Eliminating Periodic Flush Overhead of File I/O with Non-Volatile Buffer Cache

Eunji Lee, Hyojung Kang, Hyokyung Bahn, and Kang G. Shin, *Fellow, IEEE*

Abstract—File I/O buffer caching plays an important role to narrow the wide speed gap between main memory and secondary storage. However, data loss or inconsistency may occur if the system crashes before updated data in the buffer cache is flushed to storage. Thus, most operating systems adopt a daemon that periodically flushes dirty data to secondary storage. This periodic flush degrades the caching efficiency seriously because most write requests lead to direct storage accesses. We show that periodic flush accounts for 30-70 percent of the total write traffic to storage. To remove this inefficiency, this paper presents a new buffer cache architecture that adopts a small amount of non-volatile memory to maintain modified data. This novel buffer cache architecture removes almost all storage accesses due to periodic flush operations without any loss of reliability. It also improves the buffer cache performance through space-efficient management techniques, such as delta-write and fragment-grouping. Our experimental results show that the proposed buffer cache reduces the storage write traffic by 44.3 percent and also improves the throughput by 23.6 percent on average.

Index Terms—Non-volatile memory, buffer cache, file system, pdflush, reliability

1 Introduction

Due to the widening speed gap between main memory and hard disks, I/O operations are becoming the performance bottleneck of modern computer systems. In order to improve the performance of file I/O, operating systems use a buffer cache that stores requested file blocks in a certain portion of main memory, thereby servicing subsequent requests directly without accessing slow storage media.

As traditional buffer cache uses volatile media such as DRAM, the file system may enter an inconsistent and/or out-of-date state when the system crashes before the change is reflected to permanent storage. To relieve this problem, modern file systems perform journaling or periodic flush operations that transfer the updated data to non-volatile storage within a short time period [1]. The flush operation reflects the modified data directly to its original location in the file system, while the journaling operation writes the changes to separate storage area called the *journal area* and then reflects them to the original location later. Though journaling guarantees more reliable file system states because it withstands system crashes during storage updates, it generates a large number of additional storage writes. Thus, most systems adopt journaling only for metadata and flush regular data. Though journaling and flush improve the reliability of file systems, they degrade the effectiveness of buffer caching significantly due to frequent storage accesses even when the cache space is not exhausted [2], [3].

In this paper, we analyze the source of write I/Os in file system workloads and show that periodic flush operations account for 30-70 percent of total write traffic to storage. In order to eliminate these excessive flush operations, we propose a new buffer cache architecture that only adds a small amount of non-volatile memory to the buffer cache and stores modifications to this non-volatile buffer cache. When non-volatile memory is used as buffer cache, periodic flush operations are not needed since the contents of the buffer cache are retained even when the power failure occurs. This novel buffer cache architecture removes almost all storage accesses for periodic flush operations without any loss of reliability via judicious management of the non-volatile buffer cache.

Due to recent advances in non-volatile memory technologies such as phase-change memory (PCM) or spin torque transfer magnetic RAM (STT-MRAM), non-volatile memory is expected to be used as main memory of computer systems in the near future [4], [5], [6], [7]. However, as non-volatile memory will not completely replace DRAM due to cost, it is considered only as an add-on component to enhance performance [8]. In this paper, we show that only a small amount of non-volatile buffer cache suffices to eliminate most flush operations by space-efficient management techniques, such as delta write and fragment-grouping.

By replaying representative file system workloads, we show that the proposed buffer cache reduces the write traffic to storage by 44.3 percent on average and up to 73.0 percent. A prototype of this buffer cache has also been implemented on Linux 2.6.38. Our measurement with the Filebench and IOzone benchmarks show that the buffer cache improves throughput by 23.6 percent on average and up to 43.3 percent over the existing Linux buffer cache with pdflush.

A number of researchers have attempted to relieve the journaling overhead by making use of non-volatile memory technologies [9], [10], [11], [12]. Our work is different from theirs in that we reduce the flush overhead incurred by

- *E. Lee is with the Department of Software, Chungbuk National University, Cheongju 361-763, Korea. E-mail: eunji@cbnu.ac.kr.*
- *H. Kang and H. Bahn are with the Department of Computer Science and Engineering, Ewha University, Seoul 120-750, Republic of Korea. E-mail: hyoj_kang@ewhain.net, bahn@ewha.ac.kr.*
- *K. G. Shin is with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122. E-mail: kgshin@eecs.umich.edu.*

Manuscript received 4 Feb. 2014; revised 5 Aug. 2014; accepted 5 Aug. 2014. Date of publication 18 Aug. 2014; date of current version 17 Mar. 2016. Recommended for acceptance by H. Asadi, P. Ienne, and H. Sarbazi-Azad. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2014.2349525

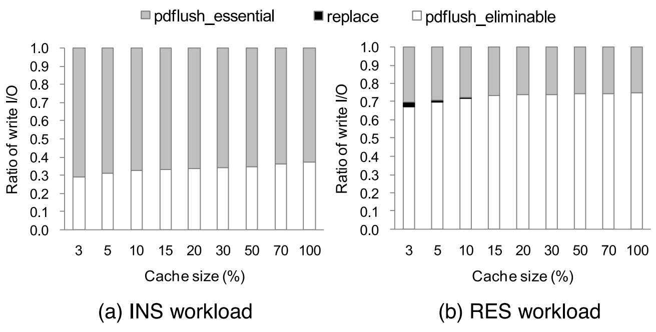


Fig. 1. Source of write I/O as the size of buffer cache is varied.

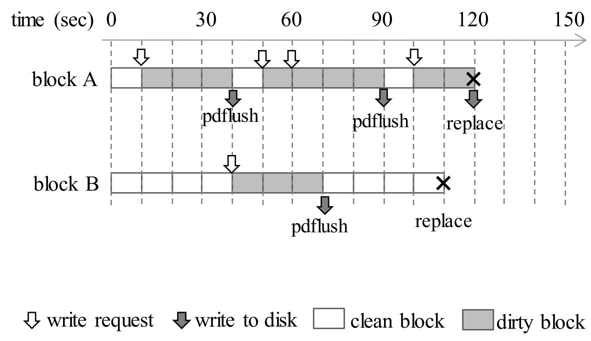


Fig. 2. An example of write I/O from the buffer cache.

regular data rather than journaling overhead for metadata. Furthermore, we use only a small amount of non-volatile buffer cache.

The main contributions of this paper can be summarized as follows.

- We quantify the overhead of periodic flush in real world file I/O traces, and show that it accounts for 30-70 percent of storage writes.
- To eliminate such overhead, we design a novel buffer cache architecture that adopts a small amount of non-volatile memory.
- We present space-efficient buffer cache management techniques such as delta-write and fragment grouping that can be utilized as soon as the limited capacity of the first generation non-volatile memory product becomes available.
- A Linux-based prototype has been implemented and measurements thereon show that our buffer cache improves the file system performance significantly by eliminating excessive write traffic to storage.
- Our approach would be preferred from the compatibility perspective since it can be implemented as an add-on design that does not influence the functionality of the existing DRAM buffer cache.

The remainder of this paper is organized as follows. Section 2 analyzes the overhead of periodic flush in the write traffic to storage. Section 3 details our buffer cache architecture and algorithm. Section 4 presents a brief description of the experimental conditions and discusses the performance evaluation results. Section 5 summarizes the work relevant to this paper, and finally, Section 6 concludes the paper.

2 Analysis of Periodic Flush Overhead in File I/O

In this section, we analyze the overhead of periodic flush operations (which we will henceforth call *pdflush*) by showing the amount of write traffic from the buffer cache to storage. We used the system-call traces collected in the NOW project at UC Berkeley, a representative file I/O workload trace [13]. Each request in the trace consists of a file ID, an offset, a request length in bytes, a requested time, and an operation type.

The total amount of data written to storage is measured while varying the buffer cache size from 3 to 100 percent of the total footprint. The LRU (Least Recently Used) algorithm is used as the buffer cache replacement policy. The cache size of 100 percent implies the infinite cache capacity where all blocks referenced in the trace can be cached simultaneously, and thus replacement is not needed. This is an unrealistic condition but presented to show the complete write traffic trend while varying the cache size.

When we use pdflush, there are two different cases of causing storage writes. The first case occurs when the cache is full and a dirty block is selected as a victim to make room for caching a new block. In this case, the block needs to be written to storage first before being removed from the cache. This kind of writes can be reduced when the cache capacity becomes large since a large cache lowers the frequency of cache replacement. The second case occurs when pdflush—which writes the updated data periodically to permanent file system locations—occurs. For example, in the default Linux configuration, the pdflush daemon wakes up every 5 seconds and flushes all dirty blocks that were updated more than 30 seconds ago. This periodic flush is necessary since the buffer cache consists of volatile memory, and thus the file system may enter an inconsistent and/or out-of-date state when the system crashes before the change is reflected to permanent storage.

Fig. 1 shows the source of write traffic as the cache size is varied. We use two representative workload traces collected in NOW projects, INS and RES. (The characteristics of these workloads will be detailed in Section 4.) In Fig. 1, we categorize the write traffic to *replace*, *pdflush_essential*, and *pdflush_eliminable*. *Replace* represents the write traffic that occurs when dirty blocks are evicted from the cache. As shown in the figure, except for the small amount of writing that occurs with the 3 percent size cache of RES, the write traffic by *replace* almost did not happen. This is because most of the dirty blocks were already written to storage by *pdflush* before replaced from the cache, remain in the clean state.

