FSLRU: A Page Cache Algorithm for Mobile Devices with Hybrid Memory Architecture

Dong Hyun Kang and Young Ik Eom

Abstract — *Even though page cache layer of operating system enhances the performance of mobile devices by reducing the number of write requests issued to the underlying mobile storage, the mobile devices still suffer from the excessive write requests. This is because mobile applications frequently trigger synchronous writes with fsync() system call to guarantee the reliability of each application. Unfortunately, these synchronous writes significantly draw both performance and battery power of mobile devices.*

This paper proposes a novel page cache algorithm, called fsync-aware LRU (FSLRU), that adopts hybrid memory architecture, which is composed of DRAM and emerging non-volatile memory (NVM). In particular, the proposed algorithm is designed to overcome the negative performance effect of NVM writes, which is measured on a real board. In order to improve performance and energy efficiency of mobile devices, FSLRU integrates the durability function into page cache layer and provides atomic update operations that are necessary to support strong durability. For detailed performance analyses, the proposed algorithm is implemented on a trace-driven simulator and is evaluated on a real board by replaying the results of the simulator. The evaluation results clearly present that FSLRU outperforms the conventional LRU algorithm by up to 3.2 times under three real world workloads while reducing power consumption by up to 99%[1].

Index Terms — Mobile device, Non-volatile memory, Hybrid memory architecture, Page cache algorithm, Durability, Atomicity.

I. INTRODUCTION

The use of mobile devices, such as smartphones, smart watches, and tablets, is continuously growing. However, mobile devices still suffer from the *excessive write requests* because most mobile applications frequently call *fsync*() system call to prevent unexpected data loss that can result from the battery outages or system crashes [1], [2]. Since these *fsync*() system calls unfortunately draw the overall performance as well as the battery power of mobile devices, reducing the number of *fsync*() system calls is a particularly important in the mobile devices.

Many researchers in both industry and academia have studied the I/O stacks of mobile devices [1]-[3] and have proposed various approaches to reduce the number of *fsync*() system calls [4]-[9]. For example, some research groups focused on the database engine for mobile devices [10], [11] and improved it for better performance because such database engine accelerates the triggering of *fsync*() system calls to guarantee data *durability* and *atomicity*.

Several other researchers have focused on emerging non-volatile memory (NVM) technologies to integrate the benefits of NVM into the current mobile devices because NVM provides several opportunities for performance and energy improvements [12] of the mobile devices. In order to minimize the negative impact of the *fsync*() calls, some approaches placed NVM on the processor memory bus to access data on NVM at a byte granularity. Several studies have shown the positive effectiveness of NVM in terms of performance of mobile devices. Some researchers also focused on database engine of mobile devices and proposed NVM-aware approaches [13]-[16] that can reduce the number of *fsync*() calls incurred by the database engine of mobile devices. Meanwhile, several studies focused on combining the benefits of NVM and the conventional software layers of operating system, including page cache [17], [18] and filesystem layer [19]-[21]. However, prior works are inefficient because the researches employ NVM only for the last level cache [17], [18] or logging area [13]-[16], [21]. In addition, even though it is widely understood that NVM can significantly reduce the power consumption, the battery power of mobile devices has not received much attention in the previous approaches.

The previous version of this work [22] also revisited the page cache layer of the system with hybrid memory architecture, which is composed of DRAM and NVM, and examined the possibility of reducing the number of *fsync*() system calls based on trace-driven simulation. This paper enhances the previous version in terms of *reliability* by providing *atomic update* operations. In addition, this paper includes the extensive evaluation results on a real board [23], [24] and proves energy efficiency of the proposed scheme, comparing it with the other algorithms.

This paper tries to achieve three major contributions. First, this paper investigates three different memory architectures that use NVM and finds a suitable candidate that can effectively leverage NVM on the mobile devices. Second, based on such investigations, this paper proposes a novel page cache algorithm, called *fsync*-aware LRU (FSLRU), that

[1] This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2015M3C4A7065696).

Dong Hyun Kang and Young Ik Eom are with the College of Software, Sungkyunkwan University, 2066, Seobu-ro, Jangan-gu, Suwon 16419, South Korea (e-mail: kkangsu@skku.edu, yieom@skku.edu).

Contributed Paper
Manuscript received 03/31/16
Current version published 06/29/16
Electronic version published 06/29/16

utilizes hybrid memory architecture which is composed of DRAM and NVM, to improve performance and power consumption of mobile devices. Especially, FSLRU efficiently eliminates the synchronous write requests by combining the functionality of page cache and mobile storage. Third, evaluation of FSLRU is performed on a real board composed of DRAM and NVM with real mobile workloads. The evaluation results clearly show that FSLRU outperforms the conventional page cache algorithm in most cases. In the best case, FSLRU reduces the elapsed time of the workloads on a real board by up to 3.2 times over the DRAM based LRU algorithm and by up to 3.7 times over the NVM based LRU algorithm. Moreover, FSLRU significantly saves the limited battery power by up to 99% compared to DRAM based LRU algorithm.

The remainder of this paper is organized as follows. Section II describes the characteristics of NVM in detail and introduces three different memory architectures. Section III presents the detailed design of FSLRU algorithm. Section IV introduces the evaluation methodology and presents evaluation results in terms of performance and energy efficiency. The related work is described in Section V, and finally, Section VI concludes this paper.

