Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory

Eunji Lee[1], Hyokyung Bahn[1], and Sam H. Noh[2]
[1] *Ewha University, {alicialee, bahn}@ewha.ac.kr*
[2]*Hongik University, http://next.hongik.ac.kr*

Abstract

Journaling techniques are widely used in modern file systems as they provide high reliability and fast recovery from system failures. However, it reduces the performance benefit of buffer caching as journaling accounts for a bulk of the storage writes in real system environments. In this paper, we present a novel buffer cache architecture that subsumes the functionality of caching and journaling by making use of non-volatile memory such as PCM or STT-MRAM. Specifically, our buffer cache supports what we call the in-place commit scheme. This scheme avoids logging, but still provides the same journaling effect by simply altering the state of the cached block to frozen. As a frozen block still performs the function of caching, we show that in-place commit does not degrade cache performance. We implement our scheme on Linux 2.6.38 and measure the throughput and execution time of the scheme with various file I/O benchmarks. The results show that our scheme improves I/O performance by 76% on average and up to 240% compared to the existing Linux buffer cache with ext4 without any loss of reliability.

1. Introduction

Non-volatile memory such as PCM (phase-change memory) and STT-MRAM (spin torque transfer magnetic RAM) is being considered as a replacement for DRAM memory [1-8]. Though currently unfit for complete replacement due to cost, non-volatile memory has become a viable component that may be added to current systems to enhance performance [9]. Temporary non-volatile memory solutions in the form of supercapacitor-supported DRAM are also finding its place in the server market [10]. As non-volatile memory is expected to provide performance competitive to DRAM while retaining non-volatile characteristics, studies on exploiting these dual characteristics have recently been catching interest [1-8]. This paper also exploits the non-volatile characteristic of these new memory technologies through marriage of the buffer cache and journaling layers. We believe, to the best of our knowledge, that our work is the first to propose such a union.

In traditional systems, as main memory is volatile, the file system may enter an inconsistent and/or out-of-date state upon sudden system crashes [11]. To relieve this problem, *journaling*, which is a technique that logs the updates to non-volatile storage within a short time period for high reliability and fast system recovery, is widely adopted in modern file systems [12]. However, journaling is a serious impediment to high performance due to its frequent storage accesses. For example, a recent study has shown that synchronous writes due to journaling dominates storage traffic in mobile handheld devices leading to severe slowdown of flash storage [13]. In cloud storage systems, even though there is a consensus that journaling is necessary, it is not deployed due to the high cost of network accesses involved in journaling [14]. In this study, we present a novel buffer cache architecture that removes almost all storage accesses due to journaling without any loss of reliability by intelligently adopting non-volatile memory as the buffer cache.

At first glance, simply adopting a non-volatile buffer cache may seem to be enough to provide a highly reliable file system. However, this is not the case as there are two requirements that are needed to support reliability in file systems: *durability* and *consistency*. A non-volatile buffer cache, simply as is, ensures durability as it maintains data even after a power failure. However, consistency cannot be guaranteed upon system crashes just by providing non-volatility to the buffer cache. Take, for example, a typical situation where a write operation requires both data and metadata updates but the system crashes after updating only metadata. Then, the data in the buffer cache loses consistency, eventually leading to an inconsistent file system state.

We propose a new buffer cache architecture, called UBJ (Union of Buffer cache and Journaling), that resolves this problem by providing the functionality of

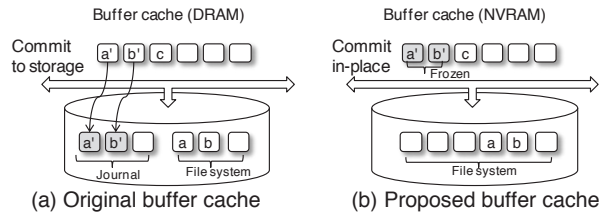


Fig. 1. Commit process of our buffer cache compared to the original buffer cache with journaling.

journaling without frequent storage accesses. Specifically, we propose *In-place Commit,* which changes the state of updated cache blocks to frozen (i.e., write protected) and manages them as a transaction right at where it is currently located. This scheme does not perform additional logging but simply reaps the same effect just by changing a state. Furthermore, as the block in the frozen state can still be used as a cache block, the effectiveness of the buffer cache is not deteriorated. Figure 1 shows the commit process of our buffer cache compared to that of the existing buffer cache in conjunction with journaling.

