite.
- Unlike desktop applications, file accesses in Android applications are write-intensive. Specifically, write operations account for 50-90% of total file accesses although it depends on application types.
- In predicting future file accesses in Android applications, frequency is a better estimator than temporal locality.
- File accesses in Android applications contain a lot of synchronous writes, which incurs immediate storage flushing.

As most of storage writes incurred by smartphone applications cannot be buffered by traditional buffer cache, we suggest the adding of a small non-volatile write buffer and present how it can be efficiently managed. Specifically, we selectively flush modified blocks to either write buffer or storage based on the write characteristics of Android block accesses. Specifically, hot blocks are absorbed in write buffer, whereas cold blocks are directly flushed to storage. As the size of write buffer is limited, some hot blocks in the write buffer should be flushed to storage eventually when there is no free space. Selecting victim blocks in our write buffer is also performed judiciously by annotating the write count of blocks while storing them in the write buffer. We perform experiments with real Android workload traces and show that our scheme reduces the storage write traffic by 21.7% on average and up to 48.1% compared to the conventional buffer cache and by 10.8% on average and up to 18.1% compared to the same non-volatile write buffer architecture without our judicious management.

TABLE 1. Brief characteristics of the android file access traces used in this paper.

Workload	Ratio of read to write	Number of accesses		
		Read	Write	Total
Web browser	23.4:76.6	5720	18712	24432
Facebook	31.3:68.7	7151	15660	22811
Youtube	45.5:54.5	1310	1568	2878
Farmstory	19.3:80.7	427	1780	2207
NaverMap	5.3:94.7	10224	182288	192512
TicToc	19.1:80.9	1572	6672	8244
No5	34.1:65.9	22001	42490	64491
Mix	47.3:52.7	18549	20643	39192

The remainder of this paper is organized as follows. In Section II, we present the analysis of file accesses in Android applications. Section III presents how Android buffer cache can be accelerated by the proposed scheme: hybrid flush and hotness-aware eviction in the write buffer. Section IV evaluates the performance of the proposed scheme in comparison with the system that does not use it. Finally, Section V concludes this paper.

II. ANALYZING FILE ACCESSES IN ANDROID APPLICATIONS

In this section, we analyze the file access characteristics of Android applications from various aspects including temporal characteristics, access skewness, re-access estimation, and read/write characteristics. To collect file block access traces, we make use of the strace utility, which has the ability of tracing the system calls of a process [20]–[22].

We collect the file read/write access traces from seven Android applications, namely, Facebook a social network service, Farmstory a social game, Youtube an online video-streaming service, Farmstory a networked game, TicToc an instant messenger, Navermap, a map service application, No5 a puzzle game, and an Android Web browser. The duration of the trace collection period is in the range of 15-20 minutes for each application. We also collect a mixed trace while performing multiple applications consisting of Facebook, Web browser, Youtube, Navermap, and TicToc for 20 minutes. The characteristics of these traces are listed in Table 1.

Figure 2 plots the file block accesses as time progresses for the eight workload traces we collected. In the figure, the x-axis represents the logical time, which is increased by one for each block access and the y-axis shows the logical block number. We separately show the read and write accesses; the blue and red plots, respectively, represent read and write accesses. As shown in the figure, a certain number of low block numbers are accessed from the launch time of applications, and they are consistently accessed as time progresses. This implies the existence of hot blocks, which should be the main target of caching.

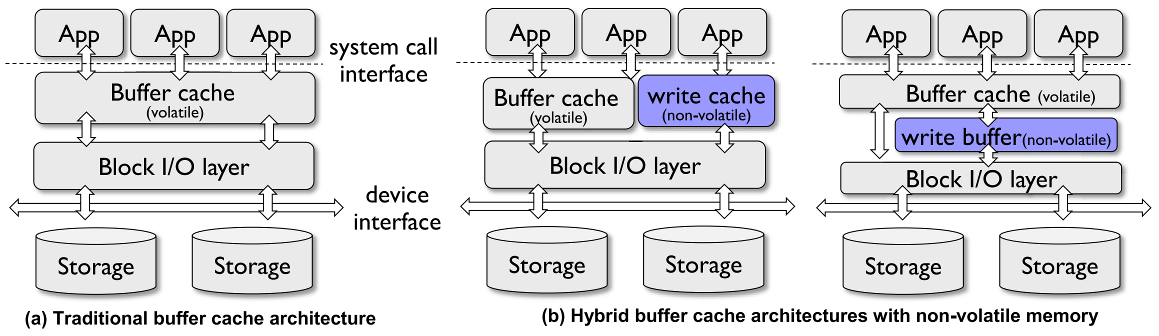
App
App
App
system call interface
Buffer cache (volatile)
Block I/O layer
device interface
Storage
Storage
(a) Traditional buffer cache architecture
write cache (non-volatile)
write buffer(non-volatile)
(b) Hybrid buffer cache architectures with non-volatile memory

FIGURE 1. Comparison of traditional buffer cache and hybrid buffer cache with non-volatile memory.

To analyze the file access characteristics of Android applications further, we investigate the identities of the hot blocks. Figure 3 shows the distributions of block accesses as the popularity rankings are varied. In the figure, the x-axis represents the ranking of blocks sorted by their total access count and the y-axis represents the number of accesses on that ranking. In the figure, we separately show the block accesses involved in SQLite and others. Note that SQLite is a lightweight database library used in most Android applications for file manipulations, which incurs a lot of synchronous write operations to storage. As shown in the figure, the top 100 blocks in the popularity ranking are mostly involved in SQLite except for Farmstory. Note that Farmstory does not make use of SQLite in file manipulations when the application was developed. Based on this observation, we can estimate that buffer cache should maintain file blocks related to SQLite for improving the cache hit ratio. However, as SQLite performs journaling, which incurs synchronous writes to storage, traditional volatile buffer cache has limitations in reducing the number of storage writes, which implies the necessity of the non-volatile buffer cache.

Figure 4 shows the cumulative ratio of block accesses for the given ratio of top blocks in popularity. Note that the x-axis is the ratio of accessed blocks sorted by their access count and the y-axis represents the ratio of accesses for the given fraction of top ranked blocks. For example, 20% in the x-axis implies the top 20% blocks, and the corresponding value in the y-axis represents the ratio of block accesses they generated.

As shown in the figure, block accesses generated by Android applications are extremely skewed. Specifically, 20% of the top ranked blocks account for about 80% of total accesses in most cases, implying that file block accesses in Android applications mostly result from some hot blocks. Unlike other applications, Farmstory does not exhibit a strong bias towards file block accesses, where 20% of the top ranked blocks account for only 40% of total block accesses. From this analysis along with the result in Figure 3, we can conclude that the source of the strong bias in Android file accesses are from SQLite. Note that Farmstory does not use SQLite as shown in Figure 3 and its cumulative block accesses in Figure 4 do not exhibit a strong bias. When compared to desktop or server systems, the skewness of Android file accesses is very strong as it is known that 50-60% of the top ranking blocks account for 80% of file accesses in server systems [23], [24].

