Unified DRAM and NVM Hybrid Buffer Cache Architecture for Reducing Journaling Overhead

Zhiyong Zhang   Lei Ju   Zhiping Jia
School of Computer Science and Technology
Shandong University, Jinan, China
Email: jzp@sdu.edu.cn

***Abstract*—Journaling techniques play an important role in addressing the reliability issue of filesystems caused by the volatile DRAM-based buffer cache. However, journaling techniques introduce a large number of extra storage writes, which greatly degrades the performance of the filesystem [10]. Emerging Non-Volatile Memory (NVM) technologies bring a new perspective of solving the write amplification issue caused by journaling. By adopting NVM as the buffer cache, the committed data can be maintained in NVM before being written back to the storage, thus eliminating the journaling overhead. However, simply replacing DRAM with NVM as the buffer cache suffers from the limited lifetime and relative slow writes of NVM.**

In this paper, we present a hybrid buffer cache architecture by combing NVM with DRAM to reduce the journaling overhead and overcome the constraints of NVM. In order to better utilize this novel architecture, we first propose a Journaling-Aware Page Management (JAPM) policy. JAPM puts infrequently updated data in NVM to reduce the journaling overhead and frequently updated data in DRAM to improve the write performance and lifetime of the hybrid buffer cache. In addition, data in one transaction may be dispersed in NVM and DRAM simultaneously and different committing policies are required for different storing media, NVM or DRAM. In order to guarantee the atomicity of the transactional execution in the hybrid cache architecture, a Partial In-Place Commit (PIPC) journaling scheme is proposed to coordinate the different committing patterns. We implement the proposed techniques on Linux 3.14.52 and measure the performance with representative I/O-intensive benchmarks. The experimental results show that our scheme effectively improves the I/O performance compared with the ext4 filesystem and prolongs the lifetime of the hybrid buffer cache compared with the Union of Buffer cache and Journaling (UBJ) scheme [10].

I. Introduction

Traditional filesystems, such as the ext2 filesystem, may be crashed with the data inconsistency issue. This issue is caused by the volatile DRAM-based buffer cache adopted in the filesystems, since data in the DRAM buffer cache may be partially committed to the non-volatile disk when the system is crashed. To relieve this issue, modern filesystems, such as the ext4 filesystem, exploit the journaling technique to firstly write data to a journal area and then periodically write it to its home location. By using journaling techniques, the partially committed data can be recovered from the journal area and the reliability of filesystems is enhanced. However, journaling techniques seriously degrade the performance of filesystems because of its extra disk writes. Previous work [10] has shown that the write traffic with journaling is about 2.7 times more than that without journaling on average. Thus, reducing the journal traffic becomes an important way to improve the filesystem performance.

Emerging non-volatile memory (NVM) technologies, such as STT-RAM (Spin-Transfer Torque RAM) and PCM (Phase-change Memory), provide a new opportunity to reduce the journaling overhead while guaranteeing the high reliability of modern filesystems. These non-volatile memories can support byte-addressable access at DRAM-like latencies and retain non-volatile characteristics [8], [19]. Since NVM is non-volatile, the modified data in NVM can act as its own journal before being committed to the persistent storage. Therefore, introducing NVM to the buffer cache can effectively reduce the write traffic caused by journaling. However, NVM technologies have the constraints of limited write endurance and relative long write latency compared with DRAM. Simply using NVM as the buffer cache may degrade the write performance of filesystems and make NVM to be worn out quickly.

Eunji Lee et al. have proposed the Union of Buffer cache and Journaling (UBJ) scheme to reduce the write amplification issue caused by journaling [10]. Although UBJ significantly reduces the journaling overhead, it completely replaces NVM as the buffer cache without considering the constraints of NVM. Considering the constraints of NVM technologies instead of the journaling overhead, some the state-of-the-art studies [2], [3], [7], [11], [12], [14]–[16] have focused on overcoming the constraints of NVM technologies. All these studies achieve better results in improving the characteristics of NVMs, but none of them employs NVM as the buffer cache in filesystems. The previous work [17] and [4] have introduced NVM to the buffer cache. But they only aim to propose a page caching algorithm to enhance the average page cache performance and reduce the endurance problem of NVM without taking the journaling overhead into account.