We subdivide the write traffic by pdflush as *pdflush_essential* and *pdflush_eliminable*. *Pdflush_essential* refers to the write I/O that occurs even when pdflush is not used. That is, the write I/O of *pdflush_essential* will eventually occur by cache replacement if pdflush is turned off. A write I/O by an explicit `sync` operation is also included to *pdflush_essential*. In contrast, *pdflush_eliminable* is a write I/O that does not occur if we do not use pdflush. This implies that an additional write request for the same block is certain to occur before that block is replaced from the cache. For example, in Fig. 2, a write request for block A occurs at 60 seconds and it is flushed to storage at 90 seconds. Then, another write request occurs at 100 seconds and block A is evicted from the cache at 120 seconds. In this case, pdflush

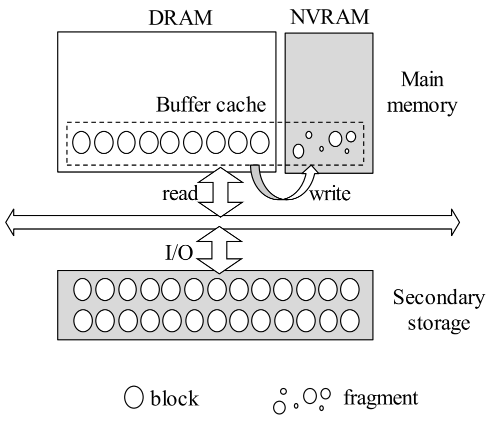


Fig. 3. System architecture of the proposed buffer cache.

at 90 seconds is *pdflush_eliminable* as there is a write request before it is evicted from the cache. In contrast, a write request for block B is at 40 seconds and there is no more writes on block B before it is evicted at 110 seconds. In this case, pdflush that occurs at 70 seconds is *pdflush_essential* as the write I/O will eventually happen upon replacement even though we do not use pdflush. Thus, *pdflush_essential* should be excluded when assessing the pure overhead of pdflush. As shown in Fig. 1, *pdflush_eliminable* accounts for 30-40 percent and 70-75 percent of the total write traffic in INS and RES, respectively, which can be eliminated with our non-volatile buffer cache. As the cache size increases, the ratio of *pdflush_eliminable* also increases. This is because the ratio of *pdflush_essential* decreases with a large cache capacity as cache replacement happens less frequently.

In summary, pdflush accounts for a considerable portion of storage writes, and is thus a potential source of performance degradation. Specifically, in the INS and RES workloads, the data written by pdflush amounts to 30-75 percent for all range of cache sizes, which can be eliminated with our novel buffer cache architecture.

3 Buffer Cache with Non-Volatile Memory

We now describe an efficient buffer cache management scheme that adopts non-volatile memory in conjunction with DRAM as buffer cache.

3.1 System Architecture

Fig. 3 shows the basic architecture of the proposed buffer cache that consists of DRAM (referred to as *volatile-buffer cache*) and a small amount of non-volatile memory (referred to as *non-volatile buffer cache*). Our non-volatile buffer cache is placed in standard DIMM slots to access it through a byte-addressable interface. There exist several types of non-volatile memory media, such as PCM, FeRAM, and STT-MRAM. Recently, PCM and STT-MRAM have been drawing considerable interest from the research community due to their rapid improvement in micro-fabrication processes [6], [14]. However, PCM has critical weaknesses to absorb write I/Os in our buffer cache as it has limited write endurance and slow write performance compared to DRAM. For this reason, PCM is usually adopted to absorb read-intensive workloads in memory systems [4], [7]. We use STT-MRAM as our non-volatile buffer cache because it does not have such limitations in write operations.

Volatile buffer cache manages cached data by the block (as a unit), while non-volatile buffer cache does this by the byte. In our buffer cache, the volatile buffer cache behaves exactly the same as the existing buffer cache upon all read and write requests. In contrast, the non-volatile buffer cache maintains only the modified part of a block (which we call *fragment*) upon a write request. Thus, modifications are reflected to both the volatile and the non-volatile buffer cache. Data reflected to the volatile buffer cache is used to service normal requests. For example, read requests for dirty blocks are serviced through the volatile buffer cache. Data in the non-volatile buffer cache is used only when a system crash occurs to restore the recent image of the file system. This eliminates the large storage write traffic caused by traditional pdflush operations.

As our buffer cache stores modified data in both volatile and non-volatile buffer cache, space-efficiency can be deteriorated. However, as our non-volatile buffer cache maintains only the modified part of a block, this inefficiency can be minimized. File access characterization studies reported that the size of a modified part within a block is very small for most write requests and a large proportion of updates are at most 10 percent different from the previous content of the block [39]. We will discuss this further in Section 3.4.

3.2 The Proposed Algorithm

For now, as the capacity of non-volatile memory is limited, a space-efficient management for the non-volatile buffer cache is needed. In this paper, instead of storing an entire block to the non-volatile buffer cache, we only maintain the modified part of a block, thereby improving the space-efficiency of the non-volatile memory. Instead, as the modification is also reflected to the volatile buffer cache, all read/write and flush operations can be performed by referencing the volatile buffer cache. Note that the non-volatile buffer cache is used only when a power failure occurs. When free space is needed in the volatile buffer cache, we basically use the LRU[1] algorithm, the most popular replacement algorithm used in the buffer cache. When a dirty block is selected as the victim block in the volatile buffer cache, it is first written to storage, and then discarded. As all modifications of a block are also maintained in the non-volatile buffer cache, we also remove the fragments of the victim block from the non-volatile buffer cache. Note that these fragments have already been reflected to storage, and do not cause inconsistencies any longer.

When free space is needed in the non-volatile buffer cache, modified fragments belonging to the same block should be merged and replaced together. This is because fragments in the non-volatile buffer cache need to be flushed to storage before their eviction and the minimum unit of storage writing is a logical block. Instead of generating a logical block to be flushed by merging dirty fragments in our scheme, the corresponding block maintained in the volatile buffer cache is searched and that block is written to

1. Actually, commodity operating systems do not use the original LRU replacement algorithm, but perform reclamation periodically to reserve a certain number of free blocks. In our implementation, we do not modify the original reclamation module of the volatile buffer cache in Linux kernel as we are interested in the management of the non-volatile buffer cache rather than that of the volatile buffer cache.

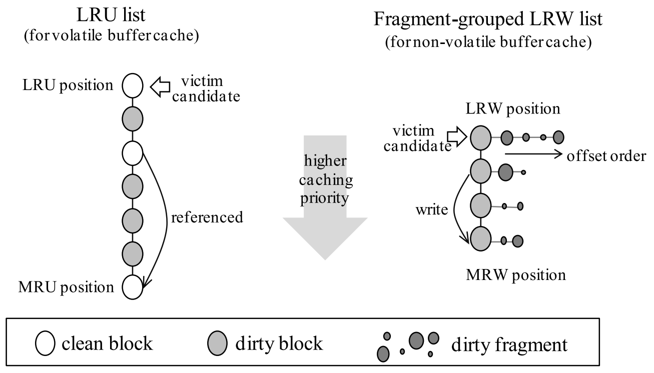


Fig. 4. The LRU and the LRW lists managed for the proposed buffer cache.

storage. This is possible as all modifications were already reflected to the block maintained in the volatile buffer cache. By so doing, we need not merge fragments in the non-volatile buffer cache to generate a block to be flushed.

After storage flushing, all fragments belonging to this block are discarded from the non-volatile buffer cache. The corresponding block in the volatile buffer cache does not change its priority but its state is changed from dirty to clean. Thus, even though a block is evicted from the non-volatile buffer cache, it is possible that the same block still has a high caching priority in the volatile buffer cache if it is a read-intensive data. Accordingly, our buffer cache guarantees the same cache hit ratio of existing buffer cache architectures, but eliminates the writing overhead of pdflush.

3.3 Implementation Issues of the Algorithm

We now describe how the non-volatile buffer cache works. Our non-volatile buffer cache maintains only the modified part of a block (i.e., fragment) for space-efficiency. A fragment consists of the modified data itself and its metadata. The metadata here refers to the pointers to link fragments, the offset within the block, and the length in bytes. Fragments belonging to the same block are linked by their offset order and the first fragment of the block is linked to the block header.

Our buffer cache maintains two lists to capture the recency of references as shown in Fig. 4. One is the LRU list for the volatile buffer cache, and the other is the least recently written (LRW) list for the non-volatile buffer cache. The LRU list keeps track of the recency of references (both for read and write references) of blocks in the volatile buffer cache. That is, all blocks in the volatile buffer cache are sorted by their last reference time regardless of their operation type (i.e., read or write). On the other hand, the LRW list keeps track of the recency of write references for fragments in the non-volatile buffer cache. As evicted data from the non-volatile buffer cache is written to storage by the block, the LRW list groups fragments from the same block and manages them together. When a fragment is written, the block header linked to this fragment moves to the MRW (Most Recently Written) position, thereby moving all fragments from the same block together.

When a write request occurs and the requested block is not in the buffer cache, it is retrieved from the storage and cached in the volatile buffer cache first. Then, the modified part is reflected to both volatile and non-volatile buffer caches. Note that the retrieval process can be eliminated and the write can be performed directly to a free cache block if a write request is of an entire logical block size. The block is, then, located at the most recently referenced position in both LRU and LRW lists. When a write request occurs and the requested block already exists in the buffer cache, then the modification is reflected to both caches and the block moves to the most recently referenced position in the two lists. Upon a read request, only the volatile buffer cache is managed by the LRU algorithm.

When a write request occurs, the modification can be partially or fully overlapping with existing fragments in the non-volatile buffer cache. In this case, it may be merged into an existing fragment, bridge a gap between two existing fragments, or create a new large fragment and delete the existing fragments. This can be performed efficiently as fragments from the same block are linked by their offset order.