Figure 5 shows the ratio of reads to writes for the file access traces we captured. As shown in the figure, all applications we considered exhibit write-intensive access patterns. In some applications such as Navermap, writes are over 90%. Also, as most Android applications use SQLite in file manipulations, which incurs synchronous writes to storage, such a large ratio of write operations increases storage traffic dramatically, leading to severe slowdown of smartphone systems [11], [13]. Note that this is different from traditional desktop applications, where read operations account for a large portion of file accesses [25]–[28].

To improve the file system performances, buffer cache should absorb as many I/O accesses as possible. As the size of buffer cache is limited, we need to estimate the re-access likelihood of cached blocks accurately and evicts those blocks not likely to be re-accessed from the buffer cache if there is no free cache space. Temporal locality and access frequency are the two well-known characteristics used to predict the re-access likelihood of file blocks [9], [29], [30].

We compare the two characteristics from the viewpoint of Android file accesses in Figure 6. That is, the effects of temporal locality and access frequency on the re-access likelihood of file blocks are compared. In the figure, the x-axis represents the block rankings in terms of the temporal locality and frequency, and the y-axis is the number of re-accesses that occur on the block ranking of the x-axis. The gray plot and the black plot represent the block rankings based on temporal locality and access frequency, respectively. For example, ranking 1 in the gray plot represents the re-access count of the most recently used position in the cache whereas ranking 1 in the black plot represents the re-access count of the most frequently used position.

As shown in the figure, the black plots are located above the gray plots in high rankings. This implies that the re-access likelihood of hot blocks can be estimated more accurately by access frequency than temporal locality in Android file accesses. This implies that blocks that have been accessed frequently in the past are likely to be re-accessed in the

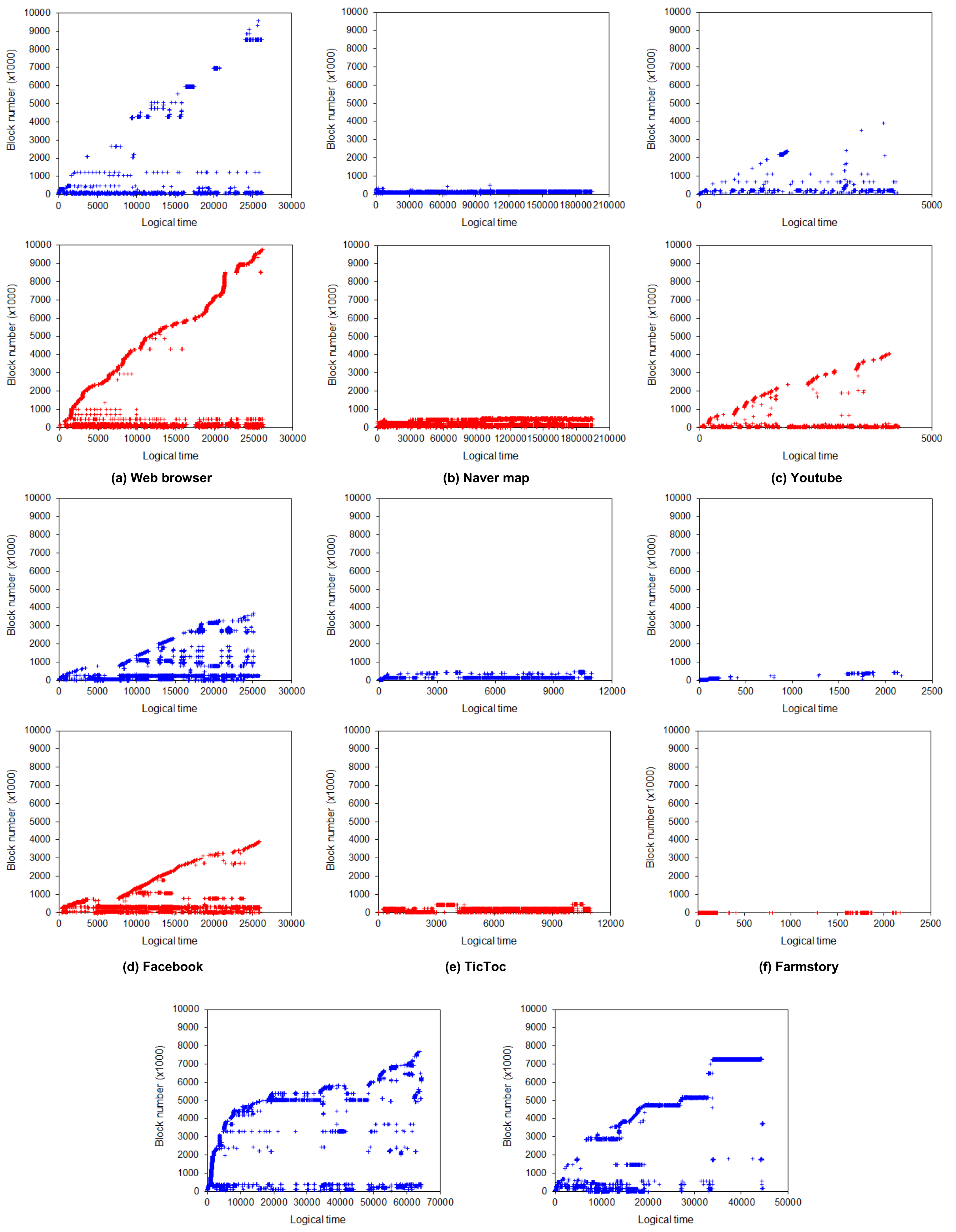

FIGURE 2. Accessed block number of Android applications as time progresses; blue plot = read; red plot = write.

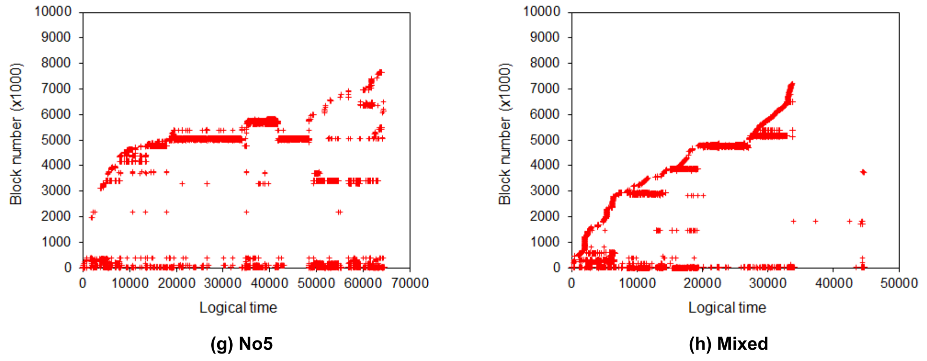

FIGURE 2. ***(Continued.)*** **Accessed block number of Android applications as time progresses; blue plot = read; red plot = write.**