Previous work most closely related to this study also attempts to relieve journaling overhead by adopting non-volatile memory for file system and database management [15, 16, 17]. They improve performance by adding non-volatile memory as a separate journal area or as a write buffer of log files. However, in these schemes, the non-volatile memory cannot function as a cache and is kept separately from the buffer cache. Our scheme is different in that we intelligently union the journaling functionality into the buffer cache architecture, thereby minimizing additional memory copy and space overhead.

We have implemented a prototype of our buffer cache with in-place commit in Linux 2.6.38. Measurement results with various storage benchmarks show that our scheme improves file system performance by 76% on average and up to 240% compared to the existing Linux buffer cache with ext4 set to the journal mode.

The remainder of this paper is organized as follows. Section 2 investigates the effect of journaling on the write traffic of storage. Section 3 describes our buffer caching architecture and algorithm in detail. In Section 4, we discuss cache performance issues inherent in our scheme. Section 5 presents a brief description of the implementation and discusses the experimental results of the implementation. Section 6 concludes this paper.

2. Analysis of Storage Write Traffic

Figure 2 shows the amount of write traffic from the buffer cache to storage when journaling is employed relative to when it is not for various workloads. As Figure 2 shows, write traffic increases dramatically when journaling is used; on average the data stored with journaling is 2.7 times more than that without journaling. When we do not use journaling, storage writes occur only when dirty blocks are evicted from the cache or when there is an explicit `sync` operation. However, with journaling, there are two more cases that cause storage writes. The first case is when a commit to the journal area occurs. The second case is when checkpointing, which writes the updated data to permanent file system locations, occurs. Checkpointing is triggered periodically and is also activated when the amount of free space in the journal area drops below a certain threshold.

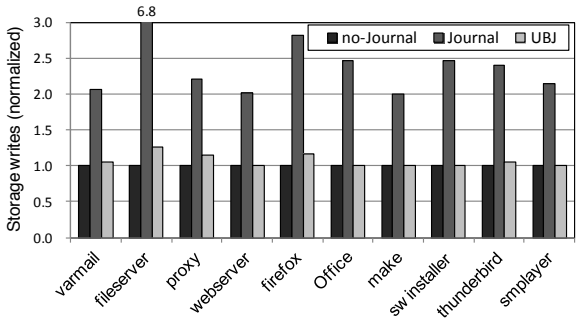


Fig. 2. Write traffic with journaling and our proposed scheme (UBJ) relative to when journaling is not used (denoted no-journal) for various workloads.

As we can see, journaling accounts for a considerable portion of storage writes for all workloads, and thus, is a potential source of performance degradation. As Figure 2 shows, however, our UBJ scheme performs journaling but eliminates most of its storage writes.

3. Buffer Cache with In-place Commit

In UBJ, the buffer cache and the journal area share the same allocated space. Every block in this space functions either as a cache block, a log block, or a cache and log block at the same time when the block has dual-purpose. We can represent the state of each block as a combination of three state indicators: frozen/normal state, dirty/clean state, and up-to-date/out-of-date state (Refer to Figure 3). The frozen/normal state distinguishes whether the block is a (normal) cache block or a (frozen) log block. The dirty/clean state indicates whether the block has been modified since it entered the cache. The up-to-date/out-of-date state indicates whether the block is the most recent version or not. This last distinction is necessary as multiple blocks for the same data may exist in the buffer/journal space.

In the following, we use these states and the transitions between these states to describe the workings of

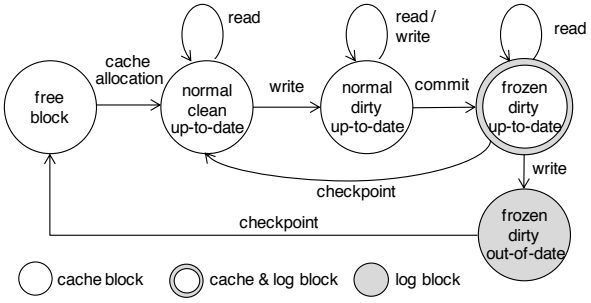


Fig. 3 State diagram of a cache block in our scheme.