In this paper, we present a NVM and DRAM hybrid buffer cache architecture that unifies the functionality of caching and journaling in filesystems. The architecture aims to reduce the journaling overhead and overcome the constraints of NVM. In order to better utilize this novel architecture, a Journaling-Aware Page Management (JAPM) policy is proposed. JAPM puts frequently updated data in DRAM to improve the write performance and lifetime of the hybrid buffer cache. Meanwhile, JAPM accelerates the migration of the uncommitted dirty data from DRAM to NVM to reduce the journaling overhead.

In addition, the hybrid buffer cache architecture makes the traditional transaction-based recovery protocols complex in keeping the consistency of filesystems. When a commit operation occurs, the journal of the transaction in NVM is committed in-place. But for the DRAM buffer cache, the journal is committed to the traditional journaling area in the secondary storage. The hybrid buffer cache architecture enables the hybrid transactions (part in NVM and part in DARM). In order to guarantee the atomicity of the transactional execution, we propose a Partial In-Place Commit (PIPC) journaling scheme to support different committing patterns.

We have implemented a prototype of our hybrid buffer cache with the proposed journaling-aware page management policy and the partial in-place commit journaling scheme on Linux 3.14.52. Experiments have been conducted with various representative I/O-intensive benchmarks. The experimental results show the proposed techniques can effectively improve the I/O performance compared with the ext4 filesystem and prolong the lifetime of the hybrid buffer cache compared with the Union of Buffer cache and Journaling (UBJ) scheme [10].

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

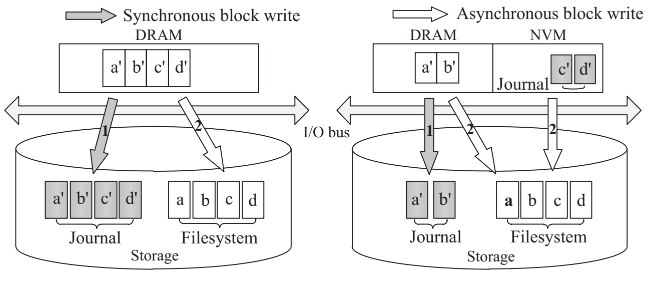
- We propose a journaling-aware page management policy for the hybrid buffer cache architecture to reduce the journaling overhead and overcome the constraints of NVM.
- We put forward a partial in-place commit journaling scheme to guarantee the atomicity semantics of the transactional execution.

The rest of this paper is organized as follows. Section II analyzes the problems to be addressed in this paper. Section III presents the journaling-aware page management policy. Section IV describes the partial in-place commit scheme for the hybrid buffer cache. Experimental results and analysis are presented in Section V and finally, this paper is concluded in Section VI.

II. Problem Analysis

The hybrid buffer cache employs DRAM and NVM to reduce the journaling overhead and overcome the constraints of NVM. We aim to enormously remove storage accesses caused by journaling to improve the performance of filesystems without any loss of reliability. Meanwhile, we maximize the lifetime of the hybrid buffer cache. To this end, there are two critical issues to be addressed.

Firstly, writing to NVM eliminates the journaling overhead, but the constraints of NVM degrade the write performance and lifetime of the hybrid buffer cache. While writing to DRAM will not suffer from these constraints, but it causes extra journaling overhead. Therefore, in order to better utilize the hybrid architecture, the management policy should be optimized for two requirements. First, the DRAM portion should absorb as many writes as possible, thus enhancing the write performance and lifetime of the hybrid buffer cache.



(a) Original DRAM-based buffer cache (b) Proposed hybrid buffer cache

Fig. 1: The journal mechanism of DRAM-based buffer cache and the proposed hybrid buffer cache.

Meanwhile, the uncommitted dirty data should be migrated from DRAM to NVM to reduce the journaling overhead.