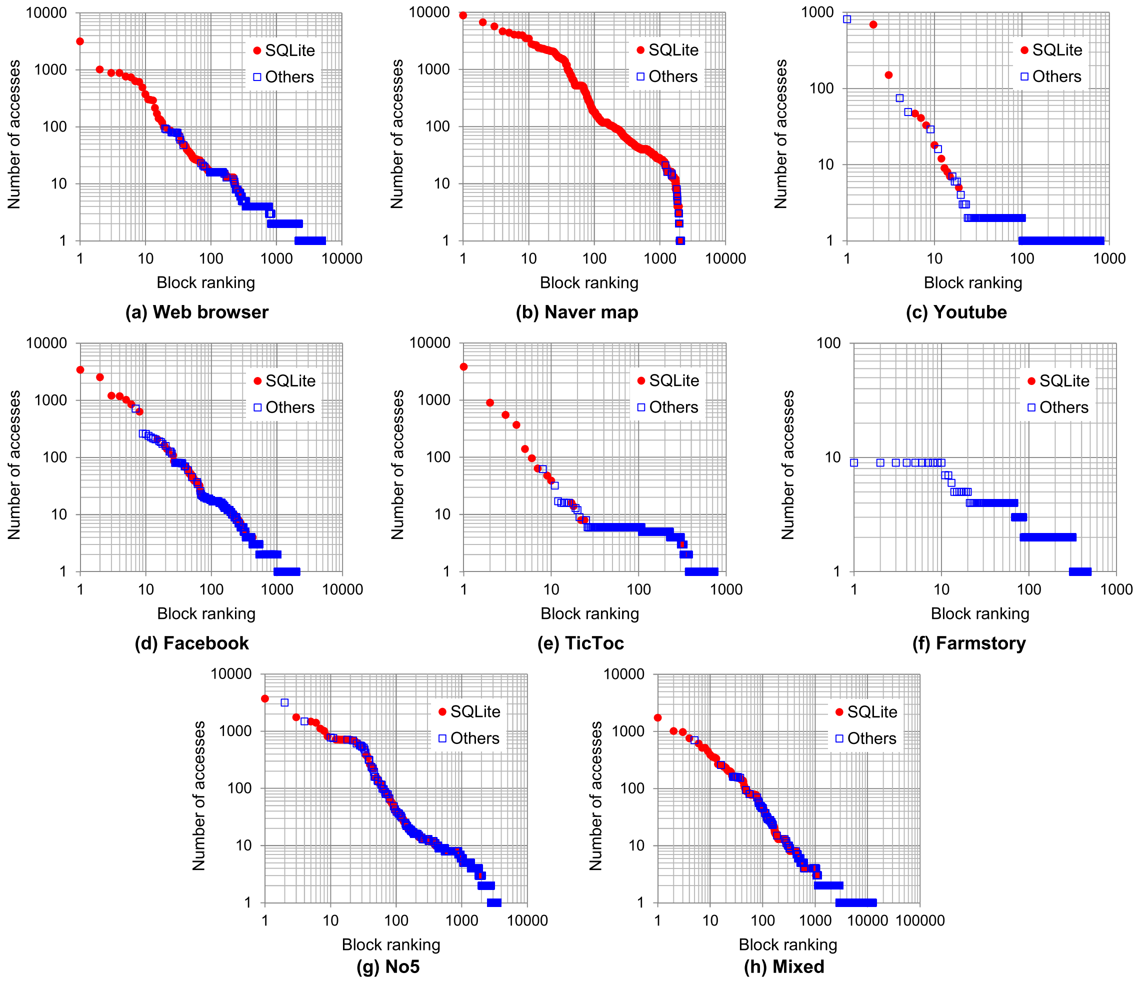

FIGURE 3. **Number of accesses that occurs on the given rankings of file blocks sorted by their access count.**

future. Thus, when eviction of blocks from the buffer cache is necessary due to the space limitation, it is better to evict less frequently accessed blocks, implying that the least frequently used (LFU) replacement algorithm will perform better than the least recently used (LRU) replacement algorithm in Android buffer cache.

However, it is also noticeable that the top ranking blocks (rankings 1 and 2) of temporal locality exhibit larger number of re-access count than those of frequency. This implies that if we aim to maximize the benefit of buffer cache, it would be necessary to maintain the most recently accessed blocks in the buffer cache although their frequency count is not large.

III. HYBRID FLUSH AND HOTNESS-AWARE EVICTION FOR ANDROID BUFFER CACHE

In this section, we present how Android buffer cache can be accelerated based on our analysis result in Section II. Buffer cache manages file data in block units, and the cached blocks

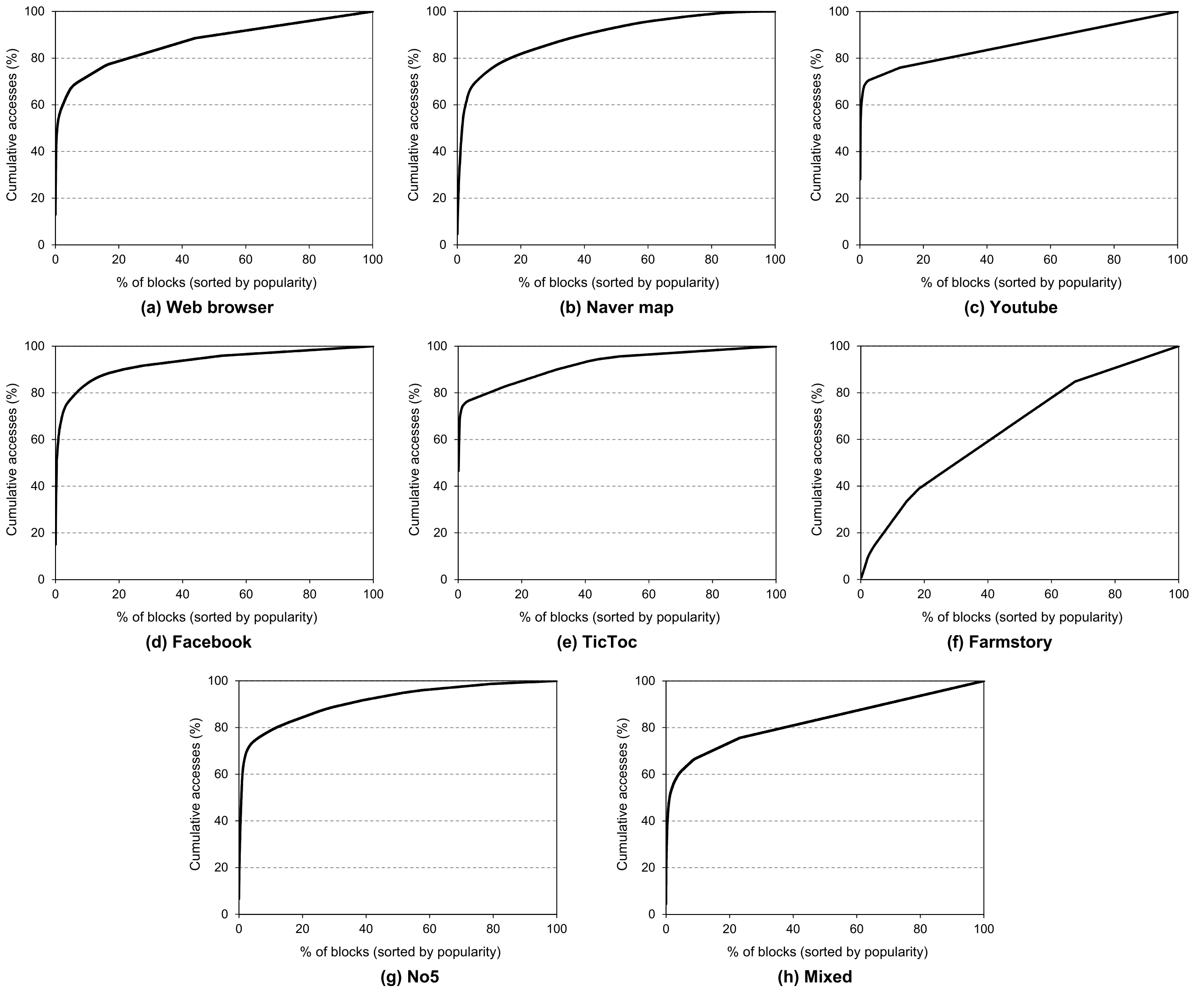

FIGURE 4. Cumulative ratio of block accesses for the given ratio of top blocks sorted by their access count.