UBJ. The scheme is described as three distinct operations: the read/write operations, the commit operation, and the checkpointing operation.

A. Read/Write Operations

Reaction to a write request depends on the state of the block. If the block is normal, it is simply updated. Otherwise, that is, if it is frozen (a log block), a copy is made to a new location and the updated data is written to the copy. The copy then becomes up-to-date, while the original becomes out-of-date. This allows logged data to remain safe against system failures.

Read requests, on the other hand, are serviced irrespective of block state and do not alter the states of the block. This is because reading data does not affect the consistency of blocks.

B. Commit Operations

As part of transaction management, UBJ periodically commits data by changing the state of normal dirty blocks to frozen dirty blocks. This is done by keeping and manipulating transactions maintained as lists. There are three types of transaction as shown in Figure 4. The first is a *running transaction* that maintains a list of normal dirty blocks that became dirty after the previous commit operation. When a commit operation is issued, this running transaction is converted to a *commit transaction*. At this point, the blocks on the commit transaction, which are in normal state, are committed in-place (hence, the name In-place Commit) simply by converting their state to frozen. New block writes arriving during this conversion process is simply added to a new running transaction created for the next commit period. During the state conversion process, UBJ also checks if the same data block had been committed before, and if so, keeps track of the previous block as an old-commit block. For efficient space management, these old-commit blocks may be released immediately at this point. When all dirty data blocks in the commit transaction become frozen, we change the state of the transaction to a *checkpoint transaction*. The checkpoint transactions are maintained in the buffer cache until they are reflected onto the file system via checkpointing. Should a system crash occur, they are the ones used to recover the file system into a consistent and up-to-date state.

C. Checkpoint Operations

The checkpoint operation updates the file system with committed data. UBJ scans the checkpoint transaction lists and reflects them on to their permanent locations in the file system. When a single data block has been committed multiple times, only the last commit block is checkpointed in order to reduce checkpointing overhead. After a transaction checkpoint is completed,

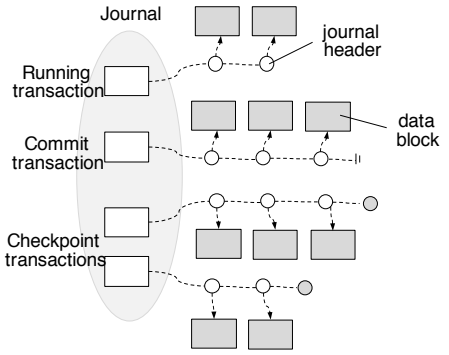
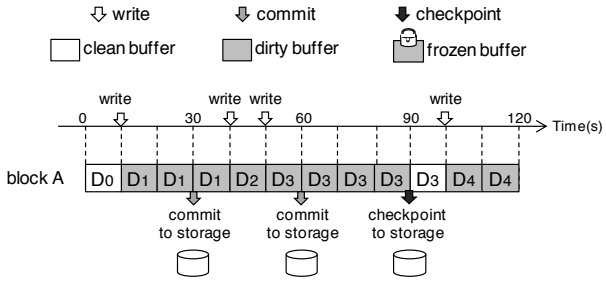


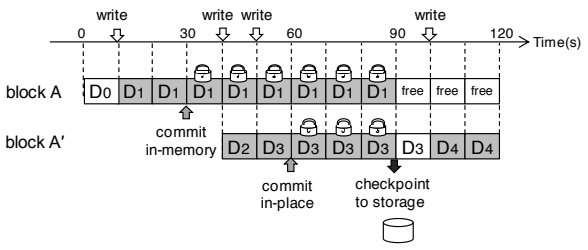
Fig. 4 Data structures for UBJ.

cache space occupied by frozen blocks is reclaimed. During this process, if the block is up-to-date, we reuse it as a buffer cache block by changing its state to normal and clean, and updating its associated metadata. On the other hand, if the block is out-of-date, the space is reclaimed as a free block.