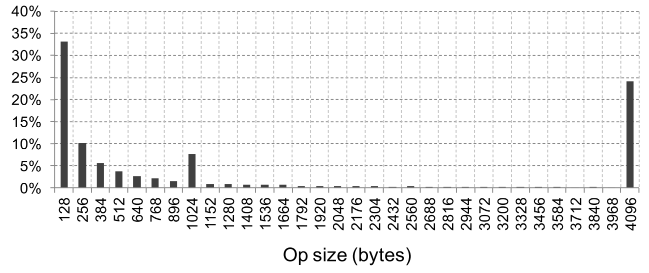
As the size of a fragment may not be uniform, allocation and reclamation of the non-volatile buffer cache possibly generate variable-size holes. This incurs the dynamic memory allocation problem that determines the hole to be allocated to a new fragment. There are well-known allocation policies such as first fit, best fit, and worst fit. However, these policies have limitations in managing a large number of variable-size holes. To solve this problem, we use the buddy system that has been widely used for dynamic memory management in kernel because it is fast and space-efficient [15]. In the buddy system, free memory space is managed by the size of power of 2. If a fragment of a certain size needs to be allocated, the buddy system tries to find the smallest slot that the fragment fits, whose size is certain to be a power of 2. The buddy system splits a free space into halves or merges two adjacent free slots to allocate a best-fit, if necessary.

Our buffer cache is capable of restoring a file system into the most recent state without data loss in the event of power failure. The system recovery consists of two phases. In the first phase, all blocks whose fragments were maintained in the non-volatile buffer cache are loaded from storage to the volatile buffer cache. Then, the blocks in the volatile buffer cache are updated via replaying the fragments in the non-volatile buffer cache, thereby enabling users to view the up-to-date data consistently. We do not need to flush the cache data during the recovery process as they will be flushed later when replaced from the cache.

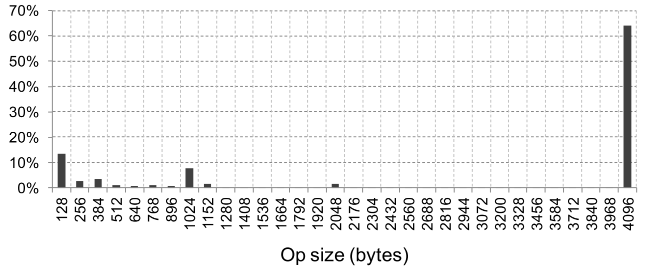
It is possible that the non-volatile memory itself fails. To cope with this situation, writing data successfully against the hardware failure of the non-volatile memory should also be ensured. However, this would be more like an issue that needs to be addressed in the device controller layer via ECC or CRC techniques rather than in system layers such as operating systems or file systems. Our research focuses on the software techniques that the operating system layer is responsible for. Thus, such an issue is beyond the scope of our research, which can be assumed to be provided as a built-in feature of commercial STT-MRAM products.

3.4 Analysis of Request Size Distributions in File Write Operations

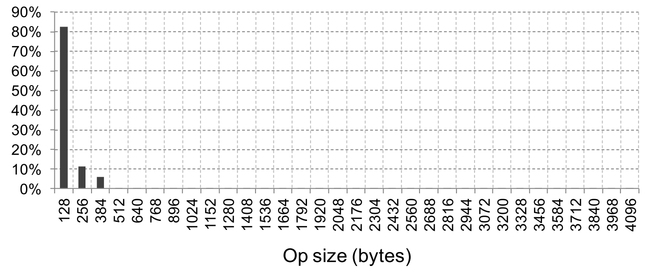
The proposed algorithm tries to optimize the space efficiency of the non-volatile buffer cache by caching only the modified part of a block. To quantify the effectiveness of the



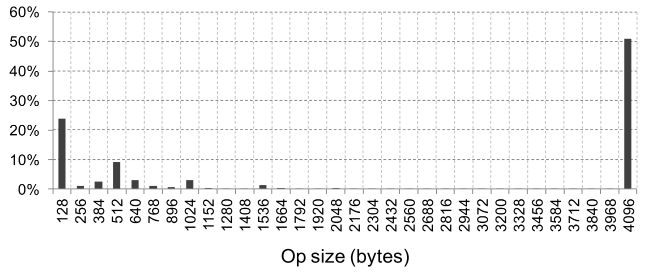
(a) INS



(b) RES



(c) TPC-C



(d) Linkbench

Fig. 5. Request size distribution in file write operations.

proposed algorithm with respect to the update size, we plot the distribution of the request size in each write operation in real file system traces. Fig. 5 shows the request size distribution for the four workloads listed in Table 1. (Details of these workloads will be explained in Section 4.) As shown in the figure, a large proportion of file write operations only involve a small modification of the previous version. Specifically, we have observed that 38 percent of write requests are less than 128 bytes on average. TPC-C contains a large number of small writes and metadata updates, and more than 80 percent of its updates are less than 128 bytes. Only 0.002 percent of write requests involve a modification of an entire block (i.e., 4 KB) or larger. This is the evidence of the space-efficiency by adopting our algorithm.

4 Performance Evaluation

In this section, we present the performance evaluation results to assess the effectiveness of the proposed buffer cache. We first show the write traffic of the proposed buffer cache in Section 4.1 using simulation. In Section 4.2, we show the recovery overhead of the proposed scheme in comparison with other schemes. Then, we compare the throughput of the proposed buffer cache with existing systems using measurements in Section 4.3.

4.1 Comparison of Write Traffic

To assess the effectiveness of the proposed buffer cache with respect to the write traffic to storage, we perform trace-driven simulations. We developed a hybrid buffer cache simulator that employs non-volatile memory along with DRAM as buffer cache. The size of a block in the volatile buffer cache is set to 4 KB, which is common to most operating systems including Linux. Though non-volatile memory such as STT-MRAM or PCM allows byte-addressability, main memory can be accessed with a unit of word. Thus, the minimum operation unit of the non-volatile buffer cache should be at least 4 bytes in 32 bit machines and 8 bytes in 64 bit machines if a memory architecture similar to DRAM is adopted. Moreover, as writing to our non-volatile buffer cache occurs when the last-level cache memory flushes its dirty block to main memory, the fragment size should also be of this cache block size. We assume a 32 bit architecture and the 128 byte block of the last-level cache in this paper.

The traces used in the experiments are one of the representative file I/O traces collected in the NOW project of UC Berkeley [13]. They are extracted from the general-purpose workstations using HP-UX. They consist of undergraduate instructional workload (denoted as INS) and graduate research workload (denoted as RES) [16]. We also collected file I/O traces of Linkbench (a graph generator used in facebook) and TPC-C (a database application that performs financial transactions). We traced the system call requests using the `strace` utility. Characteristics of the traces are described in Table 1.

Fig. 6 shows the software architecture of the developed cache simulator. Our simulator consists of eight modules implemented in seven source code files. Before the simulation, we preprocess traces in various forms into a uniform format with six fields, which are the request time, the operation type, the file id, the block number within a file, the offset within a block, and the operation size. When the simulation begins, the volatile cache simulator first runs with the converted traces, and it invokes the NVRAM write-buffer simulator or the periodic flush daemon as needed. We also implemented a system recovery module that measures the recovery time when a system failure occurs.

TABLE 1
Characteristics of the Traces Used in the Experiments

Workload	Total # of references	Total # of distinct blocks	Ratio of ops. (read : write)
INS	12,473,845	162,588	17.2 : 1
RES	750,303	46,820	1 : 2.53
TPC-C	1,563,785	57,400	1 : 1.35
Linkbench	2,306,971	205,600	1.41 : 1

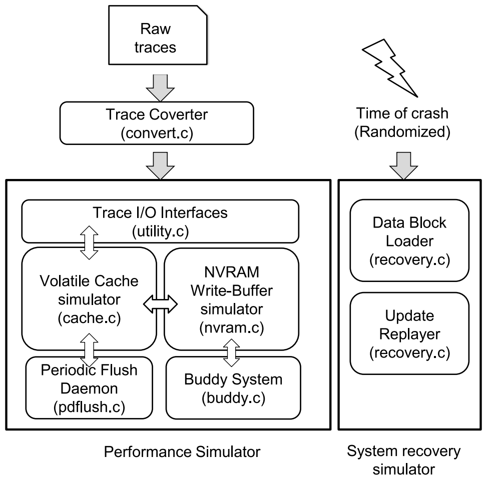
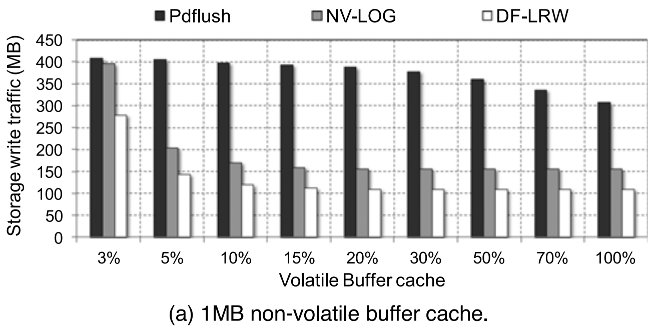

Raw traces
Trace Coverter (convert.c)
Time of crash (Randomized)
Trace I/O Interfaces (utility.c)
Volatile Cache simulator (cache.c)
NVRAM Write-Buffer simulator (nvram.c)
Periodic Flush Daemon (pdflush.c)
Buddy System (buddy.c)
Data Block Loader (recovery.c)
Update Replayer (recovery.c)
Performance Simulator
System recovery simulator

Fig. 6. Software architecture of the developed cache simulator.