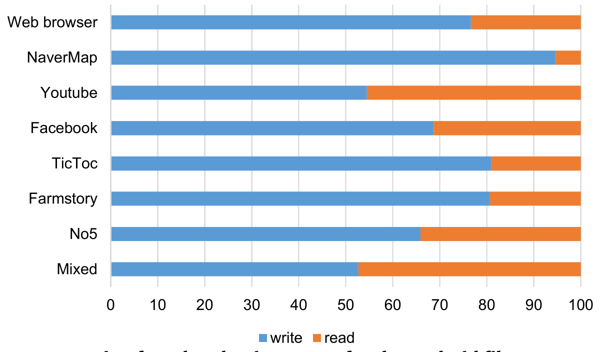

FIGURE 5. Ratio of read and write counts for the Android file access traces we captured.

can be divided into clean and dirty. A clean block means that the cached data has not been modified since it was loaded into buffer cache, and is therefore identical to the original block in storage. A cached block becomes dirty when a write operation on that block has been performed. If a clean block is selected as the victim candidate, it can simply be removed from buffer cache. On the contrary, if a dirty block is selected as the victim, it should be written to storage before removed from buffer cache. To guard against system crashes or power outages, modern file systems like Ext4 periodically flush modifications to storage without delaying the flush of dirty blocks until they are removed from buffer cache [12], [14]. For example, Ext4 flushes dirty blocks to storage every 5 seconds. In case of Android, applications usually perform synchronous writes by making use of SQLite, which also trigger the storage flushing of all dirty blocks in buffer cache. After completion of storage flushing, dirty blocks return to clean.

In Section II, we showed that a certain number of hot blocks in Android file accesses account for a large portion of storage I/O, and lots of them are involved in synchronous writes, which cannot be improved by traditional volatile buffer cache. To cope with such situations, we present two novel schemes that we call, *hybrid flush* and *hotness-aware eviction*. Figure 7 briefly shows the overall structure of the

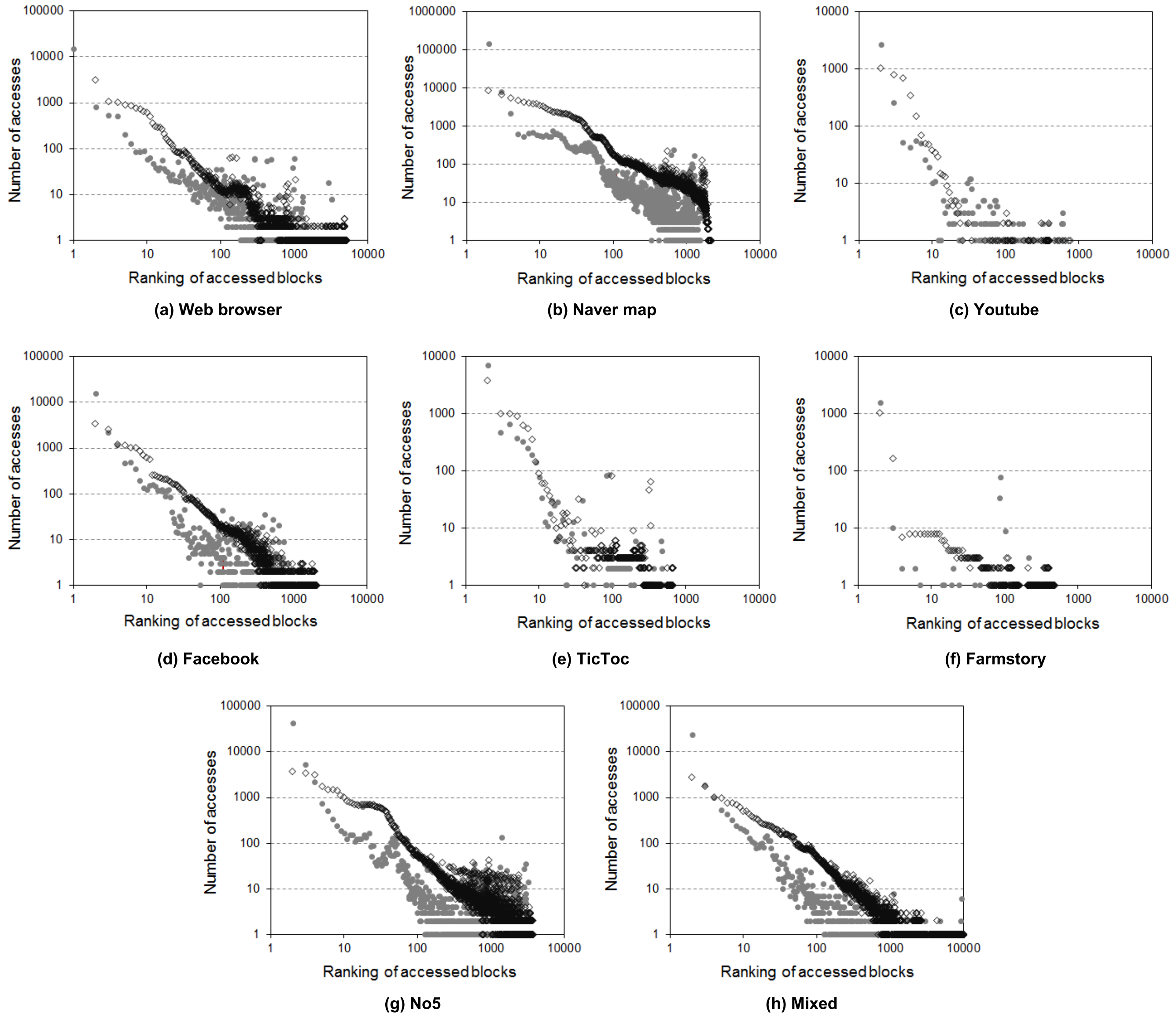

FIGURE 6. Number of accesses that occur on the ranking of accessed blocks based on temporal locality (gray) and frequency (black) of accesses.

proposed schemes. A small size (non-volatile) write buffer is located between the original (volatile) buffer cache and storage. The basic idea of *hybrid flush* is to eliminate the excessive write traffic to storage by flushing hot blocks to write buffer instead of storage. To do so, we classify blocks in the buffer cache based on their write access count. If the count is 1, we classify it as a cold block and flush it to storage. If it is more than 1, we flush the block to the write buffer. During flushing to the write buffer, the write access count of the block is also passed to the write buffer and maintained. As the write buffer we use is small, some blocks in the write buffer should be flushed to storage if there is no available space. Our *hotness-aware eviction* selects the victim block of the write buffer based on the write count of the blocks passed from the buffer cache. This is because frequency is known to estimate the re-access likelihood of Android file accesses well as analyzed in Section II.