II. BACKGROUND

This section describes the unique characteristics of NVM in detail and presents taxonomy of new memory architecture that uses NVM, to find an appropriate candidate for mobile devices.

A. Non-volatile Memory

Emerging non-volatile memory (NVM) technologies, such as spin-torque transfer magneto-resistive memory (STT-MRAM), phase change memory (PCM), and transistor-less cross point, give mobile devices more opportunities to improve the performance and energy efficiency. This is because all NVMs have DRAM-like characteristics such as high performance and byte-addressability as well as storage-like characteristics like non-volatility. However, each NVM provides different performance and density. For example, it is widely known that the access latency of STT-MRAM is almost equal to that of DRAM, but its capacity is limited due to lower density than other NVMs. Meanwhile, PCM is being considered as a potential successor to DRAM because it may be denser than DRAM. Unfortunately, PCM takes slightly higher access latency than DRAM [23]. The performance specification of transistor-less cross point is not yet published, however, some researchers assume it also has the latency gap as in the case of PCM.

B. New Memory Architecture for Mobile Devices

In mobile devices, the access latency of memory is very important because it directly affects the performance of mobile applications. This paper first presents a real access latency of PCM, which is measured on a real board [23], [24], because PCM is a good candidate for new memory architecture of mobile devices due to its high density characteristic. PCM will be referred to as NVM in the rest of this paper. Fig. 1 shows performance comparison between DRAM and NVM after completing 10,000 access requests to each memory. Fig. 1 clearly confirms that the access latency of NVM is slower than that of DRAM in all cases. Especially, the write latency of PCM increases by 5.1 times on average (up to 7.6 times). On the other hand, in case of a read request, the latency of PCM only increases by 1.2 times on average (up to 1.7 times).

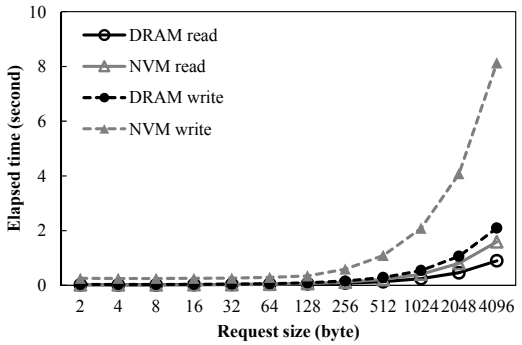

Fig. 1. Comparison of access latencies of DRAM and NVM on a real board. The X axis is in log scale and the Y axis is measured after completing 10,000 access requests.

Based on such observation, this paper introduces new memory architectures for mobile device and describes strengths and weaknesses of each architecture to find a suitable candidate that improves both the performance and energy efficiency of mobile devices. There are three candidate memory architectures for mobile devices: DRAM only memory, NVM only memory, and DRAM and NVM hybrid memory.

Generally, the DRAM only memory architecture has been adopted for mobile devices and it helps improve the performance of devices by reducing the performance gap between CPU and mobile storage. In this case, unfortunately, mobile devices suffer from the volatility characteristic of DRAM. For example, the DRAM only memory architecture negatively impacts mobile devices in terms of overall performance and endurance of mobile storages because mobile applications frequently call *fsync*() system call to guarantee both the *consistency* and *durability* of the application data on DRAM. In addition, since DRAM must be periodically refreshed to preserve data on DRAM even during its idle time, it continually draws the power consumption of mobile devices.

Recently, many researchers consider that the NVM only architecture can be a good candidate for mobile devices because NVM is being considered as a potential successor to DRAM. If this architecture is used in mobile devices, it can save the battery power of the mobile devices because NVM never requires refresh operations for maintaining data on NVM memory. Especially, this architecture can improve the overall performance of mobile devices by eliminating a lot of

I/O requests due to its non-volatility. However, deploying this architecture in mobile devices is quite difficult because NVM has slower access latency than DRAM as shown in Fig. 1.

The last architecture places DRAM and NVM on the same memory bus to directly communicate with each other. This hybrid memory architecture can be classified into two types according to the ratio of the amount each memory. First type is composed of large size DRAM and small size NVM. This type normally employs DRAM as a page cache [17], [18] and small size NVM as an additional memory to give special functionality such as swap [20] and journaling [21]. The first type can lead additional performance improvement by using small size NVM, but it also draws the battery power of mobile devices due to the same reasons of DRAM only memory architecture. Second type consists of small size DRAM and large size NVM. This type is a good candidate for mobile devices because of two reasons. One is that this hybrid memory architecture helps improve the performance and energy efficiency of mobile devices by employing large size NVM as a page cache and the other is that it can mitigate the latency gap by using small size DRAM as a cache of NVM.

III. DESIGN OF FSLRU ALGORITHM

This paper proposes a novel page cache algorithm, called *fsync*-aware LRU (FSLRU), that combines the functionality of page cache and the mobile storage by adopting the hybrid memory architecture, which is composed of small size DRAM and large size NVM. To fully take the benefits of hybrid memory architecture, FSLRU extends the traditional least recently used (LRU) algorithm and it separately manages the pages in each memory of the hybrid memory architecture with two LRU lists: DQueue for DRAM and NQueue for NVM. DQueue has a volatile list and NQueue has a persistent list. All requests from CPU are handled by scanning each list of FSLRU separately.