Secondly, the traditional journaling filesystem periodically creates transactions, which contain all the dirty data in the buffer cache, and commits transactions to the journal area. After that, the updated data is written to the filesystem, as is shown in Fig. 1 (a). However, in the hybrid buffer cache, data in one transaction may be dispersed in NVM and DRAM simultaneously. Upon receiving a commit operation, the journal of the transaction in NVM is committed in-place without flushing to the journal area while which in DRAM has to be committed to the traditional journaling area in the secondary storage (Fig. 1 (b)). In order to guarantee the atomicity of the transactional execution in the hybrid buffer cache architecture, a partial in-place commit journaling scheme is needed to coordinate the different committing patterns.

III. Journaling-aware Page Management Policy

A. Policy Overview

Fig. 2 shows the flow chart of our proposed Journaling-Aware Page Management (JAPM) policy for the hybrid buffer cache. When a new page enters into the buffer cache, since we don't know its access pattern, we place it to DRAM to prevent excessive writes on NVM. The pages in DRAM are divided into two categories, *write-hot* or *write-cold*, from the perspective of write recency. When a new page needs to enter into the buffer cache and there is no free space in DRAM buffer, the page management policy chooses a write-cold page and migrates it to NVM to make space for it.

It is worth noting that the proposed hybrid buffer cache subsumes the functionality of caching and journaling. Some pages in the NVM buffer cache are journal pages, whose states are marked as *frozen*. The frozen pages are write protected since they record the previous version of some transactions. Therefore, writing to a frozen page leads to the page being copied to a new location. Then, the updated data is written to the copy. Besides, if a page in the NVM buffer cache has been written frequently within a short period of time, we migrate it to DRAM with the consideration of the lifetime and write performance of the hybrid buffer cache. To this end, a write-burst predictor is integrated into our page management policy.

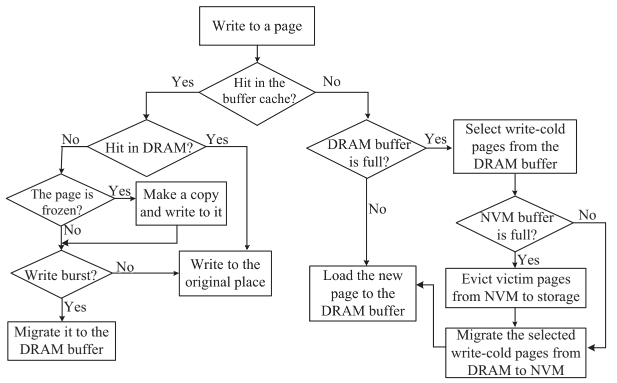

Write to a page
Hit in the buffer cache?
Yes
No
Hit in DRAM?
No
Yes
DRAM buffer is full?
Yes
No
Select write-cold pages from the DRAM buffer
The page is frozen?
Yes
No
Make a copy and write to it
NVM buffer is full?
Yes
No
Write burst?
No
Yes
Write to the original place
Load the new page to the DRAM buffer
Evict victim pages from NVM to storage
Migrate it to the DRAM buffer
Migrate the selected write-cold pages from DRAM to NVM

Fig. 2: Flow chart of the page management policy for the hybrid buffer cache.

If a write request in NVM is predicted to be a write burst access, the page will be migrated to DRAM.

If a new page needs to be loaded into the buffer cache and the hybrid buffer cache is full, we evict a victim page from NVM to storage for the reason that the data in NVM is infrequently updated.

B. Data Migration for Journaling

In order to enormously reduce the write traffic caused by journaling, we accelerate the migration of the uncommitted dirty data from DRAM to NVM. A simple solution is to migrate multiple dirty pages when page replacement occurs. However, it is not appropriate to set a fixed threshold since different applications have different access patterns. Thus, we propose the concept of *migration window* to adaptively adjust the migration number. When a page replacement occurs in DRAM, we migrate multiple victim pages from DRAM to NVM. All the victim pages are recorded in the migration window. If only a small amount of these victim pages in the window are written before the next replacement, which means these pages could be placed in NVM to reduce the journaling overhead while not adding excessive writes to NVM, we double the window size. Otherwise, the size will be halved. The window size in our policy ranges from 2 to 128.