While the commit operation is performed frequently to reduce the vulnerability window of reliability, checkpointing may be done less frequently as it does not directly affect system reliability. Nevertheless, excessive delay in checkpointing may have negative effects. As the interval between checkpointing grows, more and more cache space will be occupied by frozen blocks effectively reducing the buffer cache, potentially leading to degradation of buffer cache performance. To prevent this situation, our scheme triggers checkpointing



(a) Original buffer cache with journaling



(b) The proposed buffer cache

Fig. 5 Comparison of our UBJ scheme and the original buffer cache with journaling

Table 1. Summary of workload characteristics

workload	avg. file size	# of files	avg. op. size	# of ops.	r:w ratio
Varmail	16KB	1,000	4KB	6,638K	1:1
Proxy	16KB	10,000	5KB	6,723K	5:1
Webserver	16KB	1,000	14KB	6,432K	10:1
Fileserver	128KB	1,000	72KB	6,843K	1:2

when the number of frozen blocks becomes larger than a certain threshold as is done with conventional journaling. Checkpointing is also triggered by the page reclamation daemon when it evicts dirty blocks from the cache.

D. Example

Figure 5 shows the workings of UBJ in contrast to a typical buffer cache scheme. In this example, the commit and checkpointing periods are set to 30 and 90 seconds, respectively. The initial content of block A is D_0. At time 10, a write request modifies the content of block A to D_1, converting it into a dirty block. 20 seconds later, a commit occurs. To service the commit, UBJ simply changes the state of block A to frozen, whereas a typical buffer cache would incur a write to storage.

Suppose another write to block A occurs at time 40. UBJ at this point copies block A into another location A′, and then writes the new data to location A′. Block A, then becomes out-of-date, while block A′ is now up-to-date. (We quantify the effect of having multiple copies of the same data block later.) Subsequent writes (write at time 50, in this example) are performed on block A′ until it is committed and hence, frozen (at time 60). When checkpointing occurs at time 90, UBJ reflects only the latest version of block A′, that is D_3, into the file system eliminating unnecessary writes (D_1 and D_2). After checkpointing, space occupied by block A is reclaimed, while the up-to-date version of block A′ is reused as normal buffer cache data.

In this example, we see that UBJ writes to storage only once when checkpointing, while a typical buffer cache scheme generates storage writes every time it commits or checkpoints. This results in writes of blocks that are eventually overwritten and did not need to be saved. In practice, as commits are much more frequent than checkpoints, the reduction in the number of storage writes can become substantial.

E. System Recovery

System crashes incur inconsistency problems as data may remain partially updated in the buffer cache, the