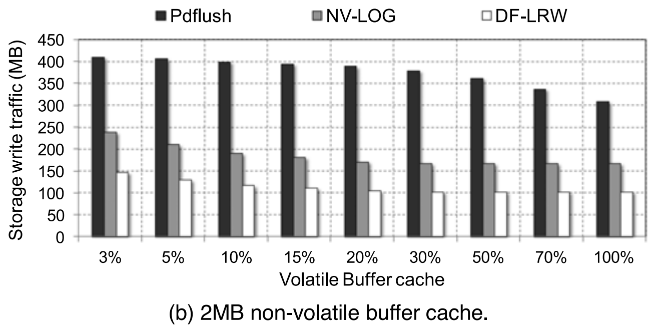
We compare the amount of data written to storage for our scheme, which we call DF-LRW (Delta write & Fragment-grouping LRW), the legacy buffer cache that uses the LRU algorithm with pdflush, and the NV-LOG scheme that adopts non-volatile memory as a logging device to reduce write traffic to storage [12]. Our scheme differs from NV-LOG in that we manage non-volatile memory as a write cache thus unifying multiple updates for the same data as a single caching item, whereas NV-LOG just appends updates in non-volatile memory. As in most operating systems, pdflush in our experiments is invoked every 5 seconds and flushes all the data updated more than 30 seconds ago. In our experiments, the size of the non-volatile buffer cache is changed from 1 to 8 MB. Note that the size of STT-MRAM used in our simulation is configured by considering the footprint of the workload traces. Thus, the relative size of STT-MRAM in comparison with the DRAM size and the workload footprint size needs to be the focus of interest rather than the physical STT-MRAM size. In addition, as the hardware technology of STT-MRAM is not mature yet, it is difficult to estimate the exact capacity of STT-MRAM products for now. Even if the capacity becomes larger than that used in this paper, investigating the effectiveness of our scheme with a small size of STT-MRAM is valuable. Specifically, as the multiprogramming degree of modern computer systems is being increased, their working-set would also become larger than the workload used in our experiments.

Fig. 7 compares the storage write traffic for the legacy buffer cache with pdflush, DF-LRW, and NV-LOG under the RES workload, where the x-axis represents the size of the volatile buffer cache ranging from 3 to 100 percent of total footprint and the y-axis represents the amount of data written to storage for the given cache size. For DF-LRW and NV-LOG, the size of the non-volatile buffer cache is also varied from 1 to 8 MB as shown in Figs. 7a, 7b, 7c, and 7d.

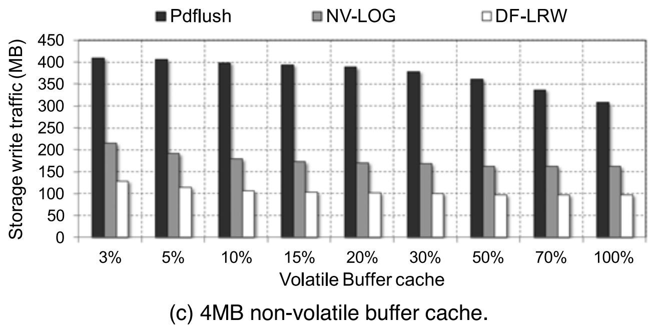
Our buffer cache reduces the storage write traffic of pdflush significantly for all cache sizes. Specifically, the amount of data written to storage is decreased by 69.8 percent on average only with a small amount of non-volatile memory. The reason for this large reduction lies in the elimination of *pdflush_eliminable* introduced in Section 2. When the size of the non-volatile buffer cache is very small, the storage write traffic is expected to be large due to frequent



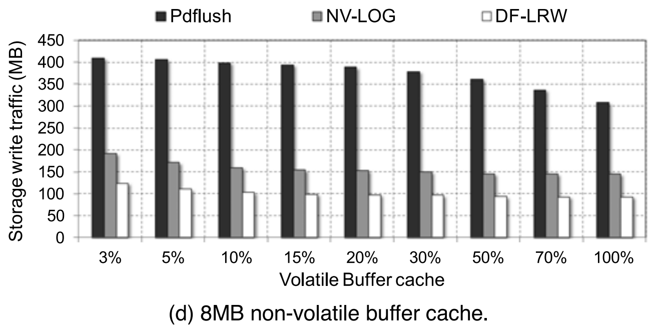
(a) 1MB non-volatile buffer cache.



(b) 2MB non-volatile buffer cache.



(c) 4MB non-volatile buffer cache.



(d) 8MB non-volatile buffer cache.

Fig. 7. Storage write traffic with the RES workload.

cache replacement. However, as shown in the figure, for all of 1, 2, 4, and 8 MB cache sizes, the storage write traffic is consistently small. The write traffic is slightly large only when the size of both volatile and non-volatile buffer caches is extremely small as shown in the leftmost graph of Fig. 7a. When compared to NV-LOG, our scheme reduces the write traffic by 36.0 percent on average. This reduction is obtained as our scheme unifies multiple updates for the same data as a single caching item, whereas NV-LOG just appends updates in non-volatile memory, degrading the space efficiency.

Fig. 8 shows the write traffic of pdflush, NV-LOG, and our scheme under the INS workload. Similar to the result for the RES workload, our buffer cache with DF-LRW

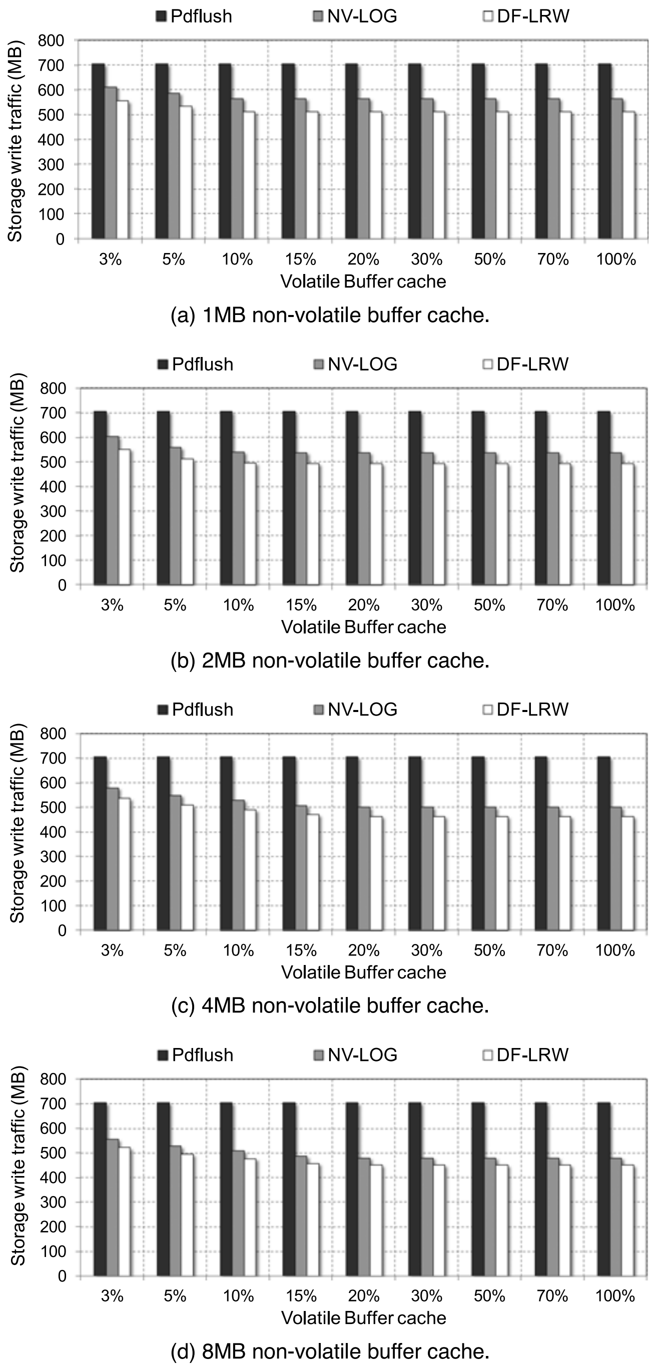


Fig. 8. Storage write traffic with the INS workload.