To take the LRU algorithm as an example, we add a doubly linked list *dirty list* to link all the dirty pages of the DRAM buffer cache. When replacement occurs, if the least recently used page is clean, we just migrate it to NVM. Otherwise, we migrate m (m is the current size of the migration window) dirty pages continuously from the *dirty list* to NVM.

C. Implementation

We present the general design of the journaling-aware page management policy for the hybrid buffer cache. The proposed design rules can be integrated into a variety of popular page management algorithms. Considering the design requirements mentioned in Section II, JAPM must ensure most writes occur on DRAM to improve the write performance and the lifetime of the hybrid buffer cache, and the infrequently updated data is migrated to NVM to reduce the journaling overhead. Here, we describe a possible implementation of JAPM policy.

We propose the Low Inter-Write Recency (LIWR) policy, which is a modified LIRS algorithm, to divide the data into two categories: write-hot data and write-cold data. LIRS is a well-known cache replacement algorithm, which puts data into HIR (High Inter-reference Recency) set and LIR (Low Inter-reference Recency) set [5]. Since LIRS is designed for the uniform memory media without considering the constraints of NVM, it has to be modified to adapt to the hybrid buffer cache. In our design, we evaluate the access pattern of a page only exploiting the write history since the write temporal locality based on the write history alone and that based on both read/write histories show similar results [11].

Different from LIRS, LIWR plicy divides the DRAM buffer cache into two sets: HIW (High Inter-Write) set and LIW (Low Inter-Write) set. LIW set keeps the pages that are written frequently within a period of time, i.e. the write-hot pages, while HIW set stores the write-cold pages. Like LIRS, LIWR adopts the similar data structures stack S and list Q to manage the HIW and LIW set. Stack S is used to maintain the write-hot pages and the potential write-hot pages. List Q is used to link all the write-cold pages.

Since the pages in HIW set are write-cold pages, when page replacement occurs, we choose victim pages from list Q and move them to NVM. Considering the requirement of quick migration of the uncommitted dirty pages to reduce the journaling overhead, we add a doubly linked dirty list to link all the dirty write-cold pages. At the time of page replacement, LIWR executes page replacement according to the rules described in Section III-B. Since the replaced victim pages are write-cold pages, the quick migration rule reduces the journaling overhead while guaranteeing less writes on NVM. The other rules of LIWR are similar to LIRS, which are not detailed due to space limitations.

In order to further reduce the write accesses on NVM and correct the unreasonable judgments of write-cold pages, a write-burst predictor is designed to monitor the pages in NVM and migrate the write-burst pages to DRAM in real-time. Here, we use a variant of CLOCK algorithm to manage the NVM buffer cache. The modified CLOCK algorithm with write-burst predictor is called Pre-CLOCK.

Since CLOCK algorithm cannot predict the future write behavior, several extra fields, *write_bit*, *rotation_count* and *write_count*, are associated to each page. The *write_bit* records whether the page has been accessed recently and *write_count* is used to record the write frequency. When a write request hits in NVM and the corresponding *write_count* exceeds the preset write-burst threshold, this is a write-burst request and the page is migrated to DRAM. Different from CLOCK, Pre-CLOCK does not evict the potential write-burst pages (with the *write_count* not less than a preset ratio of the write-burst threshold) even their *write_bit*s are 0. The *rotation_count* keeps track of how many times the page was overlooked. If the *rotation_count* reaches the preset expiration threshold, which means this page has not been written for a long time, it is evicted from NVM.