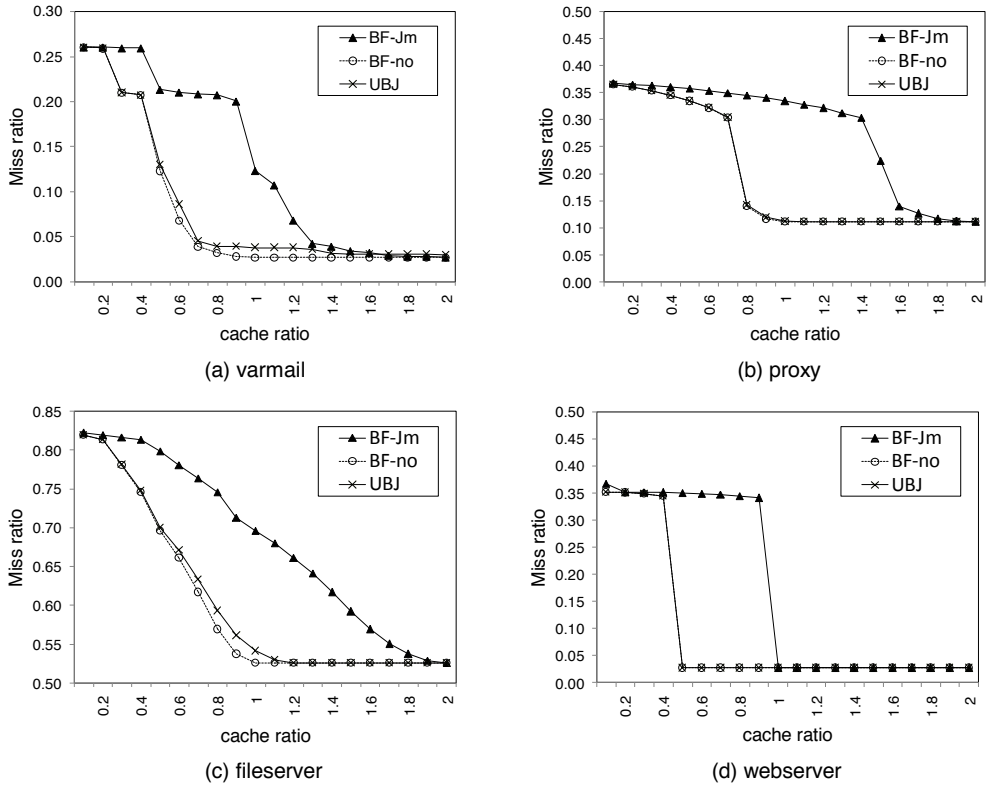


Fig. 6 Cache miss ratio as a function of the cache size.

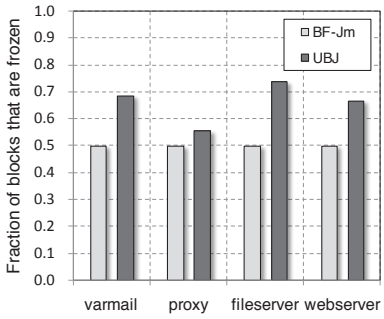


Fig. 7. Ratio of journal area for each Filebench workload

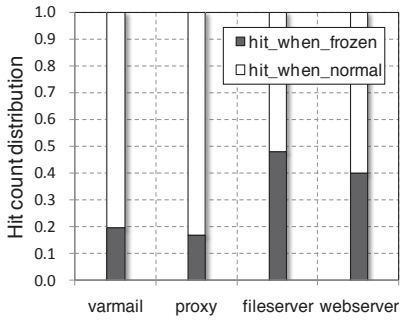


Fig. 8. Hit counts for frozen blocks for the Filebench workload.

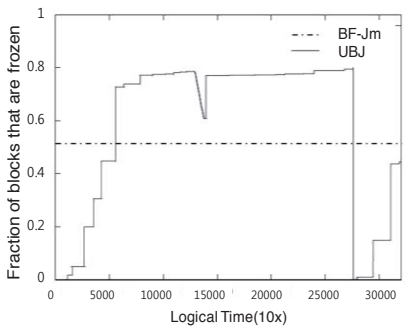


Fig. 9. Change of the journal area size as time progresses for the fileserver workload

journal area, and the file system. Hence, upon reboot from an abnormal termination, UBJ goes through the following steps for recovery.

First, the running transaction list is checked. This list holds blocks that are residing in the buffer cache and may have been partially updated. However, since these are uncommitted data and thus, not reflected on the file system, we simply invalidate these blocks.

Next, we consider the commit transaction list. When a crash occurs in the middle of committing a transaction, the commit transaction list may be in a partially committed state containing a mixture of committed and still-uncommitted blocks. In this case, as the commit operation did not complete properly, UBJ simply invalidates the data in the same way as it did with the running transaction list.

Finally, the system state is made consistent by making use of the checkpointing transactions. Even though the file system may be corrupted if a crash occurs while executing checkpoint operations, the system can be restored to a consistent state simply by redoing the operation as all the checkpoint transactions still reside in the buffer cache.

4. Cache Performance

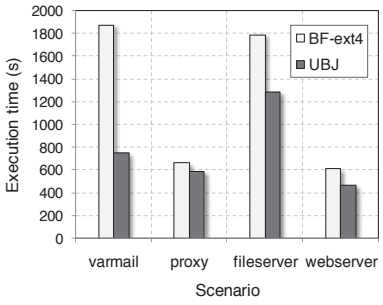
As our scheme holds log blocks in the buffer cache, a single data block may occupy multiple cache blocks, degrading the space efficiency of the buffer cache. To investigate this effect, we compare the buffer cache miss ratio of the original buffer cache that is used only for caching and with no log blocks (BF-no) and UBJ, where cache and journal space is in union. In addition, for comparison purposes, we also observe the miss ratio of a hypothetical scheme that uses half of its allocated memory space as a dedicated buffer cache and the other half as a dedicated journal area. We will call this scheme buffer caching with in-memory journaling (BF-Jm). This scheme works just like conventional journaling, except for the fact that the journal area is not in storage but in memory, possibly non-volatile memory.