Now, let us explain the details of our scheme with the example given in Figure 7. When a file block is requested, it is retrieved from storage and inserted into the buffer cache. For each cached block, we maintain the write count as well as the dirty bit of the block. That is, when a write operation on a file block occurs, the dirty bit of the block is set to 1 and also the write count is increased by 1. When a synchronous write or a (periodic) flush request occurs, all dirty blocks in the buffer cache are flushed to either storage or write buffer. Blocks with their write count of 1 are flushed to storage whereas those of the others are flushed to the write buffer. This is different from existing approaches adopting the non-volatile write buffer, which flush all dirty blocks to write buffer first

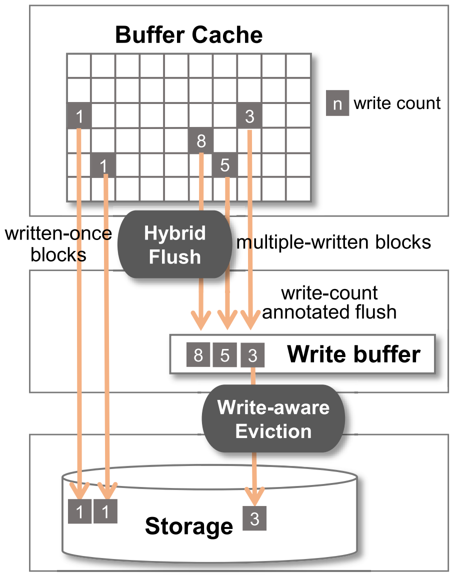

FIGURE 7. Our architecture with the two proposed schemes: hybrid flush and write-aware eviction.

instead of storage. Note that our write buffer absorbs only frequently written blocks and prohibits written-once blocks from entering the write buffer, improving the space utilization of the limited write buffer. After flushing all dirty blocks, their dirty bits are reset to 0 (i.e., changed from dirty to clean). Similar activities are also performed when a dirty block is selected as the victim block in the (volatile) buffer cache. In this case, instead of flushing all dirty blocks in the buffer cache, only the victim block is flushed.

Flushing to write buffer also passes the write access count of blocks, which will be used for selecting the eviction victim when free space in the write buffer is needed. Specifically, we select the block with the least write count as the victim block, and flush it to storage. After flushing, we remove it from the write buffer. For example, in Figure 7, among the three blocks with their write count of 8, 5, and 3 in the write buffer, our write-aware eviction selects the block with the write count of 3 as the victim block.

However, if the victim block is the most recently accessed block or the second most recently accessed block in the buffer cache, we do not evict it from the write buffer as temporal locality is stronger than frequency in case of the top rankings (rankings 1 and 2) as shown in Figure 6 of Section II.

IV. EXPERIMENTAL RESULTS

We perform trace-driven simulations to assess the effectiveness of our scheme. For a comparison purpose, we additionally simulate two architectures: Flush-storage and Flush-NVM. Flush-storage does not use non-volatile write buffer, and thus dirty blocks are directly flushed to storage. Flush-NVM uses the non-volatile write buffer similar to our scheme, but it flushes all dirty blocks to the write buffer first and does not use our write-aware eviction when an eviction is needed in the write buffer. In our simulation, the block size is set to 4KB and the flush period of Ext4 is set to 5 seconds, which is the default setting of Android and Linux.

Figure 8 shows the storage write traffic of our scheme in comparison with Flush-storage and Flush-NVM for each workload as the write buffer size is varied. As shown in the figure, our scheme reduces the storage write traffic significantly for a variety of workloads and a wide range of the write buffer size.

Specifically, the reduced write traffic of the proposed scheme is an average of 21.7% and up to 48.1% compared to Flush-storage and an average of 10.8% and up to 18.1% compared to Flush-NVM. This is mainly because our scheme reduces frequent storage writes caused by hot blocks making use of the small size write buffer and judicious software management.

Let us discuss the detailed comparison of our scheme and Flush-NVM. In case of Web browser, Navermap, Facebook, and No5, the reduced write traffic of our scheme against Flush-NVM is over 10%. This is because the popularity bias of blocks in these applications is clearly observed in Figures 3 and 4. Thus, selective flushing of hot blocks to write buffer and hotness-aware eviction are effective in these applications. In contrast, the improvement of our scheme against Flush-NVM is relatively small in Youtube, TicToc, Farmstory, and Mixed. Specifically, the improvement of our scheme against Flush-NVM is only 1% to 2% in Farmstory. This is because Farmstory does not make use of SQLite, and the popularity bias of blocks is very low as shown in Figures 3(f) and 4(f).

Now, let us compare our scheme with Flush-storage. In case of Youtube, TicToc, and No5, the reduced write traffic of our scheme against Flush-storage is over 30%. This is because most hot blocks in these applications can be absorbed by a small size write buffer. As shown in Figure 3, top 10-100 blocks account for most storage accesses in these applications. In Navermap, Facebook, Web browser, and Mix, the reduced write traffic is relatively small. Although block accesses in these workloads are skewed, blocks of rankings over 100 also account for a certain large portion of storage accesses as shown in Figure 3. Thus, a small write buffer has limitations in absorbing most of storage accesses in these applications.

To see the effectiveness of the proposed scheme further, we compare the total storage access time of Flush-NVM, Flush-storage, and the proposed scheme by simulating buffer cache, storage, and write buffer together. In our simulation, the read/write performances of NVM write buffer and flash storage are set to the maximum throughput of UFS 3.1 storage [39] and Optane^{TM} memory [40]. Figure 9 shows the storage access time of our scheme in comparison with Flush-storage and Flush-NVM for each workload as the write buffer size is varied. In the figure, we separately plot the latency caused by storage read, storage write, and NVM flush operations. Similar to the write traffic result in Figure 8, our scheme reduces the storage access time significantly for a