IV. Partial in-place Commit Journaling Scheme

A. Journaling in the Hybrid Buffer Cache

The journaling scheme in NVM is called In-Place Commit. We reference the idea of *frozen blocks* proposed in [10], which only changes the state of updated cache blocks to frozen right at where it is currently located when a commit operation is issued. When read requests hit in a frozen block, if the frozen block is up-to-date, it can still be used as a cache block. On the other hand, frozen blocks are write protected since they record the previous version of some transactions, that is, writing to a frozen block leads to a copy is made to a new location and the updated data is written to the copy. Accordingly, the copy becomes up-to-date, while the original frozen block becomes out-of-date. This guarantees the journal blocks remain unchanged before the corresponding updates are written back to the filesystem.

Furthermore, the in-place commit generates additional benefits. When a block is committed multiple times, traditional DRAM-based journaling mechanism writes the block to the journal area each time a commit occurs, while the in-place commit scheme only commits the last version, since the old-commit blocks are maintained in the buffer cache and can be kept track of very easily (e.g. using hash table). This feature makes a further decline in write traffic.

As a result, there are three kinds of transactions in the hybrid buffer cache: transactions in DRAM, transactions in NVM and transactions part in DRAM and part in NVM. The former two cases will not cause confusions since the transactions are committed in their respective committing patterns. But for the third situation, since data in different media is committed in different ways, we need design a new journaling mechanism to coordinate the different committing patterns and guarantee the atomicity of the transactional execution.

B. Journaling for Hybrid Transactions

A whole journal is composed of *journal_superblock*, *descriptor_block*, *data_block*, *commit_block* and *revoke_block* (not necessary for all transactions). Similar to the superblock in filesystem, *journal_superblock* manages the whole journal space. *Descriptor_block* records the correspondences between journal blocks and filesystem blocks. *Commit_block* identifies whether a transaction is committed successfully. In our Partial In-Place Commit (PIPC) journaling scheme, we use the same journal organization for DRAM and NVM. The journal of a hybrid transaction is shown in Fig. 3. In this figure, an extra *hybrid_flag* field in *commit_block* indicates the transaction is a hybrid transaction with sequence 276. The transaction includes three data blocks recording the updates of filesystem blocks 9,12 and 17. Where, the NVM part logs the changes of block 12 and block 17, while the DRAM part records the update of block 9, which will be committed to the journal area.

1) Transaction Commit: Our PIPC journaling scheme periodically commits transactions by committing the updates to the journal area and/or changing the normal blocks to frozen blocks. This is done by keeping transactions in multiple

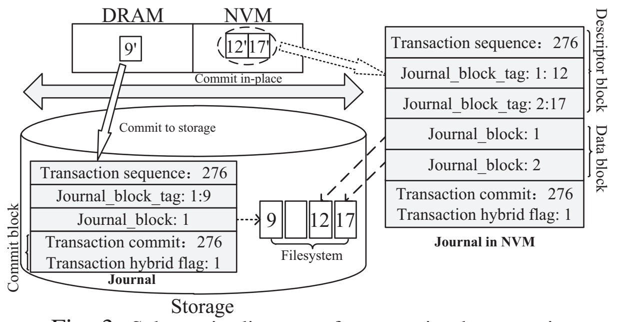


Fig. 3: Schematic diagram of a committed transaction.

lists. From the view of operating system, in the buffer cache, transactions can be divided into three types by the states of transactions: running transaction, committing transaction and checkpoint transaction. The running transaction maintains the uncommitted dirty blocks after the previous commit. Upon receiving a commit operation, the running transaction is converted to a committing transaction. When all dirty blocks are committed to journal area or converted to frozen, the running transaction becomes a checkpoint transaction. It is worth noting that there are two checkpoint transaction lists in the hybrid buffer cache, one in DRAM and one in NVM. For a hybrid transaction, different committing policies are required for different storing media, NVM or DRAM. Both parts are committed with the *hybrid_flag*s set to 1 and the same sequence to guarantee the atomicity of the transactional execution.