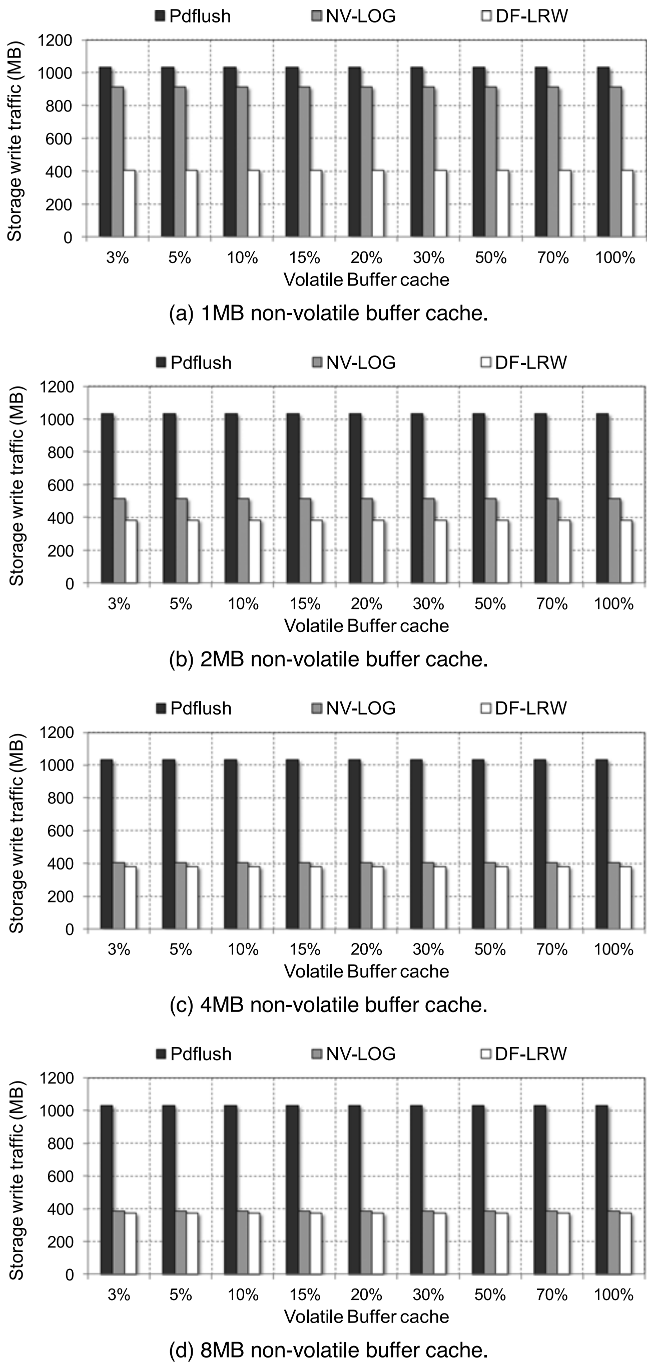
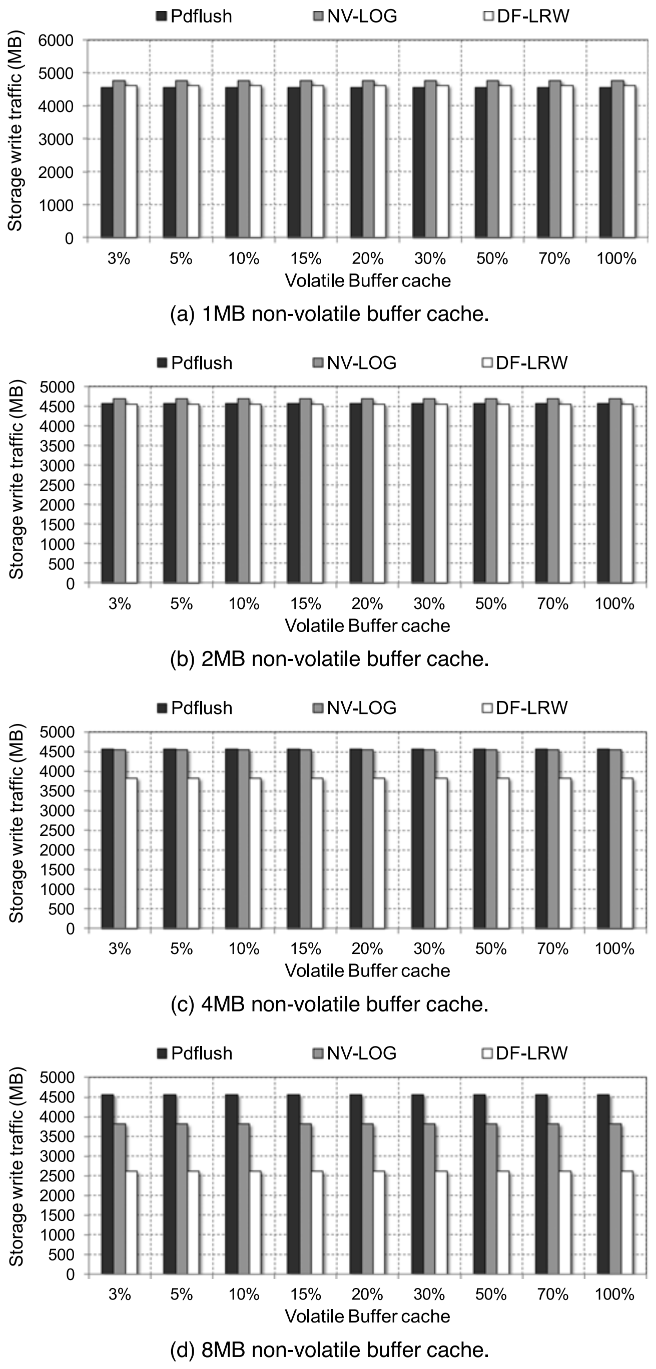


Fig. 9. Storage write traffic with the TPC-C workload.

reduces the amount of data written to storage significantly. Specifically, the write traffic is decreased by 30.1 and 8.0 percent on average compared to pdflush and NV-LOG, respectively. The improvement under the INS workload is relatively smaller than that under RES because INS is a read-intensive workload while RES is write-intensive. As shown in Table 1, the ratio of read to write operations is 1:2.53 in the RES workload, but it is 17.2:1 in the INS workload. As under RES, we can observe that the size of the non-volatile buffer cache needed to reduce this write traffic is very small.

Fig. 9 compares the storage write traffic for pdflush, DF-LRW, and NV-LOG under the TPC-C workload. As shown in the figure, DF-LRW reduces the storage write traffic significantly for all cache sizes compared to pdflush. This is because TPC-C has a large proportion of small updates as shown in Fig. 5, which allows DF-LRW to buffer more write requests with a small non-volatile buffer cache. The write traffic of NV-LOG is also reduced dramatically when the size of the non-volatile buffer cache is larger than 2 MB, but the performance gap between NV-LOG and DF-LRW becomes wider as the cache size becomes small. The write traffic of DF-LRW is decreased by 62.6 and 21.7 percent compared to pdflush and NV-LOG, respectively, on average.

Fig. 10 compares the storage write traffic when the Linkbench workload is used. As shown in the figure, our scheme reduces the storage write traffic of pdflush significantly



(a) 1MB non-volatile buffer cache.

(b) 2MB non-volatile buffer cache.

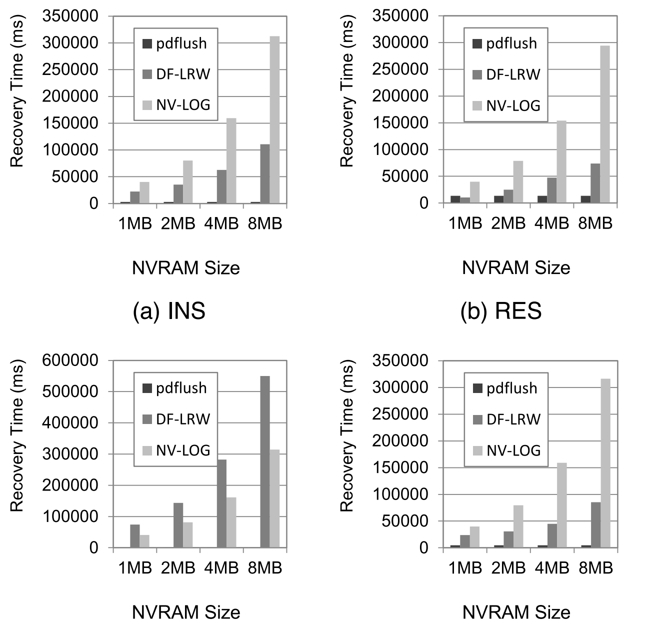
(c) 4MB non-volatile buffer cache.

(d) 8MB non-volatile buffer cache.

Fig. 10. Storage write traffic with the Linkbench workload.

when the non-volatile buffer cache size is larger than 4 MB, but there is almost no reduction under the cache size of 1 and 2 MB. Since plotting a graph needs to access a huge amount of data at the same time, Linkbench has a relatively large working set. In this workload, thrashing occurs if the non-volatile buffer cache is not large enough to hold the working set, failing to absorb storage writes. Though our scheme gives no benefit with respect to the write traffic in this case, it still has the effect of shortening the vulnerability window of data loss. That is, upon a system crash, pdflush loses the updated data after the last pdflush period (e.g., 30 seconds), whereas our scheme does not do so.

Considering the results in Figs. 7, 8, 9, and 10 and the workload characteristics, we conclude that a small



(a) INS (b) RES

(c) TPC-C (d) Linkbench

Fig. 11. Comparison of recovery time varying the NVRAM size.

STT-MRAM size of 1-4 MB is sufficient in general systems but more space is needed for specific memory-intensive applications like Linkbench. However, we do not believe that our cases can be generalized to all environments. In reality, the size of the non-volatile buffer cache should be configured based on the scale of target system's workloads as well as the relative DRAM size of the system. Our current conclusion is that only a small amount of first generation STT-MRAM products will be sufficient to eliminate pdflush regardless of their density and capacity, if our software technology is utilized.

4.2 Comparison of Recovery Time

We compare the system recovery time of the proposed DF-LRW scheme with that of a legacy buffer cache that uses pdflush and the NV-LOG scheme where non-volatile memory serves as a logging device. We assume that the system crash occurs at any timepoint and measure the data traffic between DRAM, NVRAM, and storage devices for the recovery. Then, we measure the time components of the memory copy and the data loading from storage in our experimental systems, which are 5 and 150 us for a 4 KB data access, respectively. Finally, we calculate the recovery time by multiplying the amount of data and the time component per unit. We repeat this 10 times and take an average of the results.