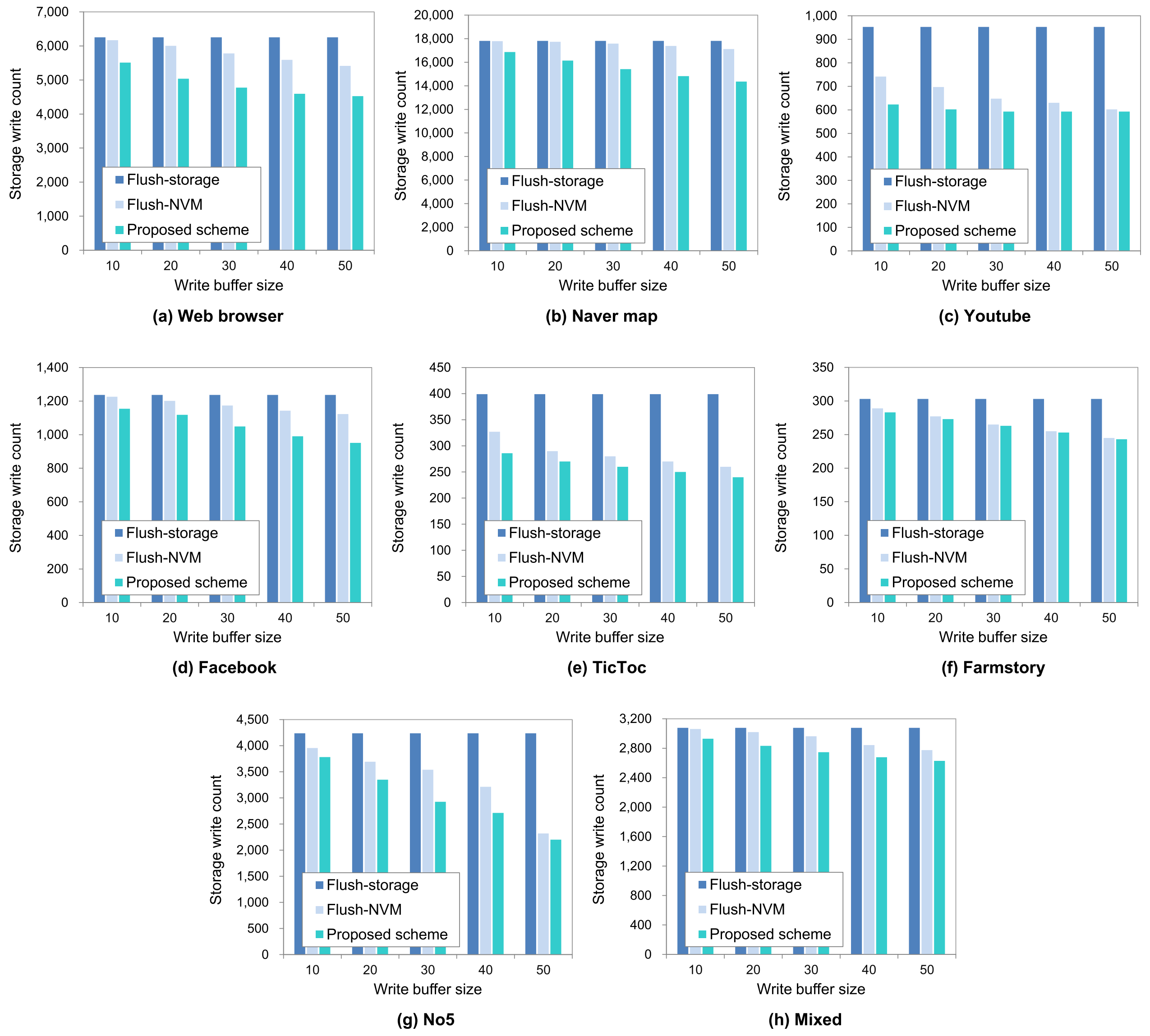

FIGURE 8. **Comparison of the proposed scheme with Flush-storage and Flush-NVM with respect to the storage write traffic.**

variety of workloads and a wide range of the write buffer size. Specifically, the reduced storage access time of the proposed scheme is an average of 18.5% and up to 41.0% compared to Flush-storage and an average of 7.6% and up to 16.0% compared to Flush-NVM. This is mainly because smartphone applications perform frequent synchronous writes to storage, which accounts for the major portion of storage access time. As shown in the figure, the latency of NVM flush accounts for very small portion, and the relative impact of storage write is higher than that of storage read in all cases. Specifically, storage read in Navermap accounts for the lowest portion among the applications we simulated as shown in Figure 9(b). This is consistent with the read/write ratio of file accesses analyzed in Figure 5, where only 6% of file accesses are reads in Navermap. In contrast, Mixed has the highest ratio of reads and also it has the largest number of read accesses in the trace, and thus the relative impact of storage read is quite high as shown in Figure 9(h). However, in any case, reads are mostly absorbed by the buffer cache, so using any scheme made little difference, whereas writes showed a large performance gap depending on the schemes used.

V. RELATED WORKS

A. FILE ACCESSES AND USER BEHAVIOR

Similar to traditional desktop or server systems, file accesses in smartphone applications are generated by the program activities of applications in execution. However, we observed some unique file access characteristics in smartphone applications from several perspectives and investigated their potential implications at the buffer cache layer. Program behaviors are essentially influenced by user activities, and as smartphones have various input devices such as touch