A. Design Goals

FSLRU is designed to achieve the following goals. First, the proposed algorithm should keep higher hit ratio in order to reduce the number of slow storage I/Os. Second, it should try to eliminate the storage writes caused by *fsync*() system calls because these synchronous writes heavily draws the overall performance and endurance of mobile storage. Third, FSLRU has to provide the storage functionality like *durability* and *atomicity*, which can be achieved by converting storage writes caused by *fsync*() call into memory writes, to satisfing the second design goal.

B. In-memory Fsync

In order to eliminate the synchronous writes issued to the underlying mobile storage, FSLRU integrates the *durability* function into the page cache layer, which is implemented by *in-memory fsync* operation. For *in-memory fsync*, NQueue uses a special bit for each page, called *dura bit*, that is set when *fsync*() system call is triggered by a mobile application. The *dura bit* of each page tells whether the page is durable or not. When *fsync*() is called by an application, FSLRU performs *in-memory fsync* operation by the following steps in order. (1) FSLRU first searches the volatile list of DQueue to find dirty pages related to the specified *fsync*() system call; (2) It persistently copies the dirty pages to the persistent list of NQueue instead of flushing them to the underlying mobile storage; (3) Then, FSLRU changes the dirty bit of the copied (original) pages in the volatile list of DQueue from 1 to 0; (4) Finally, FSLRU finds all dirty pages related to the specified *fsync*() call in the persistent list of NQueue again and then it sets the *dura bit* of the pages in the persistent list to 1. A page whose *dura bit* is 1 will be referred to as a *durable page* in the rest of this paper. In this way, FSLRU completely guarantees the *durability* of dirty pages in the page cache layer.

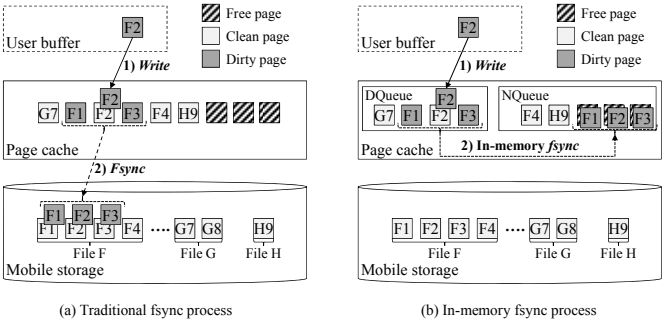

Fig. 2. Comparison between the traditional fsync and in-memory fsync process

Fig. 2 demonstrates the traditional *fsync*() call process with that of the FSLRU. In this figure, the traditional *fsync*() simultaneously writes three dirty pages (F1, F2, and F3) of the file F to the underlying mobile storage (Fig. 2a). On the other hand, FSLRU simply copies the dirty pages on the volatile list of DQueue into the persistent list of NQueue to make them *durable pages* (Fig. 2b). A *durable page* in the persistent list of NQueue is flushed to the underlying mobile storage when the page is evicted from the persistent list according to its replacement policy.

C. Page Cache Replacement Algorithm

Fig. 3 shows the page management and replacement process of that FSLRU algorithm, which has two sub-algorithms, DQueue LRU algorithm and NQueue LRU algorithm.

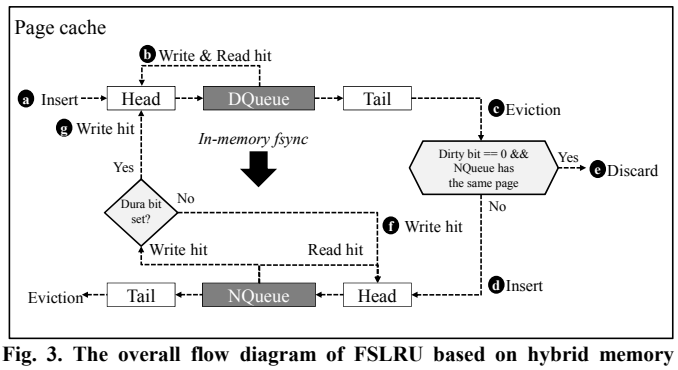

Fig. 3. The overall flow diagram of FSLRU based on hybrid memory architecture composed of small size DRAM and large size NVM

In the case of a read request, both FSLRU algorithms follow the basic rules of the traditional LRU policy. Therefore, in the rest of this section, the key technique for a write request will be described in detail. If a write or read request is handled by DQueue algorithm, the request is not forwarded to NQueue algorithm because NQueue can also have the same page in its persistent list as a *durable page* as mentioned before.

DQueue LRU Algorithm: As shown in Fig. 1, the write latency of NVM can negatively impact the performance of mobile applications. Therefore, in FSLRU, a new page is always inserted into the volatile list of DQueue (Fig. 3a). If DQueue receives a write request, it first scans its volatile list to find the requested page. If the page is found in the volatile list (cache hit), DQueue first updates the contents of the page with its dirty bit set and then records the updated offset and size for *atomic update* operation. Finally, it moves the updated page in the volatile list of DQueue (Fig. 3b). On a cache miss, DQueue forwards the missed request to NQueue to make it handle the request. If NQueue also does not have the requested page in its persistent list, DQueue finds a free page in its volatile list for insert operation. If DQueue has a free page in the volatile list, it simply returns the page to handle the write request. Otherwise, to reclaim a free page, DQueue selects a victim page at the tail of the volatile list and evicts the selected page to NQueue (Fig. 3c).