Fig. 11 shows the recovery time of the three schemes. As shown in the figure, the proposed scheme provides shorter recovery time than the NV-LOG scheme. Specifically, the reduction of the recovery time is 28.2 percent on average. The system recovery of DF-LRW includes loading old versions of blocks to the volatile buffer cache for each partially written fragment in the non-volatile buffer cache, and reflecting the fragments to the volatile buffer cache. In comparison, NV-LOG commits the logged data in the non-volatile memory to the storage file system during the recovery process. As the legacy buffer cache with pdflush performs journaling only for the metadata, it restores the file system

TABLE 2
Characteristics STT-MRAM Compared to Other Memory and Storage Technologies

	DRAM	STT-MRAM	PCM	NAND
Maturity	Product	Prototype	Product	Product
Read Latency	10-50 ns	10-50 ns	50-100 ns	25 us
Write Latency	10-50 ns	10-50 ns	100-500 ns	200 us
Erase Latency	-	-	-	200 ms
Energy (per bit access)	2 pJ	0.02 pJ	r: 20 pJ w: 100 pJ	10 nJ
Static Power	Yes	No	No	No
Endurance	10^{16}	10^{16}	10^7-10^8	10^5
Cell Size	6~8 F^2	> 6 F^2	5~10 F^2	4~5 F^2
MLC	No	4 bits/cell	4 bits/cell	4 bits/cell

consistency by committing the metadata log to the file system locations, losing the updates for regular data except for metadata between flush timepoints.

In the case of TPC-C, DF-LRW has the longest recovery time among the three schemes, which is 1.7 times longer than that of NV-LOG. Since the write operations of TPC-C are mostly of small sizes, it has a large number of partial fragments in the non-volatile buffer cache, which require the original blocks to be loaded from the storage before being reflected to DRAM. This results in a significant increase of the storage accesses during the system recovery. However, this feature is advantageous for most of the time when the system is working well as small fragments are space efficient and thus effective in reducing the storage writes. The performance of normal situations and the system recovery may have trade-off relations.

4.3 Implementation and Measurement

To assess the effectiveness of the proposed buffer cache further, we implemented a prototype of DF-LRW on Linux 2.6.38. Our scheme is compared with the original Linux with pdflush. We use EXT4 as the file system and the journaling option is set to the ordered-mode, which is the default option that journals only metadata. Our experimental platform consists of an Intel Core i3-2100 CPU running at 3.1 GHz and 4 GB of DDR2-800 memory.

Though our design assumes STT-MRAM as the non-volatile buffer cache, commercially available STT-MRAM products are limited for now, and thus we simply use a portion of DRAM as the non-volatile buffer cache. Note that the performance characteristics of STT-MRAM are similar to those of DRAM as shown in Table 2 [14]. We measure the performance with IOzone [17] and Filebench [18], representative storage benchmarks.

Fig. 12 shows the throughput of our buffer cache in comparison with that of Linux with pdflush in the Filebench benchmark. Filebench provides a series of predefined I/O workloads that emulate different types of real server systems. The workloads used in the experiments are varmail, proxy, webserver, and fileserver. We change the number of threads for each workload and measure the total throughput of all threads. As shown in the figure, our buffer cache performs better than existing Linux pdflush for all workloads. Specifically, the performance improvement is 24.6 percent on average, and varmail shows the largest

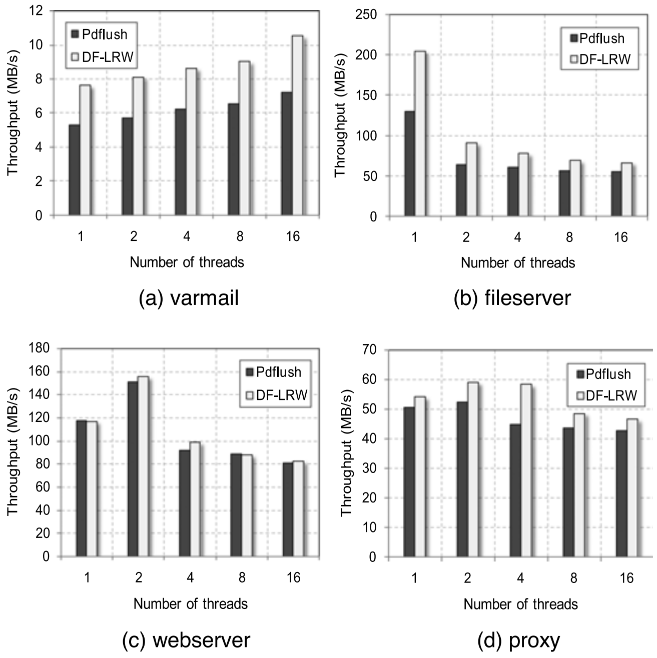


Fig. 12. Throughput of Filebench as the number of threads is varied.

improvement of 43.3 percent among the four workloads. The reason for such large improvements is that varmail contains a large number of write requests, thus incurring a large number of flush operations. Moreover, varmail issues read and write requests concurrently.

Though our scheme does not have any explicit policy to improve read performance, eliminating frequent storage writes frees the contention for hardware resources like the memory bus and DMAs, improving I/O performance in general. Even for read-intensive workloads like webserver and proxy, our buffer cache improves performance by 2.5 percent and 15.6 percent, respectively, due to this reason. For fileserver, which is write-intensive, the improvement by our scheme is relatively small compared to varmail. This is because most write requests in fileserver are large and sequential writes that lead to frequent cache replacement. As the number of threads increases, the throughput is also improved due to higher parallelism as shown in Fig. 12a. However, when the workload exceeds the maximum capacity of the system, the performance drops rapidly with an increased number of threads as shown in Fig. 12b.

Fig. 13 compares the throughput of our buffer cache and Linux pdflush under the IOzone benchmark. IOzone is a well-known micro benchmark that measures file I/O performance by generating particular types of operations in batches. It creates a single large file and performs a series of operations on that file. We measure the performance with four IOzone scenarios, initial write, random write, sequential write, and fwrite. Initial write creates a file and then writes data sequentially to that file. The other write scenarios do not include file creation procedures but involve writing to existing files with different access patterns. The random write scenario issues a series of write requests at a randomly selected offset of the file, whereas the sequential write performs write operations from the start of the file without changing the offset. The write requests are generated using the POSIX standard library in all scenarios. We

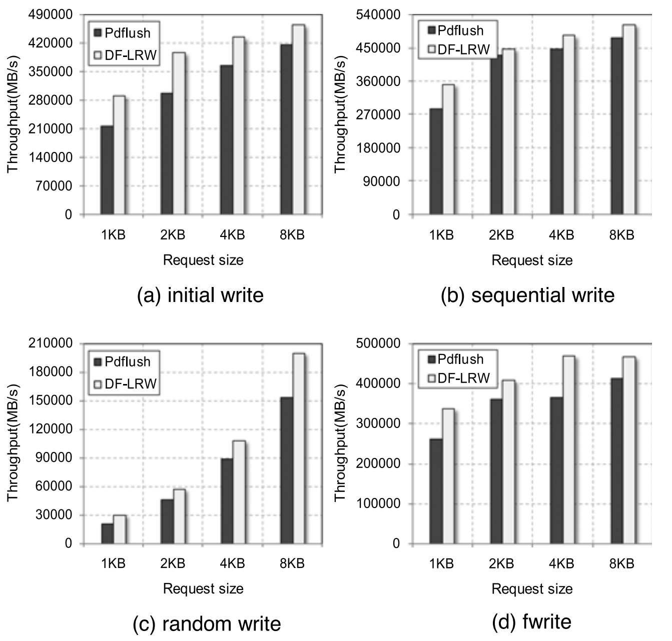


Fig. 13. Throughput of IOzone as the request size is varied.

vary the sizes of write requests from 1 to 8 KB. Since the total amount of request data in IOzone is identical in each run, the number of requests decreases as the size of each request increases from 1 to 8 KB. As shown in the figure, our buffer cache outperforms Linux pdflush in all the write scenarios for all request sizes. Specifically, the performance improvement by our buffer cache over Linux pdflush is 22.6 percent on average. This remarkable enhancement is achieved from the large reduction of writes with the space-efficient management of our non-volatile buffer cache.

When considering the request size, both Linux pdflush and our scheme perform better as the request size increases in all the scenarios. This is because a workload with a larger request size contains less write requests, reducing software overhead. Note that each I/O request invokes software stacks, and thus the overhead is proportional to the number of requests.

When we look at the different IOzone scenarios, shown in Figs. 13a, 13b, 13c, and 13d, we see that the results for random writes are very sensitive to the request size. This is because random writes are largely affected by the number of requests as random accesses incur large seek overhead to move the disk head to the requested location. The relative performance improvement of our scheme over Linux pdflush is also the largest in random writes because the overhead of disk seek is the largest in random writes. The throughput of initial writes is a bit less than sequential writes for both schemes, which can be attributed to the extra overheads involved in file creation.

5 Related Work

5.1 Buffer Caching Algorithms

To narrow the speed gap between main memory and storage, buffer caching algorithms have been studied extensively for decades. The LRU and LFU algorithms are representative algorithms that consider the recency and the frequency of block references, respectively.

There have been attempts to combine the advantages of LRU and LFU. O'Neil et al. present the LRU-*k* algorithm to address the problem of LRU that cannot consider the reference frequency of blocks [19]. LRU-*k* decides blocks to be replaced based on the time of the *k*th-to-last reference. A larger *k* can discriminate better between frequently and infrequently referenced blocks. Johnson and Shasha proposed a block replacement policy called 2Q [20]. This algorithm divides the LRU cache list into two queues to remove blocks referenced only once quickly from the cache, while maintain repeatedly referenced blocks for an extended period of time. Lee et al. proposed the least recently/frequently used (LRFU) algorithm that subsumes the LRU and LFU algorithms [21]. In LRFU, each cached block is associated with a combined recency and frequency (CRF) value that estimates the re-reference likelihood of the block in terms of both recency and frequency. All past references to a block during its residence in the cache are reflected in CRF and a reference's contribution decreases as time progresses.