2) Checkpointing: We cannot reclaim the journal space until all the committed data has been rewritten to the filesystem. Flushing all the old buffers to reclaim the occupied space is called checkpointing. Checkpointing is a transaction-based operation. It will not complete until all the dirty blocks of a transaction are reflected on the permanent filesystem. After that, the relevant journal blocks in journal space are reclaimed and the frozen blocks in NVM are converted to normal blocks. Then, the transaction is removed from the corresponding checkpoint list.

3) Crash Recovery: When system crash occurs after unexpected software or hardware failures, the recovery mechanism is activated. Upon remounting the filesystem, PIPC scheme executes the recovery as follows.

At the time of the system crash, some buffer blocks in NVM may still reside in the running transaction list and the committing transaction list. Since the recovery protocol is transaction-based, the transaction in running transaction list with uncommitted data is simply ignored. Similarly, the transaction in committing transaction list is also discarded for it's not completely committed. The system is recovered according to the committed journal, which resides in the NVM checkpoint list and the journal area of storage.

PIPC goes through the journal area in storage and the checkpoint list in NVM to write the committed data successively back to the filesystem. If the *hybrid_flag* of a journal equals to 1, PIPC tries to find the other part journal of the transaction with the same transaction sequence. Once PIPC

TABLE I: Characteristics of different workloads.

Workloads	Avg. file size	# of files	Avg. req. size	% of read
varmail	16K	1000	4K	50.00
webproxy	16K	10000	4K	83.33
fileserver	128K	10000	64K	33.33
webserver	16K	1000	16K	90.91

finds the other part, it restores the updates of the two parts as a whole transaction. Otherwise, it is ignored as it is an incomplete commit. After all the journal records are rewritten to the filesystem, the filesystem will be in a consistent state again.

V. Experimental Evaluation

A. Experimental Setup

We integrated our journaling-aware page management (JAP-M) policy and partial in-place commit (PIPC) journaling scheme on Linux 3.14.52. We compared our scheme with the ext4 filesystem and UBJ proposed in [10]. UBJ adopts NVM as the whole buffer cache which represents the performance upper bound of our scheme since all the journal can be maintained in NVM. We set the journaling option of ext4 to *journal-mode*, which logs both data and metadata, to provide the same consistency semantics as UBJ and our scheme.

The hardware platform contains an Intel Core i5-3470 CPU running at 3.2GHz and 4GB DDR2 memory. Since NVM products are not yet widely available on the market, we use a portion of DRAM as NVM. There are a wide variety of projections for NVM's performance, and the specific design of the memory system can have a great impact on performance. So, we focus on the most important aspect of performance: slow writes. According to [1], [18], to account for NVM's slower writes relative to DRAM, we insert 150ns of extra latency to pull the timestamp counter (tsc). We evaluate these schemes with three IO-intensive benchmarks: Filebench [13], IOzone [6] and Postmark [9].

B. Experimental Results

1) Performance Evaluation: Filebench is a filesystem benchmark which uses loadable workload personalities to allow easy emulation of complex applications. As shown in Table I, we use four applications to measure the throughput of the three schemes. We run the four workloads 30 minutes under different schemes. As shown in Fig. 4(a), the throughput of our journaling scheme is better than ext4 by 72.75%, while with only 9% performance gap compared with UBJ, on average. Varmail generates a large number of writes, incurring frequent commit operations. However, since the small memory footprint, the varmail workload does not cause frequent check-pointing. Hence, the introduce of NVM significantly improves the performance in this situation. But frequent commits affect the performance benefits of our scheme since the updates in the DRAM buffer cache have to be committed to the journal area, which leads to largest performance gap, about 17%, with UBJ. But for fileserver, the result is reversed. Even for read-intensive workloads like webproxy and webserver, our scheme and UBJ improve the performance due to the reduction of journaling

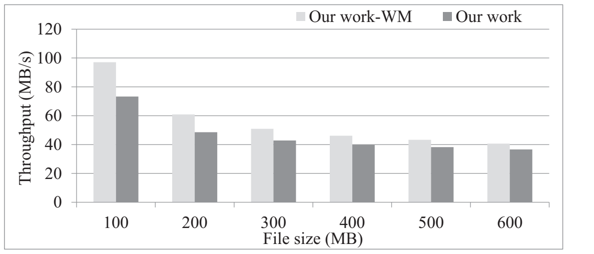


Fig. 5: Performance evaluation without migration overhead.