NQueue LRU Algorithm: If NQueue receives a page that is evicted from DQueue, it first tries to find the same page in its persistent list. This is because the same page might be copied from DQueue to NQueue as a *durable page* when the *in-memory fsync* is triggered. If NQueue does not find the *durable page* in its persistent list (cache miss), it inserts the received page into the head of the persistent list of NQueue for improving the cache hit ratio (Fig. 3d). On a cache hit, NQueue checks the dirty bit of the received page. If the dirty bit of the page is 0, the received page is discarded without any storage write (Fig. 3e) because the page has not been updated after the last *in-memory fsync* operation. Otherwise, NQueue carefully reflects the received page to the *durable page* in the persistent list of NQueue through *atomic update* operation.

If NQueue receives a write request, it searches its persistent list to find the requested page. If the page is found in the persistent list (cache hit), NQueue first checks whether the page is *durable* or not. If the page is not *durable*, NQueue directly updates the contents of the page with its dirty bit set and then moves the page to the head of the persistent list (Fig. 3f). Otherwise, since this page should be preserved on the persistent list of NQueue to guarantee the *durability* of the page, NQueue inserts a new page into the volatile list of DQueue by copying the *durable page* (Fig. 3g). As a result, the same page can be placed on both the volatile list of DQueue and the persistent list of NQueue simultaneously. The page inserted to the volatile list of DQueue will be reflected to the *durable page* in the persistent list of NQueue when a mobile application calls *fsync*() system call or the inserted page is selected as a victim page of the volatile list of DQueue (Fig. 3c).

D. Atomic Update

In order to completely preserve *durable pages* placed in the persistent list of NQueue, FSLRU should provide *atomicity* function in the page cache layer, which is performed by *atomic update* operation. This operation atomically updates the *durable pages* to prevent a partial page update and is triggered during the *in-memory fsync* or the eviction process of DQueue algorithm. To reduce the expensive copy cost, FSLRU supports two atomic update operations: *byte atomic update* and *page atomic update*.

Fig. 4 shows the pseudo-code of the two atomic update operations. FSLRU goes through the following steps for each atomic operation. When *atomic update* operation is triggered, FSLRU first checks the update size of the page which is the target of the *atomic update*. If the update size is smaller than half of a page size (2048 byte), the page is reflected to its *durable page* in the persistent list of NQueue with *byte atomic update*: this operation first copies the part that is to be updated in the *durable page* to a free page for recovery and then reflects the latest data of the target page to the *durable page* at a byte granularity. This operation helps improve the overall performance and energy efficiency of mobile devices because it significantly reduces the amount of copied data. On the other hand, if the update size of the target page is larger than half of a page size, this page is reflected to its *durable page* with *page atomic update*: first, this operation copies the data of the target page to a free page in the persistent list of NQueue and then sets its *dura bit* to 1, and second, it deletes the old *durable page* in the persistent list.

```
runAtomicupdate (NQueue lru, page *p) {
   dp = lru.getDurablepage(p);
   np = lru.getFreepage();
   if (p->updatedsize <= 2048) {  // byte atomic update
      np = copyByte(dp, p->offset, p->length);
      dp = copyByte(p, p->offset, p->length);
      delete np;
   } else {  // page atomic update
      np = lru.getFreepage();
      np = copyPage(p);
      delete dp;
   }
   updateList(lru);
}
```

Fig. 4. The pseudo code for atomic update operation

IV. EVALUATION

This section first describes the methodology for evaluation. Then, it shows analysis results based on trace-driven simulation and performance results experimented on a real board [23], [24] which is composed of real DRAM and NVM.

A. Evaluation Methodology

This paper introduces an evaluation framework that is based on trace-driven simulations and experiments on a real board.

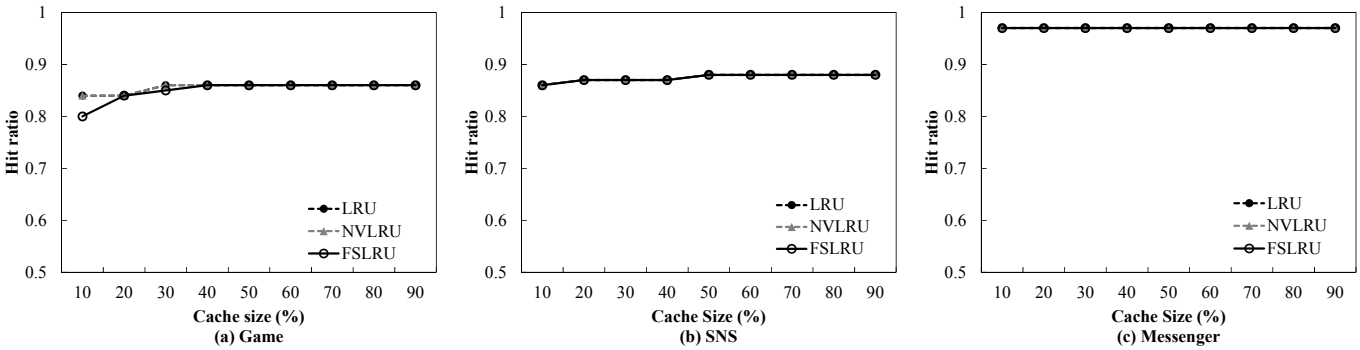

Fig. 5. Comparison of the cache hit ratio of each algorithm for three real world traces