Adaptive replacement cache (ARC) is another algorithm that adaptively considers the recency and frequency of references [22]. ARC maintains two LRU lists to capture the recency and frequency of refetences with separated lists, and adaptively adjusts their sizes according to the evolution of workloads.

Unified buffer management (UBM) detects reference patterns and allocates separate cache space to each detected pattern [23]. Specifically, UBM classifies referenced blocks into three patterns, sequential, looping, and other references, and then allocates cache space to each pattern based on their marginal gain. Low inter-reference recency set (LIRS) replacement uses the concept of inter reference recency to accurately estimate future block references [24]. LIRS divides blocks into two sets: High inter-reference recency (HIR) and low inter-reference recency (LIR) sets. LIRS gives higher caching priorities to the LIR set as it contains frequently accessed blocks.

Recently, as NAND flash memory has been widely adopted as the secondary storage of mobile systems, there have been extensive studies on buffer caching algorithms for NAND flash memory. Clean-first LRU (CFLRU) is a cache replacement algorithm for flash memory that considers the physical characteristics of NAND flash memory, in which reading and writing have different I/O costs [25]. CFLRU can accommodate the different eviction costs of a clean block, which can simply be discarded, and a dirty block, which should be written back to flash memory. CFLRU maintains a certain cache area called window and delays the eviction of dirty blocks in the window as long as a clean block is available for eviction. LRU with write sequence reordering (LRU-WSR) is another algorithm that favors dirty blocks [26]. Basically, LRU-WSR also manages blocks using the LRU list. Instead of setting the window area, LRU-WSR gives one more chance to a dirty block when it reaches the LRU position in the list. Though CFLRU and LRU-WSR delay the replacement of dirty blocks, their effect on reducing write traffic is limited as dirty blocks are written to storage by pdflush or journaling before eviction.

Flash-aware buffer management (FAB) was proposed as a buffer replacement algorithm in flash-based PMP systems [27]. PMP systems commonly have long sequential accesses for media data and some short accesses for metadata at the same time. One problem with this situation is that short write accesses cannot be buffered for a long time because they are pushed away by a large amount of sequential data. To cope with this problem, FAB manages buffered data from the same NAND flash block as a group, and replaces them together. Specifically, FAB replaces a group with the largest number of buffers first, which is usually large sequential data.

Block padding least recently used (BPLRU) is a write buffer management algorithm to improve the random write performance of flash storage [28]. Similar to FAB, BPLRU groups buffers from the same NAND flash block, and replaces them together. When a buffer is accessed by a write operation, buffers in the same group are moved together to the MRU position of the list. BPLRU selects buffers in the LRU position as a victim, and flushes all data in the group. BPLRU has a similar aspect to our idea in that it also groups a certain type of caching items and manages them together. However, the two schemes are fundamentally different. First, BPLRU is devised for the internal write buffer of flash storage and it is not affordable in the host side buffer cache layer. BPLRU replaces multiple cached items belonging to the same physical flash block together, but logical-to-physical mapping information is only accessible inside the device. Thus, the host buffer cache cannot figure out the cached items belonging to the same flash block under the current standard I/O interfaces. BPLRU and DF-LRW are also different from an algorithmic perspective. DF-LRW downsizes the caching unit to a fragment in order to improve space efficiency, whereas BPLRU uses the conventional caching unit of a logical block (or flash page). Instead, BPLRU extends the replacement unit to multiple logical blocks in order to improve the efficiency of garbage collection, whereas our scheme uses the conventional I/O unit of a single logical block.

Cold and largest cluster (CLC) is another write buffer replacement algorithm for NAND flash memory [41]. Similar to FAB and BPLRU, CLC manages buffered data from the same NAND flash block as a group. When replacement is needed, CLC selects a group with the largest number of buffers that have not been referenced recently as a victim.

Shi et al. proposed another write buffer replacement algorithm for flash memory called expectation-based LRU (ExLRU) [42]. ExLRU accurately maintains access history information in the write buffer based on a new cost model, and replaces data with the minimum write cost to be written to flash memory. Kim et al. proposed a new buffer cache replacement scheme called SpatialClock for mobile flash storage [43]. SpatialClock considers the spatial locality of writes to flash storage in order to reduce the overall I/O time. Wu et al. presented a hybrid page/block architecture for flash-based SSDs, and proposed an adaptive write buffer management scheme under this architecture called block-page adaptive cache (BPAC) [44]. BPAC considers both temporal and spatial locality of references in flash memory. To do this, BPAC adaptively partitions the SSD write cache to separately hold pages with high temporal locality and clusters of pages with low temporal but high spatial locality. Recently, Kim and Kim proposed a write buffer management scheme in solid-state drives, called QLRU [45]. QLRU exploits the future buffer reference patterns by using I/O commands information in native command queuing (NCQ) of SATA SSDs.

More recently, non-volatile buffer cache is used to improve system reliability. Lee et al. presented a design of a buffer cache that subsumes the functionality of caching and journaling [9]. Their scheme supports the in-place commit that avoids storage journaling, but still provides the same journaling effect by simply altering the state of the cached block to frozen. Our work is different from this work as we reduce the flush overhead incurred by regular data rather than journaling overhead for metadata. Furthermore, we use only a small amount of non-volatile buffer cache while their scheme adopts non-volatile memory for entire buffer cache.

5.2 Using Non-Volatile Memory in Memory and Storage Hierarchy

Recently, high-speed non-volatile memory technologies such as PCM and STT-MRAM have been catching interest, and they may be used as main memory in future computer systems. Specifically, non-volatile memory is byte-accessible and its access time is (optimistically) projected to be almost identical to that of DRAM, while consuming less energy and providing higher scalability than DRAM [4], [5], [6], [7].

Mogul et al. suggested an efficient memory management policy for a hybrid memory system consisting of both DRAM and PCM [29]. They proposed a page-attribute-aware memory allocation policy that tries to place read-only pages like code segments in PCM, while writable pages in DRAM, thereby reducing the number of PCM writes. In line with this research, Qureshi et al. proposed a memory architecture that uses a small amount of DRAM as the write buffer of PCM memory in order to prolong the lifetime of PCM and hide the long write latency of PCM [4]. Lee et al. also suggested a PCM main memory architecture and attempted to improve the write performance between the last level cache and PCM memory [5]. They proposed two policies, buffer reorganization and partial writes that track data modifications and write only modified cache lines or words to the PCM array. Lee et al. proposed the CLOCK-DWF algorithm for the hybrid memory architecture consisting of both DRAM and PCM [6]. They allocate read-intensive pages to PCM and write-intensive pages to DRAM based on memory reference characterization. Dhiman et al. presented a hybrid PCM and DRAM memory architecture and try to balance the write count of PCM by moving data located at a PCM page to a DRAM page if the write count of the PCM page becomes large [30]. Zhou et al. presented DRAM cache partitioning and replacement algorithms for PCM main memory [31]. Their algorithms aim to reduce the cache miss ratio as well as write-backs from DRAM cache. They also considered the balancing of PCM write queues in the design of replacement algorithms.

There is another category of research that focused on file system design for non-volatile memory. As the capacity of these non-volatile memories was initially small, only a limited portion of the entire file system image could be placed in the non-volatile memory partitions. For example, Pramfs is designed to store frequently accessed or important data in non-volatile memory to support fast rebooting and resist crashes [32]. MRAMFS [33] and the NEB file system [34] also maintain a certain part of the file system, such as the metadata on non-volatile memory. They aim to improve the space efficiency of non-volatile memory-based storage. Specifically, MRAMFS saves space by compressing metadata, while the NEB file system does this by extent-based file management.

As the density of non-volatile memory gets improved, recent studies have focused on the design of file systems that retain the entire file system image on non-volatile memory. Baek et al. implemented a software layer to support both file objects and memory objects together in the unified memory system in which non-volatile memory serves as both main memory and storage [35]. Condit et al. redesigned a copy-on-write file system, called BPFS, for byte-addressable storage [36]. BPFS performs in-place write, when the updated data size is smaller than an atomic operation unit. This can significantly reduce the out-of-place-update overhead of copy-on-write. Wu and Reddy proposed a file system for byte-addressable storage [37]. Assuming that byte-addressable storage resides on the memory bus and can be accessed directly from CPU, they proposed a file system that accesses files through the same address space of virtual memory systems. Lee et al. suggested a write-efficient journaling file system that reduces writes for journaling in future non-volatile memory based storage systems [38]. The proposed file system handles incoming journaling operations efficiently based on partial writability and random accessibility of memory-based devices.

Ousterhout et al. proposed a RAMCloud architecture that is a scalable in-memory data store system by aggregating the main memory of thousands of servers [40]. RAMCloud logs only the modified part rather than the entire object in backup servers to avoid data loss upon a crash of a single server. This mechanism is conceptually similar to the delta-write technique in our work, but we manage non-volatile memory as a write cache thus unifying multiple updates for the same data as a single caching item, whereas RAMCloud just appends updates in memory. In addition, RAMCloud does not consider the use of non-volatile memory, and thus its target system architecture is different to that of our work.

6 CONCLUSION

Periodic flush operations from the buffer cache improve the file system reliability but degrade the caching efficiency greatly. This paper showed that the periodic flush operations account for 30-70 percent of total write traffic to storage. To eliminate this inefficiency, we proposed, implemented, and evaluated a new buffer cache architecture that uses only a small amount of non-volatile memory and stores modifications to the non-volatile buffer cache. This buffer cache architecture removes almost all storage accesses due to periodic flush operations without any loss of reliability. It also improves the buffer cache performance via space-efficient cache management such as delta-write and fragment-grouping with only a small amount of non-volatile memory. Our simulation and measurement results have shown that the proposed scheme reduces the number of write I/Os by 44.3 percent and also improves the throughput by 23.6 percent on average for the specific workloads we tested.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No.2011-0028825). Hyokyung Bahn is the corresponding author of this paper.