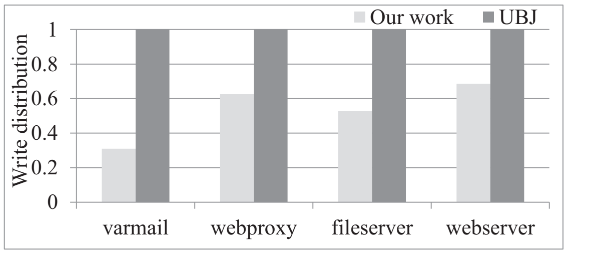


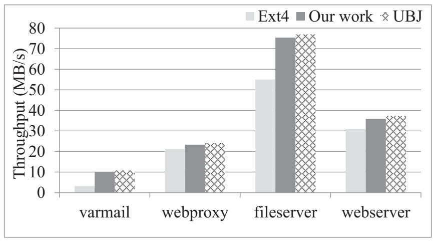
Fig. 6: The normalized write access on NVM.

overhead relieves the contention to hardware resources like memory bus and DMAs.

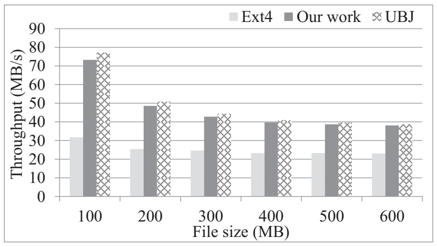
Fig. 4(b) and Fig. 4(c) show the throughput of different schemes for IOzone and Postmark. We measure the performance of random write for IOzone with varied file size ranging from 100MB to 600MB, and the write performance for Postmark with varied number of transactions ranging from 2,000 to 12,000. For IOzone, our scheme outperforms ext4 by 86.44%, with 7.4% performance loss compared with UBJ. And for Postmark, our scheme achieves 137% write performance improvement over ext4, with 11% performance gap compared with UBJ. The performance benefits of the NVM buffer cache are certified in all situations. As the file size and transaction number increase, the performance of our scheme gets closer to UBJ since checkpointing is triggered more frequently.

2) Migration Overhead: Different from ext4 and UBJ, our scheme is designed for the hybrid buffer cache. In order to reduce the journaling overhead and overcome the constraints of NVM, we migrate the infrequently updated data to NVM and the write-burst data to DRAM. In particularly, JAPM accelerates the migration of dirty data from DRAM to NVM to reduce the journaling overhead. Meanwhile, the accelerated migration affects the performance of the hybrid buffer cache since it causes extra writes on NVM. In order to investigate the performance degradation caused by migrations, when a migration occurs, we only mark the data as *migrated* instead of performing the migration which removes the migration overhead. Fig. 5 shows the performance comparisons of the modified scheme without migration overhead (represented as Our work-WM) and the original scheme for IOzone. As file size increases, the journaling overhead makes a greater impact on performance than the migration overhead. For the different file size, the migration causes 19.6% performance degradation, on average.

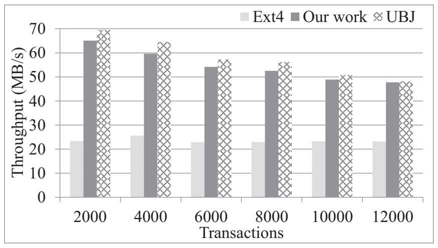
3) Write Access on NVM: In order to evaluate the benefits of our scheme in terms of the lifetime of the hybrid buffer



(a) Throughput of Filebench workloads.



(b) Random write of IOzone with varied file size.



(c) Write of Postmark.

Fig. 4: Performance evaluation of different benchmarks.

cache, we record and compare the writes occurred on NVM under our scheme and UBJ with different Filebench workloads. We normalized the results with the baseline of UBJ.