The framework uses real mobile workloads that are collected by running popular mobile applications on real mobile devices. Table I summarizes the statistics of the collected workloads. The game trace was collected by running one of the famous mobile games. The social network service (SNS) trace was collected by performing a lot of relevant operations, such as reading news feed, sending messages, and updating photo files. Finally, the messenger trace was obtained from well-known messenger application that sends and receives a lot of text messages. These real traces are used for input of the cache simulator that implements the three page cache replacement algorithms: LRU, NVLRU, and FSLRU. LRU and NVLRU were implemented for DRAM only and NVM only architecture, respectively. To take the benefits of NVM, NVLRU provides *durability* function by using copy-on-write (COW) mechanism like the *page atomic update* of FSLRU. Since FSLRU employs the hybrid memory architecture composed of small size DRAM and large size NVM, the size of DRAM was set to be 10% of total hybrid memory size.

All experimental results were collected based on the total memory footprint of each workload. Each replacement algorithm was run by varying its page cache size from 10% to 90% to produce the cache hit ratio and the *output trace*, which is composed of storage I/O and memory access requests used by the page cache algorithm. To compare FSLRU with the real memory performance, the *output trace* of each algorithm was replayed on a real board that is equipped with a dual core CPU, 1GB DRAM, 512MB PCM, and 16GB SD Card [23], [24].

TABLE I
STATISTICS OF WORKLOADS

Trace	Sync counts	Read counts	Write counts	Memory footprint
Game	312	4027	6657	5.7MB
SNS	2339	2475	11198	6.3MB
Messenger	14399	43663	15412	6.6MB

B. Cache Hit Ratio

This section first presents the comparison of the cache hit ratio, considering that FSLRU may simultaneously place a page on both the DRAM and NVM to provide efficient *durability* feature. Fig. 5 shows the cache hit ratios simulated for the three page cache algorithms with three different workloads. As shown in Fig. 5, all algorithms maintain high hit ratios regardless of the workloads. The main reason is that mobile workloads have a small working set size. As a result, even in the case when the page cache size is 10%, most requests can be handled in the page cache with a cache hits. Fig. 5 clearly confirms that the FSLRU maintains similar hit ratios to other algorithms in all workloads, even though it sometimes violates the page cache philosophy by locating a page on both DRAM and NVM.

C. Analysis of Storage I/O Requests

The storage I/O requests in an *output trace* are composed of write, read, and sync write requests. The write and read requests are generated by an eviction operation or cache miss operation, respectively. Meanwhile, the sync write requests are generated when *fsync*() system call is triggered, and the sync write may issue excessive synchronous writes to the storage.

Fig. 6 shows distributions of each type of requests generated by each replacement algorithm. As shown in Fig. 6, the total number of write requests for each algorithm is continually reduced as its page cache size increases. Meanwhile, the results clearly confirm that the total number of requests issued by LRU algorithm is the largest in all cases. This is because LRU algorithm periodically issues synchronous write requests to the underlying mobile storage to guarantee *durability* of dirty pages. In addition, these synchronous writes are harmful on mobile devices because the number of write requests is not reduced even though the total page cache size increases.

For detailed analysis of this reason, Table II lists the distance of *fsync*() system calls, which means the number of write requests between two consecutive *fsync*() system calls, that is calculated based on the collected traces. Table II clearly confirms the fact that the page cache should frequently issue the synchronous write requests to the underlying mobile storage because all applications have lower *fsync* distance than 4 on average.

TABLE II
DISTANCE OF FSYNC SYSTEM CALL

Trace	Min	Max	Average
Game	1	108	3.48
SNS	1	13	1.89
Messenger	1	16	1.59

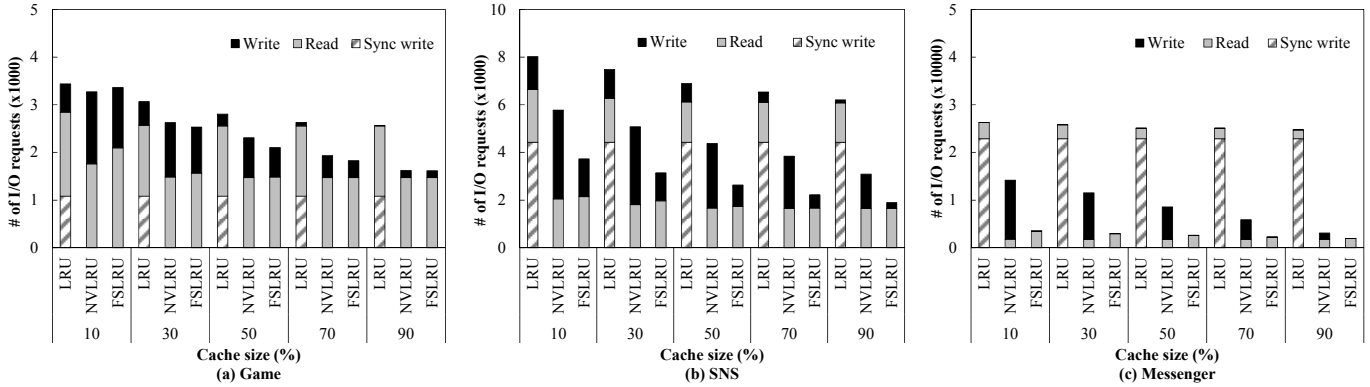

Fig. 6. Distributions of each type of I/O requests that are generated by each replacement algorithm