Figure 6 plots the cache miss ratio of the three buffer cache schemes for the Filebench benchmark, which is a representative file system benchmark. Table 1 summarizes the characteristics of the Filebench workloads used in this paper.

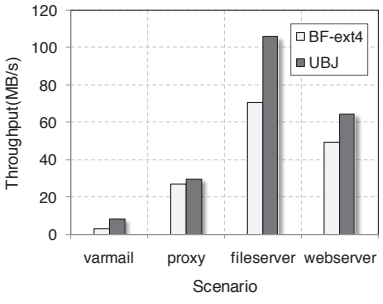
We vary the cache size from 0.1 (10%) to 2.0 (200%) relative to the I/O footprint. Generally, the cache size of 1.0 is identical to infinite cache capacity since the cache will simultaneously accommodate all block references of the trace. However, as UBJ and BF-Jm use a certain portion of the buffer cache space for the journal area, cache size of 1.0 may be insufficient to accommodate all the requests. For BF-Jm, which dedicates half of the total space to the buffer cache and the other half to the journal area, a cache size of 2.0 is equivalent to infinite cache capacity. Moreover, to monitor the worst case performance degradation of UBJ, we delay checkpointing of frozen log blocks until they are selected as victims by the cache replacement policy.

The results shown in Figure 6 reveals that the cache performance of UBJ exhibits only marginal degradation compared to that of BF-no, while performance of BF-Jm degrades significantly. To understand this more clearly, we also plot in Figure 7 the portion of the buffer/journal space occupied by frozen blocks when the cache size is 0.7 (70%) of the workload footprint, which is where the performance gap between UBJ and BF-Jm is largest.

We observe that UBJ uses a substantial portion of the buffer/journal space for the journal area for all workloads. This results in read requests being serviced by these frozen blocks. As Figure 8 shows, roughly 16% to 48% of all hits occur in the journal area for UBJ. In contrast, schemes like BF-Jm that simply dedicate (non-volatile) journal space cannot take advantage of the data that resides there. Figure 9 plots the ratio of journal

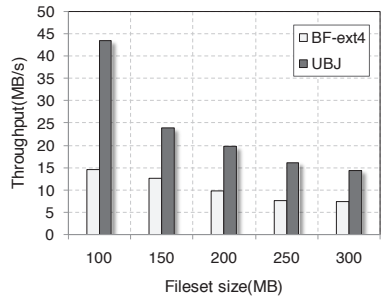


(a) execution time

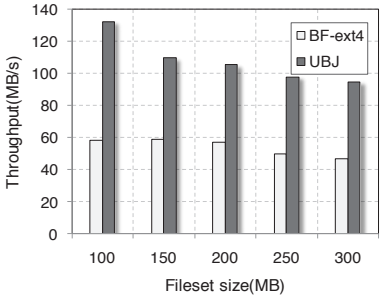


(b) throughput

Fig. 10. Throughput and execution time of the Filebench workloads.

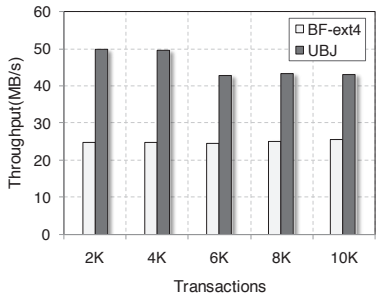


(a) random write

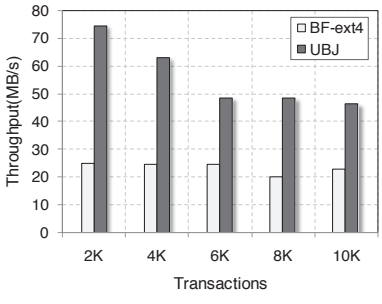


(b) sequential write

Fig. 11. Throughput of IOzone as the fileset size is varied.



(a) read



(b) write

Fig. 12. Throughput of Postmark as the number of transactions is varied.

space for the fileserver workload as time progresses. The figure shows that the journal space of UBJ dynamically changes according to workload evolution. (Results for other workloads that are not shown show similar dynamic behavior.)