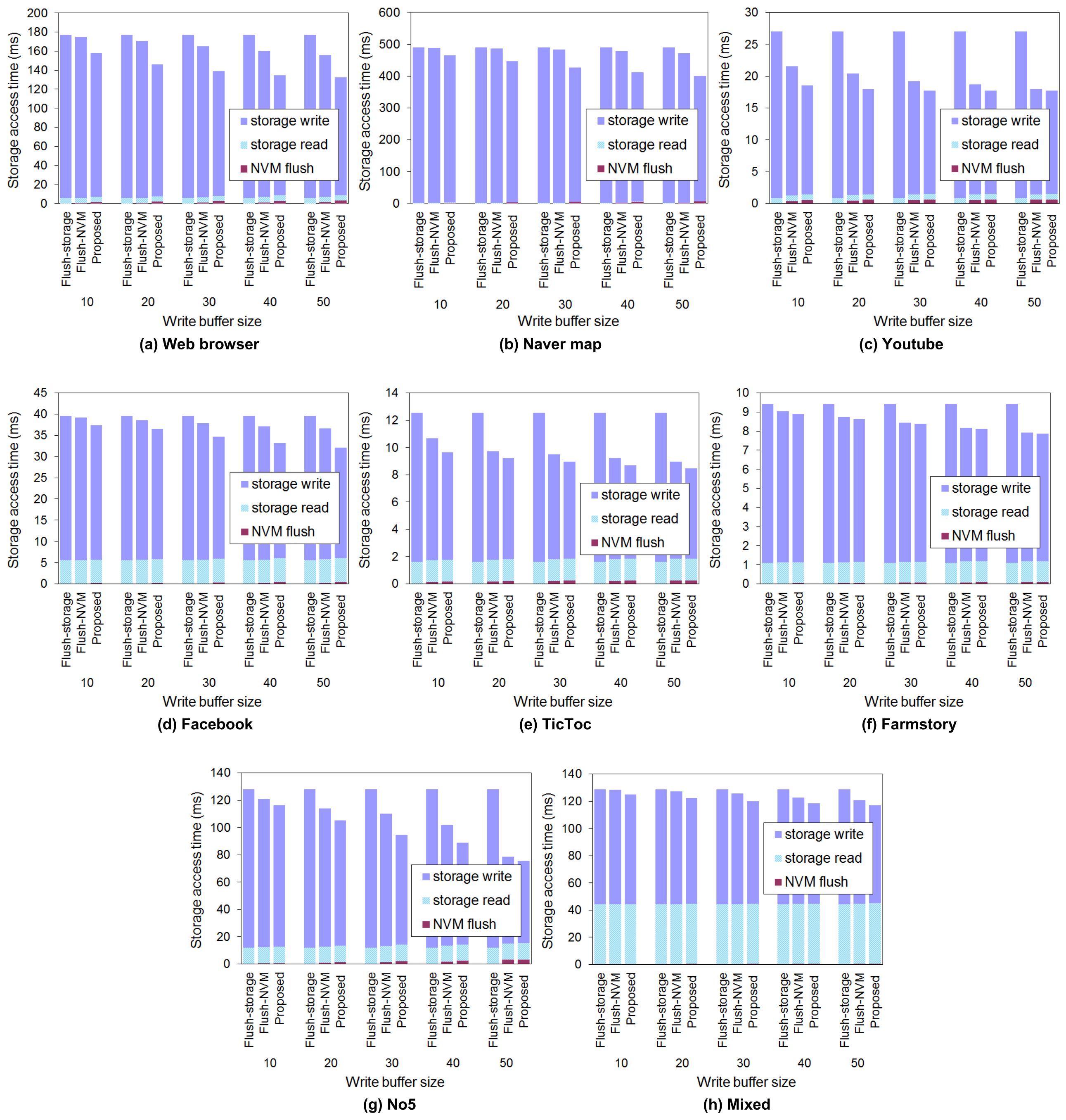

FIGURE 9. Comparison of the proposed scheme with Flush-storage and Flush-NVM with respect to the storage access time.

interfaces and sensors for detecting user activities, one may wonder if the peculiar access patterns come from user behaviors. However, our observations have shown that a user's input behavior does not significantly affect the file access patterns in smartphones. Specifically, during our trace collection process, we ran the same applications several times with different user's input behaviors, but the file access trends did not show meaningful differences. That is, the file access pattern of smartphones seems to be mainly affected by the characteristics of an application itself and the development process, i.e. libraries and tools used, rather than the user's input behavior. Although the file access patterns of an application are not significantly affected by user behaviors, however, application access patterns are highly dependent on user behaviors. Thus, studies on smartphone user behaviors mainly focus on the usage patterns of applications throughout a day (e.g., frequency and duration of using each application) rather than file access patterns by user activities.

Falaki *et al.* analyze the smartphone usage pattern of 255 users and find that the application usage patterns are very different among users, but the application popularity can be modeled using an exponential distribution with different parameters for each user [31]. Li *et al.* characterize and compare the behaviors of Android, iOS, and Windows smartphone users, with respect to network traffic and application types. They identify the mobile platform of each smartphone based on HTTP signatures, and analyze that web browser and instant message services are the two important applications that attract users for choosing their mobile platforms [32]. Ding *et al.* explore the relationship between energy consumption and user behavior in smartphones [33]. To capture user behavior such as application preference and usage time, they design user behavior models and also analyze hardware usage patterns for each application type. Based on this model, they provide the implications of the proposed model for smartphone manufacturers and application developers.

B. BUFFER CACHING WITH NVM

Buffer cache management techniques have been extensively studied to bridge the latency gap in memory and storage accesses. A lot of research has focused on the cache replacement algorithms to improve the cache hit ratio under slow HDD storage. Recently, as storage becomes increasingly fast and the application characteristics are different from traditional systems, cache management policies for emerging hardware and software environments are suggested [34], [35]. Kim *et al.* propose a smartphone buffer cache management policy customized for flash storage systems [34]. Their policy, which is called Spatial-CLOCK, sorts blocks in the buffer cache by the sector number in flash storage and selects victim blocks based on the sector number order. This has the effect of managing flash storage efficiently as cached items whose original storage locations are adjacent are evicted together. Lee *et al.* investigate the effectiveness of buffer cache for fast storage media, and show that caching is still efficient because of the I/O stack overhead and the access patterns of workloads [8]. However, since the gain of caching becomes small, they argue that caching is beneficial only for the data that are certain to be accessed again in the future, and thus some admission control mechanism is necessary. Lee *et al.* show that the effectiveness of buffer cache is limited in NVM, and in some cases, direct I/O performs better than using the buffer cache [18]. Also, they show that I/O traffic is more important than I/O frequency in NVM storage, and thus reducing I/O traffic should be further emphasized in buffer cache management.

Non-volatile buffer cache has been studied to improve the reliability of cached data. Lee *et al.* propose a buffer cache system that unifies the function of caching and journaling [14]. Their policy allows the in-place journaling that eliminates storage accesses while commit, but still provides the same reliability level by simply changing the state of the cached block. Kim and Ahn present the BPLRU (Block Padding Least Recently Used) algorithm for the write buffer replacement in flash storage [36]. BPLRU groups dirty blocks from the same flash storage location, and evicts them together based on the LRU algorithm. When a write access occurs, blocks in the same group are moved together to the highest priority position in the LRU list to reflect the temporal locality. Lee *et al.* present a journaling file system for NVM that aims at reducing the write traffic in journaling [37]. Their file system manages journaling writes efficiently by considering the size of modifications within a block. Shi *et al.* present a write buffer management scheme called ExLRU (Expectation-based LRU) [38]. ExLRU maintains the reference history of blocks in the write buffer and selects a block with the minimum cost as an eviction victim when free space in needed.

VI. CONCLUSION

In this paper, we analyzed the file access characteristics of Android applications and showed that there exist a limited number of hot write blocks, which cannot be buffered under traditional buffer cache as they incur immediate storage flushing. We also analyzed that the re-access likelihood of these hot blocks can be estimated better by frequency rather than temporal locality. Based on these observations, we presented an efficient buffer cache management scheme for smartphone systems by making use of a small non-volatile write buffer. Unlike previous studies, our scheme selectively flushes dirty blocks to write buffer or storage based on the write characteristics of Android block accesses. Eviction in write buffer is also performed judiciously by making use of the write count annotated during the flush operation from buffer cache. Experiment results with real Android workload traces showed that our scheme reduces the storage write traffic by 21.7% on average and up to 48.1% compared to conventional buffer cache and by 10.8% on average and up to 18.1% compared to the same non-volatile write buffer architecture without our judicious management.

In this study, we did not consider the user behaviors in file accesses. As program behaviors are affected by the user behaviors and various sensor and touch devices in smartphones detect user behaviors, it will be interesting to find the dependency between smartphone user behaviors and file access patterns. This will be a good direction for our future research.

REFERENCES

[1] H. Cao and M. Lin, "Mining smartphone data for app usage prediction and recommendations: A survey," *Pervasive Mobile Comput.*, vol. 37, pp. 1–22, Jun. 2017.

[2] N. Islam and R. Want, "Smartphones: Past, present, and future," *IEEE Pervasive Comput.*, vol. 13, no. 4, pp. 89–92, Oct. 2014.

[3] T. Li, M. Zhang, H. Cao, Y. Li, S. Tarkoma, and P. Hui, "'What apps did you use?': Understanding the long-term evolution of mobile app usage," in *Proc. Web Conf.*, Apr. 2020, pp. 66–76.