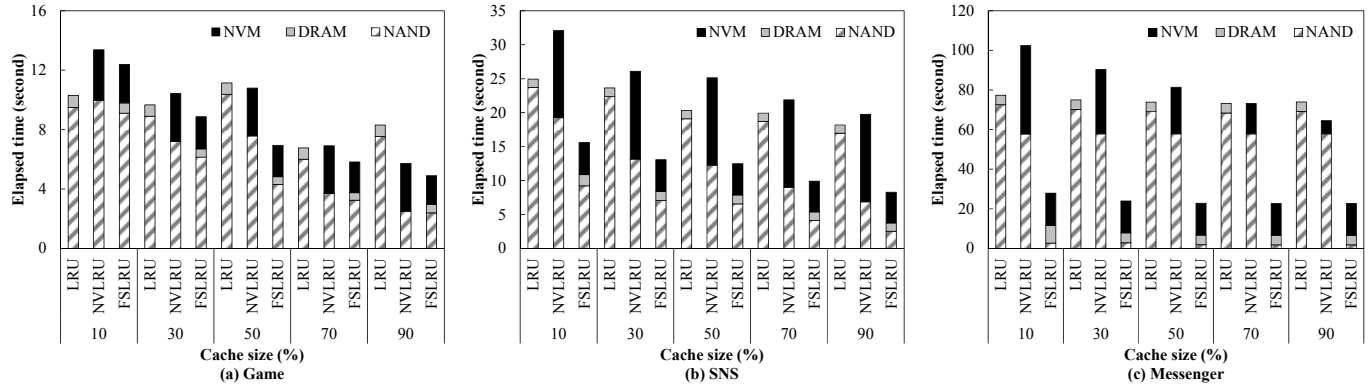

Fig. 7. Calculated real elapsed time based on the output traces of each algorithm

On the other hand, NVLRU and FSLRU present lower amount of I/O requests than LRU algorithm in all cases because both algorithms efficiently provide *durability* of dirty pages in the page cache layer instead of flushing them to the underlying mobile storage. Unfortunately, NVLRU reveals a much higher write requests than FSLRU. This is because NVLRU provides *durability* of dirty pages at a page granularity with COW operation even though a page is partially updated. In other words, NVLRU sometimes evicts a write request to the underlying storage to handle small size update. As expected, FSLRU shows the lowest number of I/O requests as shown in Fig. 6, because it significantly eliminates the number of storage writes by using both *in-memory fsync* and *atomic update* operations. In the case of messenger workload, FSLRU can reduce the total number of requests by up to 11.8 times compared with LRU and by up to 4 times compared with NVLRU.

D. Performance Effect on a Real Board

It is hard to present the effectiveness of *in-memory fsync* and *atomic update* of FSLRU without using real NVM memory. Thus, the evaluation framework replays the *output trace* of each replacement algorithm on a real board.

Fig. 7 presents the total elapsed time of each workload on a real board as a bar type. The time is measured by calculating each memory access time on the board: DRAM, NVM, and NAND. Unlike the analysis results in Fig. 6, Fig. 7 shows two interesting performance results. First, even though NVLRU significantly reduces the number of requests issued to the underlying mobile storage, it sometime reveals slower performance than LRU algorithm. The main reason of these results is that NVLRU does not consider the performance characteristics of the NVM writes shown in Fig. 1. The second interesting result is that frequent *fsync*() system calls considerably exacerbates both the performance and endurance of mobile storage because DRAM based LRU algorithm spends most of the time on flushing dirty pages to the underlying mobile storage. On the other hand, FSLRU achieves good performance as shown in Fig. 6. In the best case, FSLRU reduces the elapsed time by up to 3.2 times over the LRU algorithm and by up to 3.7 times over the NVLRU algorithm. The main reason for these reductions is that FSLRU efficiently eliminates both expensive storage writes and NVM writes by providing *durability* and *atomicity* functionalities in the page cache layer. In particular, these performance results clearly confirm that just 10% DRAM in the hybrid memory architecture is large enough to mitigate the negative performance effect of NVM writes.

E. Energy Efficiency

In mobile devices, power consumption is one of the most important issues since most devices must cope with limited energy resources. This section shows the benefits of FSLRU in terms of the energy efficiency. Emerging NVM technologies give mobile devices two opportunities to save its battery power.

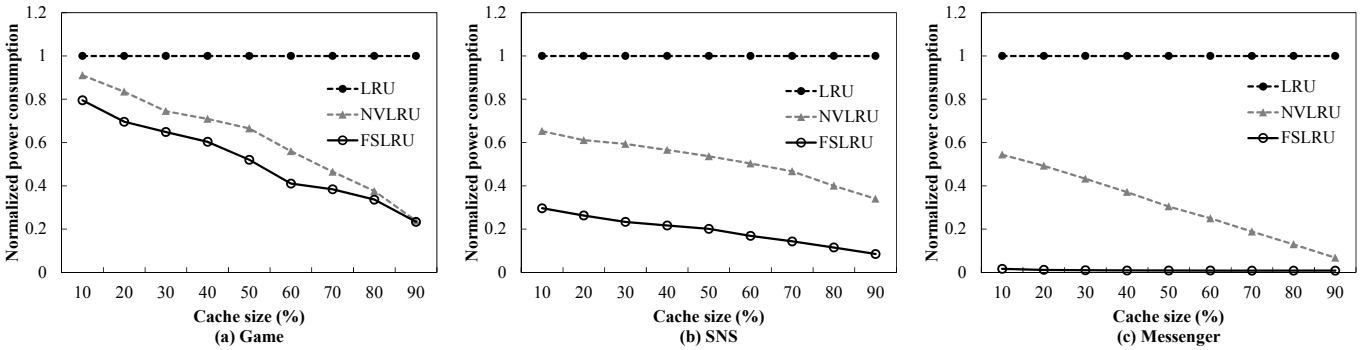

Fig. 8. Normalized power consumption (watt): all values are normalized by results to those of LRU algorithm.