5. Performance Evaluation

A. Implementation

We implemented a UBJ prototype on Linux 2.6.38. We compare our scheme with ext4 that runs on the original Linux (BF-ext4). We set the journaling option of ext4 to journal-mode, which logs both data and metadata, to provide the same consistency semantics as UBJ. According to conventional configurations, the commit period is set to five seconds for both schemes; checkpointing is triggered when a fourth of the journal area is filled or five minutes have elapsed since the last checkpointing time. For better space efficiency, we adopt the early reclamation policy that releases old versions of frozen log blocks, that is, old-commit blocks, immediately when a recent commit version of the same data block is generated.

Our experimental platform consists of the Intel Core i3-2100 CPU running at 3.1GHz and 4GB of DDR2-800 memory. Though our design assumes non-volatile main memory like PCM or STT-MRAM, as it is not yet commercially available, we simply use a portion of the DRAM as buffer/journal space. We measure the performance with three representative storage benchmarks: Filebench [18], IOzone [19], and Postmark [20].

B. Experimental Results

Figure 10 shows the execution time and the throughput of UBJ compared to BF-ext4 for the Filebench workloads. As shown in the figure, UBJ performs better than BF-ext4 for all workloads. The execution time and throughput of UBJ is better than BF-ext4 by 30.7% and 59.8% on average. Specifically, the performance improvement of varmail is the largest among the four workloads, which is 60% and 240%, respectively, in execution time and throughput. The reason behind such large improvements is that varmail contains a large number of write requests, thus incurring frequent commits (though checkpointing is not performed frequently due to its small memory footprint). Moreover, varmail issues read and write requests concurrently. Though our scheme does not have an explicit policy to improve read performance, eliminating frequent storage writes relieves the contention to hardware resources like the memory bus and DMAs, leading to improvements in I/O performance in general. Even for read-intensive workloads like webserver and proxy, UBJ improves performance by 23% and 11%, respectively, due to this reason. For fileserver, which is write-intensive, the improvement of our scheme is relatively small compared to varmail. This is because most write requests in file-

server are large and sequential writes, which leads to frequent checkpointing.

Figure 11 shows the throughput of UBJ and BF-ext4 for the IOzone benchmark. IOzone 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 two IOzone scenarios, random write and sequential write, varying the total fileset sizes ranging from 100MB to 300MB.

As the figure shows, UBJ outperforms BF-ext4 in all configurations. Specifically, the performance improvement is 2.1 times on average. The source of this enhancement is the large reduction in storage writes achieved by in-place commit. Note also that UBJ performs better relative to BF-ext4 as the fileset size is reduced. This is because with smaller fileset size checkpointing is done less frequently leading to reduced storage accesses. Unlike UBJ, BF-ext4 is less sensitive to the fileset size. This is because the performance of BF-ext4 is dominated by the number of commits, which is independent of the fileset size.

Another observation that can be made with IOzone is that UBJ is more sensitive to the fileset size for random writes. This is because random writes are largely affected by seek times as checkpointing for random accesses incur large seek overhead to move the disk head to the requested location.

Figure 12 shows the throughput for the Postmark benchmark. Postmark emulates an email server that concurrently performs read and write operations. The benchmark reports separate performance numbers for read and write operations. We vary the number of Postmark transactions from 2,000 to 10,000 on 1,000 files, whose average size is 500KB. As shown in the figure, UBJ exhibits 109% better throughput than BF-ext4 on average. Note that read operation throughput also improves due to the reduced storage writes. We also observe that the performance improvement of UBJ decreases as the number of transactions increases. Similarly to IOzone, this is because with a larger number of transactions UBJ increases checkpointing, while keeping other factors constant. Hence, the relative performance gap between UBJ and BF-ext4 is reduced.

Finally, we see how the performance of UBJ is affected as the commit period changes. Figure 13 shows the latency per operation for the varmail workload. As we can see, the latency of BF-ext4 becomes small as the commit period changes from 1 to 5 seconds. This is because storage writes become less frequent. This, of course, increases the window of vulnerability for BF-ext4. In contrast, the latency of UBJ is not sensitive to the commit period as the overhead of commit is very small in UBJ. This result indicates that UBJ can shorten the window of vulnerability without sacrificing performance.