We observe that, varmail issues small write requests frequently. Since our scheme ensures frequently updated data is accessed in the DRAM buffer cache, our scheme absorbs only 31% of the writes compared with UBJ, as shown in Fig. 6. But for fileserver, the write size of NVM is relatively large, 52.7%, since the large memory footprint leads to considerable frozen blocks on NVM. Therefore, for write-intensive workloads, varmail and fileserver, our scheme improves the lifetime of the hybrid buffer cache by 223% and 89%, compared with UBJ, respectively. But for the read-intensive workloads, DRAM absorbs almost all the writes. However, in our statistics, when a data block is loaded into DRAM or NVM for the first time, it causes a block write on the hybrid buffer cache. As we can see, the write ratio of NVM is relatively large for webproxy and webserver. However, the large write ratio will not cause significant influence on the lifetime of the hybrid buffer cache under the read-intensive workloads. Therefore, our scheme enhances the write endurance of NVM effectively.

VI. Conclusion

In this paper, we present an unified DRAM and NVM hybrid buffer cache architecture to reduce the journaling overhead for modern filesystems. We propose a journaling-aware page management policy for the hybrid buffer cache architecture to reduce the journaling overhead and overcome the constraints of NVM. Moreover, we put forward a partial in-place commit journaling scheme to guarantee the atomicity semantics of the transactional execution. We implement the proposed techniques on Linux 3.14.52 and measure the performance with representative I/O-intensive benchmarks. The experimental results show that our scheme effectively improves the I/O performance and prolongs the lifetime of the hybrid buffer cache without any loss of reliability.

VII. Acknowledgements

This research is sponsored by the State Key Program of National Natural Science Foundation of China No. 61533011, National High-tech R&D Program of China (863 Program) No. 2015AA011504, Shandong Provincial Natural Science Foundation under Grant No. ZR2015FM001 and the Fundamental Research Funds of Shandong University No.2015JC030.

References

[1] F. Chen, M. P. Mesnier, and S. Hahn. A protected block device for persistent memory. In *Proceeding of the 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2014.
[2] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceeding of the 46th ACM/IEEE Design Automation Conference*, pages 664–669, 2009.
[3] C. H.-Y. et al. EECache: A comprehensive study on the architectural design for energy-efficient last-level caches in chip multiprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(2):17, 2015.
[4] H. S. et al. Efficient page caching algorithm with prediction and migration for a hybrid main memory. *ACM SIGAPP Applied Computing Review*, 11(4):38–48, 2011.
[5] S. J. et al. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.
[6] W. D. N. et al. Iozone filesystem benchmark. *URL: www. iozone.org*, 55, 2003.
[7] Z. Fan, D. H. Du, and D. Voigt. H-ARC: A non-volatile memory based cache policy for solid state drives. In *Proceeding of the 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2014.
[8] Y. Hao and Y. Wang. *Design Exploration of Emerging Nano-scale Non-volatile Memory*. Springer, 2014.
[9] J. Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, www. netapp.com/tech_library/3022.html, 1997.
[10] E. Lee, H. Bahn, and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *FAST*, pages 73–80, 2013.
[11] N. S. H. Lee S, Bahn H. CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. *IEEE Transactions on Computers*, 63(9):2187–2200, 2014.
[12] J. A. K. Li Y, Chen Y. A software approach for combating asymmetries of non-volatile memories. In *Proceeding of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 191–196.
[13] R. McDougall. Filebench: Application level file system benchmark, 2014.
[14] J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi. Operating system support for NVM+ DRAM hybrid main memory. In *HotOS*, 2009.
[15] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.
[16] B. R. Ramos L E, Gorbatov E. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95, 2011.
[17] H. Seok, Y. Park, and K. H. Park. Migration based page caching algorithm for a hybrid main memory of DRAM and PRAM. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 595–599.
[18] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Notices*, 46(3):91–104, 2011.
[19] B. Wu. An architecture-level cache simulation framework supporting advanced PMA STT-MRAM. In *Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 7–12.