First, unlike DRAM, NVM can prevent continual battery drain because it never requires refresh operations for maintaining data on NVM memory. Second, NVM can reduce power consumption induced by memory or storage operations by providing *durability* function. Since the first opportunity is clear and well-known, this paper deals with the energy efficiency that is related to memory and storage accesses. Table III lists the parameters [23]-[25] for calculating energy efficiency of each replacement algorithm.

TABLE III
PARAMETERS FOR CALCULATING ENERGY EFFICIENCY

Trace	DRAM	NVM	NAND
Read energy	0.1 nJ/bit	0.2 nJ/bit	3.3 mA/page
Write energy	0.1 nJ/bit	1nJ/bit	3.3 mA/page
Read latency	50 ns	50 ns	30 μs
Write latency	50 ns	250 ns	300 μs

Fig. 8 presents normalized power consumption for each replacement algorithm based on the *output trace*. The power consumption is calculated by multiplying the request size of each access and the read/write energy and then by dividing the result into the read/write latency. As shown in Fig. 8, FSLRU significantly reduces power consumption compared with LRU algorithm: from 20% to 76% for game, from 70% to 91% for SNS, and from 98% to 99% for messenger. Moreover, compared with NVLRU, the proposed algorithm saves battery power by up to 26.6%, 74.9%, and 97.6% in game, SNS, and messenger workload, respectively. The reason of this significant difference is that FSLRU efficiently reduces the number of expensive writes issued to the underlying mobile storage by providing *durability* function. Also, these results clearly confirm that the hybrid memory architecture composed of small size DRAM and large size NVM is suitable mobile devices because it can reduce the expensive energy cost of NVM writes by using small size DRAM.

V. Related Work

The performance of mobile devices has received a lot of interests in both industry and academia. This section briefly describes the related work.

Early works focused on I/O behavior [1]-[3] and software layers [4]-[9] of mobile applications to improve the performance of mobile devices. Nguyen *et al*. [6] investigated the I/O behavior on smartphones and proposed SmartIO approach to minimize application I/O delay. Jeong *et al*. [8] studied the effectiveness of asynchronous I/O on mobile devices and suggested QASIO scheme to reduce response latency. Jeong *et al*. [1] investigated the relationship between database engine for mobile devices and journaling of the operating system and reported the journaling of journal problem. Kim *et al*. [10] focused on data structures of a mobile database engine and proposed LS-MVBT approach to reduce the I/O traffic caused by such database engine. Lee *et al*. [11] proposed WALDIO journal mode to bypass the page cache layer of the operating system.

In order to improve the performance of mobile devices, some researchers suggested employing NVM in mobile devices [13]-[21]. A lot of research and approaches have shown the positive effectiveness of NVM in mobile environments. Kim *et al*. [13] proposed a journaling technique of database engine for mobile devices because the engine frequently calls *fsync*() to guarantee both the *consistency* and *durability* of the data at the application level. This technique efficiently reduces the number of synchronous writes by using NVM as a dedicated logging area. Oh *et al*. [14] also modified the database engine of mobile devices to gain the advantages of NVM. Kim *et al*. [21] proposed delta journaling scheme that utilizes small size NVM to reduce the journaling overhead of the filesystem layer. Lin *et al*. [17] proposed a page cache algorithm based on hybrid memory architecture which is composed of DRAM and NVM. In order to reduce the number of *fsync*() calls, the proposed algorithm used NVM as the last level cache between the DRAM page cache and the underlying mobile storage. However, prior work is inefficient because it employs NVM only for the last level cache [17], [18] or logging area [13], [15], [16], [21]. In addition, even though it is widely understood that NVM can significantly reduce the power consumption, the battery power of mobile devices has not received much attention in the previous approaches.

VI. Conclusion

In mobile devices, page cache layer in the operating system helps improve the performance of mobile devices by reducing

the number of write requests issued to the underlying mobile storage. However, excessive synchronous writes of mobile applications draw overall performance and energy efficiency of mobile devices by reducing the benefits of page cache layer. Therefore, reducing the amount of synchronous writes is very important in mobile devices.

In order to reduce such synchronous writes, this paper first focused on the three different memory architectures using emerging NVM and proposed a suitable candidate to effectively leverage NVM on the mobile devices. This paper also proposed a novel page cache algorithm, called FSLRU, whose key mechanism is providing *durability* and *atomicity* functionalities in page cache layer by adopting the hybrid memory architecture composed of small size DRAM and large size NVM. Evaluation results on a real board clearly confirm that FSLRU improves the total elapsed time by up to 3.2 over the conventional LRU and by up to 3.7 over the NVLRU while saving the battery power of mobile devices.

REFERENCES

[1] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," *in Proc. USENIX Annual Technical Conference*, San Jose, USA, pp. 309-320, Jun. 2013.