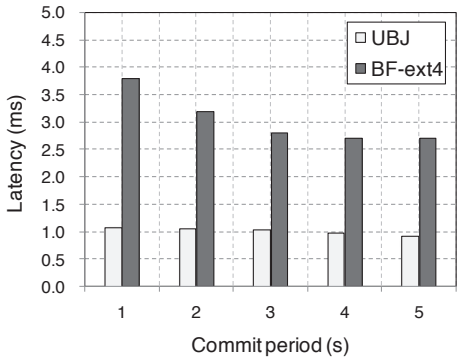


Fig. 13. Latency of varmail varying the commit period

6. Conclusion

In this paper, we presented a novel buffer cache architecture that subsumes the function of caching and journaling in a unified non-volatile memory space. Specifically, blocks in the buffer cache are converted to journal logs through what we call In-place Commit. In-place Commit simply changes the state of cached block from a normal state to a frozen state. By so doing, it reaps the same effect as journaling. Furthermore as the block in frozen state can still be used as a cache block the effectiveness of the buffer cache is not deteriorated. Based on In-place Commit, we implement UBJ (Union of Buffer cache and Journaling) on Linux 2.6.38. Measurement studies showed that UBJ significantly improves file I/O performance compared to the existing Linux ext4 journaling scheme without any loss of reliability. The effectiveness of UBJ will increase in environments such as cloud storage and networked storage systems where storage access cost are higher. We are planning to extend our scheme to this area in the near future.

Acknowledgements

We would like to thank our shepherd Anna Povzner and the anonymous reviewers for their insight and suggestions for improvement. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No.2011-0028825) and (No. 2012R1A2A2A01045733) and by the IT R&D program MKE/KEIT (No.10041608, Embedded System Software for New-memory based Smart Devices.) Hyokyung Bahn is the corresponding author of this paper.

References

[1] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for NVM+DRAM hybrid main memory," In Proceedings of the 12th USENIX Workshop on Hot Topics in Operating Systems (HotOS), 2009.

[2] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," In Proceedings of the 36th ACM/IEEE International Symposium on Computer Architecture (ISCA), 2009.

[3] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase change memory architecture and the quest for scalability," Communications of the ACM, Vol. 53, No. 7, pp.99-106, 2010.

[4] S. Lee, H. Bahn, and S. H. Noh, "Characterizing memory write references for efficient management of hybrid PCM and DRAM memory," In Proceedings of the 19th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp.168-175, 2011.

[5] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," In Proceedings of the 36th ACM/IEEE International Symposium on Computer Architecture (ISCA), 2009.

[6] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," In Proceedings of the 36th ACM/IEEE International Symposium on Computer Architecture (ISCA), 2009.

[7] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," In Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2010.

[8] 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 Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), 2009.

[9] http://www.vikingtechnology.com/uploads/nv_whitepaper.pdf

[10] http://denalimemoryreport.com/2012/08/20/44tbyte-skyera-skyhawk-ssd-employs-everspin-mram-as-write-cache/

[11] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio file cache: Surviving Operating System Crashes," In Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1996.

[12] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," In Proceedings of the USENIX Annual Technical Conference (ATC), 2005.

[13] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," In Proceedings of the 10th USENIX File and Storage Technologies (FAST), 2012.

[14] M. Rubin, "File systems in the Cloud," Linux Foundation Collaboration Summit, 2011

[15] D. Phillips, "Zumastor linux storage server," In Proceedings of the Linux Symposium, pp. 135-143, 2007.

[16] R. Fang, H. Hsiao, B. He, C. Mohan, Y. Wang, "High performance database logging using storage class memory," In Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE), pp. 11-16, 2011.

[17] 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," Journal of Korean Institute of Information Scientists and Engineers, Vol. 16, No. 2, pp. 247-251, 2010.

[18] W. Norcutt, IOzone Filesystem Benchmark, available at http://www.iozone.org/.

[19] J. Katcher, "Postmark: a new file system benchmark," Technical report TR-3022, Network Appliances, 1997.

[20] Filebench, http://www.solarisinternals.com/wiki/index.php/FileBench