[4] D. Kim, S. Lee, and H. Bahn, "An adaptive location detection scheme for energy-efficiency of smartphones," *Pervas. Mobile Comput.*, vol. 31, pp. 67–78, Sep. 2016.

[5] E. Lee and H. Bahn, "Electricity usage scheduling in smart building environments using smart devices," *Sci. World J.*, vol. 2013, pp. 1–11, Oct. 2013.

[6] *Google Pixel 5 the Ultimate 5G Google Phone*. Accessed: Oct. 13, 2021. [Online]. Available: https://store.google.com/product/pixel_5

[7] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *ACM Trans. Storage*, vol. 8, no. 4, pp. 1–25, Nov. 2012.

[8] E. Lee and H. Bahn, "Caching strategies for high-performance storage media," *ACM Trans. Storage*, vol. 10, no. 3, pp. 1–22, Jul. 2014.

[9] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modelling Comput. Syst.*, 1999, pp. 134–143.

[10] T. Kim, H. Bahn, and K. Koh, "Popularity-aware interval caching for multimedia streaming servers," *IET Electron. Lett.*, vol. 39, no. 21, pp. 1555–1557, Oct. 2003.

[11] W. Lee, K. Lee, H. Son, W. Kim, B. Nam, and Y. Won, "WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 235–247.

[12] E. Lee, H. Kang, H. Bahn, and K. G. Shin, "Eliminating periodic flush overhead of file I/O with non-volatile buffer cache," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1145–1157, Apr. 2016.

[13] D. Kim, E. Lee, S. Ahn, and H. Bahn, "Improving the storage performance of smartphones through journaling in non-volatile memory," *IEEE Trans. Consum. Electron.*, vol. 59, no. 3, pp. 556–561, Aug. 2013.

[14] E. Lee, H. Bahn, and S. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 73–80.

[15] E. Lee, J. Kim, H. Bahn, S. Lee, and S. H. Noh, "Reducing write amplification of flash storage through cooperative data management with NVM," *ACM Trans. Storage*, vol. 13, no. 2, pp. 1–13, Jun. 2017.

[16] M. Talebi, A. Salahvarzi, A. M. Hosseini Monazzah, K. Skadron, and M. Fazeli, "ROCKY: A robust hybrid on-chip memory kit for the processors with STT-MRAM cache technology," *IEEE Trans. Comput.*, early access, Nov. 26, 2020, doi: 10.1109/TC.2020.3040152.

[17] H. Wang, Y. Zhao, C. Li, Y. Wang, and Y. Lin, "A new MRAM-based process in-memory accelerator for efficient neural network training with floating point precision," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

[18] E. Lee, H. Bahn, S. Yoo, and S. H. Noh, "Empirical study of NVM storage: An operating system's perspective and implications," in *Proc. IEEE 22nd Int. Symp. Modelling, Anal. Simulation Comput. Telecommun. Syst.*, Sep. 2014, pp. 405–410.

[19] Z. Zhang, Z. Shen, Z. Jia, and Z. Shao, "UniBuffer: Optimizing journaling overhead with unified DRAM and NVM hybrid buffer cache," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 9, pp. 1792–1805, Sep. 2020.

[20] W. E. Loewe, R. M. Hedges, T. T. McLarty, and C. J. Morrone, "LLNL's parallel I/O testing tools and techniques for ASC parallel file systems," in *Proc. IEEE Cluster Comput. Conf.*, Apr. 2004, pp. 1–4.

[21] D. Ye, J. Ray, and D. Kaeli, "Characterization of file I/O activity for SPEC CPU 2006," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 112–117, 2007.

[22] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. Mclarty, "File system workload analysis for large scale scientific computing applications," in *Proc. IEEE/NASA Goddard Conf. Mass Storage Syst. Technol.*, Apr. 2004, pp. 139–152.

[23] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 126–137, May 1996.

[24] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. IEEE Conf. Comput. Commun. 18th Annu. Joint Conf. IEEE Comput. Commun. Soc., Future Now (INFOCOM)*, Mar. 1999, pp. 126–134.

[25] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proc. USENIX Annu. Tech. Conf.*, 2000, pp. 1–15.

[26] T. F. Sienknecht, R. J. Friedrich, J. J. Martinka, and P. M. Friedenbach, "The implications of distributed data in a commercial environment on the design of hierarchical storage management," *Perform. Eval.*, vol. 20, nos. 1–3, pp. 3–25, May 1994.

[27] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2187–2200, Sep. 2014.

[28] E. A. Muharemagic, I. O. Mahgoub, and M. Milenkovic, "Analysis of file usage in personal computer environments," *Distrib. Parallel Databases*, vol. 3, no. 4, pp. 315–324, Oct. 1995.

[29] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 187–200.

[30] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, pp. 115–130.

[31] A. A. Bhih, P. Johnson, and M. Randles, "Diversity in smartphone usage," in *Proc. 17th Int. Conf. Comput. Syst. Technol.*, Jun. 2016, pp. 179–194.

[32] Y. Li, J. Yang, and N. Ansari, "Cellular smartphone traffic and user behavior analysis," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2014, pp. 1326–1331.

[33] M. Ding, T. Wang, and X. Wang, "Establishing smartphone user behavior model based on energy consumption data," *ACM Trans. Knowl. Discovery Data*, vol. 16, no. 2, pp. 1–40, Jul. 2021.

[34] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?" in *Proc. 12nd ACM SIGMETRICS Conf. Meas. Modeling Comput. Syst.*, 2012, pp. 235–246.

[35] J. Park, H. Lee, S. Hyun, K. Koh, and H. Bahn, "A cost-aware page replacement algorithm for NAND flash based mobile embedded systems," in *Proc. 7th ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2009, pp. 315–324.

[36] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2008, pp. 239–252.

[37] E. Lee, S. H. Yoo, and H. Bahn, "Design and implementation of a journaling file system for phase-change memory," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1349–1360, May 2015.

[38] L. Shi, J. Li, C. J. Xue, C. Yang, and X. Zhou, "ExLRU: A unified write buffer cache management for flash memory," in *Proc. 9th ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2011, pp. 339–348.

[39] *UFS Version 3.1*. Accessed: Oct. 13, 2021. [Online]. Available: https://www.micron.com/products/managed-nand/universal-flash-storage

[40] *Intel Optane Persistent Memory*. Accessed: Oct. 13, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html

SOOJUNG LIM received the B.S. degree in computer science and engineering from Ewha University, Republic of Korea, in 2012, where she is currently pursuing the Ph.D. degree in computer science and engineering. Her research interests include operating systems, storage systems, embedded systems, software platform technologies, cloud computing, and block chain technologies.

HYOKYUNG BAHN (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1997, 1999, and 2002, respectively. He is currently a Full Professor in computer science and engineering with Ewha University, Seoul, Republic of Korea. He has published more than 100 papers in leading conferences and journals, including USENIX FAST, IEEE Transactions on Computers, IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on Industrial informatics, and *ACM Transactions on Storage*. His research interests include operating systems, caching algorithms, storage systems, embedded systems, system optimizations, and real-time systems. He received the Best Paper Awards at the USENIX Conference on File and Storage Technologies, in 2013.