[2] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," *in Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 287-293, Feb. 2014.

[3] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *in Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 1-14, Feb. 2012.

[4] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?," *in Proc. ACM SIGMETRICS/PERFORMANCE joint International Conference on Measurement and Modeling of Computer Systems*, London, England, pp. 235-246, Jun. 2012.

[5] D. H. Kang, C. Min, and Y. I. Eom, "Block utilization-aware buffer replacement scheme for mobile NAND flash storage," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 9, pp. 2510-2513, Sep. 2014.

[6] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang, "Reducing smartphone application delay through read/write isolation," *in Proc. ACM Annual International Conference on Mobile Systems, Applications, and Services*, Florence, Italy, pp. 287-300, May 2015.

[7] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," *In Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 273-286, Feb. 2015.

[8] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting quasi-asynchronous I/O for better responsiveness in mobile devices," *in Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 191-202, Feb. 2015.

[9] K. Lee and Y. Won, "Smart layers and dumb result: IO characterization of android-based smartphone," *in Proc. ACM International Conference on Embedded Software*, Tampere, Finland, pp. 23-32, Oct. 2012.

[10] W.-H. 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 Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 273-285, Feb. 2014.

[11] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, "WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly," *in Proc. USENIX Annual Technical Conference*, Santa Clara, USA, pp. 235-247, Jul. 2015.

[12] X. Luo, D. Liu, L. Liang, Y. Li, K. Zhong, and L. Long, "MobiLock: An energy-aware encryption mechanism for NVRAM-based mobile devices," *in Proc. IEEE Non-Volatile Memory System and Applications Symposium*, Hong Kong, China, pp. 1-6, Aug. 2015.

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

[14] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," *in Proc. International Conference on Very Large Database*, Kohala Coast, Hawaii, pp. 1454-1465, Aug. 2015.

[15] H. Luo, L. Tian, and H. Jiang, "qNVRAM: Quasi non-volatile RAM for low overhead persistency enforcement in smartphones," *in Proc. USENIX Workshop on Hot Topics in Storage and File Systems*, Philadelphia, USA, pp. 1-5, Jun. 2014.

[16] S. Ryu, K. Lee, and H. Han, "In-memory write-ahead logging for mobile smart devices with NVM," *IEEE Transactions on Consumer Electronics*, vol. 61, no. 1, pp. 39-46, Feb. 2015.

[17] Y.-J. Lin, C.-L. Yang, H.-p. Li, and C.-Y. M. Wang, "A buffer cache architecture for smartphones with hybrid DRAM/PCM memory," *in Proc. IEEE Non-Volatile Memory System and Applications Symposium* Hong Kong, China, pp. 1-6, Aug. 2015.

[18] J. Park, E. Lee, and H. Bahn, "DABC-NV: A buffer cache architecture for mobile systems with heterogeneous flash memories," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 4, pp. 1237-1245, Nov. 2012.

[19] H. G. Lee, "High-performance NAND and PRAM hybrid storage design for consumer electronics," *IEEE Transactions on Consumer Electronics*, vol. 56, no. 1, pp. 112-118, Feb. 2010.

[20] K. Zhong *et al*., "Building high-performance smartphones via non-volatile memory: the swap approach," *in Proc. ACM International Conference on Embedded Software*, New Delhi, India, pp. 1-10, Oct. 2014.

[21] J. Kim, C. Min, and Y. I. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *IEEE Transactions on Consumer Electronics*, vol. 60, no. 2, pp. 217-224, May 2014.

[22] D. H. Kang and Y. I. Eom, "FS-LRU: A page cache algorithm for eliminating fsync write on mobile device," *in Proc. IEEE International Conference on Consumer Electronics*, Las Vegas, USA, pp. 351-352, Jan. 2016.

[23] H. Chung *et al*., "A58nm 1.8V 1Gb PRAM with 6.4MB/s program BW," *in Proc. IEEE International Solid-State Circuits Conference Digest of Technical Papers*, San Francisco, USA, pp. 500-502, Feb. 2011.

[24] T. Lee, D. Kim, H. Park, S. Yoo, and S. Lee, "FPGA-based prototyping systems for emerging memory technologies," *in Proc. IEEE International Symposium on Rapid System Prototyping*, New Delhi, India, pp. 115-120, Oct. 2014.

[25] S. Eilert, M. Leinwander, and G. Crisenza, "Phase Change Memory: A new memory enables new memory usage models," *in Proc. IEEE International Memory Workshop*, Monterey, USA, pp. 1-2, May 2009.

BIOGRAPHIES

Dong Hyun Kang received the B.S. degree in Computer Engineering from Korea Polytechnic University, Korea in 2007 and the M.S. degree in College of Information and Communication Engineering from Sungkyunkwan University, Korea in 2010. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at Sungkyunkwan University. His research interests include storage systems, operating systems, and embedded systems.

Young Ik Eom received his B.S., M.S., and Ph.D. degrees in Computer Science and Statistics from Seoul National University, Korea in 1983, 1985, and 1991, respectively. From 1986 to 1993, he was an associate professor at Dankook University in Korea. He was also a visiting scholar in the Department of Information and Computer Science at the University of California, Irvine, from Sep. 2000 to Aug. 2001. Since 1993, he has been a professor at Sungkyunkwan University in Korea. His research interests include virtualization, operating systems, cloud systems, and system securities.