REFERENCES

[1] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 1–16.
[2] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. USENIX Conf. File Storage Technol.*, 2012, pp. 209–222.
[3] M. Rubin, "File systems in the cloud," presented at the Linux Foundation Collaboration Summit, San Francisco, CA, USA, 2011.
[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. Int. Symp. Comput. Archit.*, 2009, pp. 24–33.
[5] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase change memory architecture and the quest for scalability," *Commun. ACM*, vol. 53, no. 7, pp. 99–106, 2010.
[6] S. Lee, H. Bahn, and S. H. Noh, "Characterizing memory write references for efficient management of hybrid PCM and DRAM memory," in *Proc. IEEE Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst.*, 2011, pp. 168–175.
[7] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proc. Int. Symp. Comput. Archit.*, 2009, pp. 14–23.
[8] A. Proctor. (2012). Non-volatile memory & its use in enterprise applications [Online]. Avaialble: http://www.vikingtechnology.com/uploads/nv_whitepaper.pdf
[9] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. USENIX Conf. File Storage Technol.*, 2013, pp. 73–80.
[10] Y. Kim, I. H. Doh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Design and implementation of transactional write buffer cache with storage class memory," *J. Korean Inst. Info. Sci. Eng.*, vol. 16, no. 2, pp. 247–251, 2010.
[11] D. Phillips, "Zumastor linux storage server," in *Proc. Linux Symp.*, 2007, pp. 135–143.
[12] R. Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *Proc. IEEE Int. Conf. Data Eng.*, 2011, pp. 11–16.
[13] Now, The NOW Trace Collection Project. (1998). [Online]. Available: http://tracehost.cs.berkeley.edu/traces.html
[14] X. Guo, E. Ipek, and T. Soyata, "Resistive computation: Avoiding the power wall with low-leakage, STT-MRAM based computing," in *Proc. Int. Symp. Comput. Archit.*, 2010, pp. 371–382.
[15] U. Vahallia, *UNIX Internals: The New Frontiers*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1995.
[16] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proc. USENIX Annu. Tech. Conf.*, 2000, pp. 41–54.
[17] W. Norcutt, *The IOzone Filesystem Benchmark*. (2006). [Online]. Available: http://www.iozone.org/
[18] Filebench. (2014). [Online]. Available: http://www.solarisinternals.com/wiki/index.php/FileBench
[19] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1993, pp. 297–306.
[20] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. Int. Conf. Very Large Data Bases*, 1994, pp. 439–450.
[21] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Comput.*, vol. 50, no. 12, pp. 1352–1361, Dec. 2001.
[22] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. USENIX Conf. File Storage Technol.*, 2003, pp. 115–130.

[23] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references," in *Proc. USENIX Symp. Oper. Syst. Des. Implement.*, 2000, pp. 119–134.
[24] S. Jiang and X. Zhang, "Making LRU friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance," *IEEE Trans. Comput.*, vol. 5, no. 8, pp. 939–952, Aug. 2005.
[25] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: Replacement algorithm for flash memory," in *Proc. Int. Conf. Compilers, Archit., Synthes. Embed. Syst.*, 2006, pp. 234–241.
[26] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR: Integration of LRU and writes sequence reordering for flash memory," *IEEE Trans. Consum. Electron.*, vol. 54, no. 3, pp. 1215–1223, Aug. 2008.
[27] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee, "FAB: Flash-aware buffer management policy for portable media players," *IEEE Trans. Consum. Electron.*, vol. 52, no. 2, pp. 485–493, May 2006.
[28] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in *Proc. USENIX Conf. File Storage Technol.*, 2008, pp. 239–252.
[29] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for NVM+DRAM hybrid main memory," in *Proc. USENIX Workshop Hot Topics Oper. Syst.*, 2009, p. 14.
[30] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *Proc. ACM/IEEE Design Autom. Conf.*, 2009, pp. 664–669.
[31] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mosse, "Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems," *ACM Trans. Archit. Code Optimiz.*, vol. 8, no. 4, article 53, 2012.
[32] Pramfs, (2013). [Online]. Available: http://pramfs.sourceforge.net
[33] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt, "MRAMFS: A compressing file system for non-volatile RAM," in *Proc. IEEE Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst.*, 2004, pp. 596–603.
[34] S. Baek, C. Hyun, J. Choi, D. Lee, and S. H. Noh, "Design and analysis of a space conscious nonvolatile-RAM file system," in *Proc. IEEE Region 10 Conf.*, 2006, pp. 1–4.
[35] S. Baek, J. Choi, D. Lee, and S. H. Noh, "Energy-efficient and high-performance software architecture for storage class memory," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 3, article 81, 2013.
[36] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM Symp. Oper. Syst. Principles*, 2009, pp. 133–146.
[37] X. Wu and A. L. N. Reddy, "SCMFS: A file system for storage class memory," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 1–11.
[38] E. Lee, S. Yoo, J. Jang, and H. Bahn, "Shortcut-JFS: A write efficient journaling file system for phase change memory," in *Proc. IEEE Int. Conf. Massive Data Storage*, 2012, pp. 1–6.
[39] E. Lee, J. Jang, and H. Bahn, "DTFS: Exploiting the similarity of data versions to design a write-efficient file system in phase-change memory," in *Proc. ACM Symp. Appl. Comput.*, 2014, pp. 1535–1540.
[40] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Oper. Syst. Review*, vol. 43, no. 4, pp. 92–105, 2010.
[41] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices," *IEEE Trans. Consum. Electron.*, vol. 58, no. 6, pp. 744–758, 2009.
[42] L. Shi, J. Li, C. Jason Xue, C. Yang, and X. Zhou, "ExLRU: A unified write buffer cache management for flash memory," in *Proc. ACM Int. Conf. Embed. Softw.*, 2011, pp. 339–348.
[43] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?" in *Proc. ACM SIGMETRICS Conf.*, 2012, pp. 235–246.
[44] G. Wu, X. He, and B. Eckart, "An adaptive write buffer management scheme for flash-based SSDs," *ACM Trans. Storage*, vol. 8, no. 1, article 1, 2012.
[45] S. Kim and T. Kim, "QLRU: NCQ-aware write buffer management algorithm for SSDs," *Electron. Lett.*, vol. 49, no. 17, pp. 1079–1081, 2013.

Eunji Lee received the PhD degree in computer engineering from Seoul National University in 2012. She was a visiting scholar in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, and a senior engineer at the Memory Division of Samsung Electronics, Co., Ltd. She is currently an assistant professor in the Department of Software, Chungbuk National University, Cheongju, Korea. Her research interests include operating systems, embedded systems, real-time systems, and storage systems. She has published more than 40 papers in leading conferences and journals in these fields, including the *IEEE Transactions on Computers*, the *IEEE Transactions on Knowledge and Data Engineering*, and *ACM Transactions on Storage*. She also received the Best Paper Awards at the USENIX Conference on File and Storage Technologies in 2013.

Hyojung Kang received the BS degree in computer science and engineering from Ewha University, Republic of Korea, in 2009. She is currently working toward the MS degree in the Department of Computer Science and Engineering, Ewha University, Seoul, Republic of Korea. Her research interests include operating systems, system optimization, storage system, file systems, embedded software, and low-power systems.

Hyokyung Bahn received the BS, MS, and PhD degrees in computer science and engineering from Seoul National University, in 1997, 1999, and 2002, respectively. He is currently a professor of computer science and engineering at Ewha Womans University, Seoul, Republic of Korea. 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.

Kang G. Shin (S'75-M'78-SM'83-F'92) is the Kevin & Nancy O'Connor professor of computer science in the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor. His current research focuses on QoS-sensitive computing and networking as well as on embedded real-time and cyber-physical systems. He has supervised the completion of 74 PhDs, and authored/coauthored more than 800 technical articles (more than 300 of these are in archival journals), one textbook, and more than 20 patents or invention disclosures, and received numerous best paper awards, including the Best Paper Awards from the 2011 ACM International Conference on Mobile Computing and Networking, the 2011 IEEE International Conference on Autonomic Computing, the 2010 and 2000 USENIX Annual Technical Conferences, as well as the 2003 IEEE Communications Society William R. Bennett Prize Paper Award, and the 1987 Outstanding IEEE Transactions of Automatic Control Paper Award. He has also received several institutional awards, including the Research Excellence Award in 1989, Outstanding Achievement Award in 1999, Distinguished Faculty Achievement Award in 2001, and Stephen Attwood Award in 2004 from The University of Michigan (the highest honor bestowed to Michigan Engineering faculty); a Distinguished Alumni Award of the College of Engineering, Seoul National University in 2002; 2003 IEEE RTC Technical Achievement Award; and 2006 Ho-Am Prize in Engineering (the highest honor bestowed to Korean-origin engineers). He has chaired several major conferences, including 2009 ACM MobiCom, 2008 IEEE SECON, 2005 ACM/USENIX MobiSys, 2000 IEEE RTAS, and 1987 IEEE RTSS. He is the fellow of both the IEEE and ACM, and served on editorial boards, including *IEEE TPDS* and *ACM TECS*. He has also served or is serving on numerous government committees, such as the US National Science Foundation (NSF) Cyber-Physical Systems Executive Committee and the Korean Government R&D Strategy Advisory Committee. He has also cofounded a couple of startups.