

An Efficient Buffer Cache Management Policy Considering Single-Access Journal Data in Mobile Devices

¹Hyunkyung Choi, ²Dohee Kim, ^{3*}Hyokyung Bahn

¹Dept. of Computer Engineering, Ewha University

Email: bluechk@ewhain.net

²Dept. of Electrical and Computer Engineering, Ajou University

Email: kdohee@ajou.ac.kr

^{3*}Dept. of Computer Engineering, Ewha University

Email: bahn@ewha.ac.kr

Abstract

Buffer cache maintains file data retrieved from storage, thereby servicing subsequent requests on the same data without accessing slow storage. Although buffer cache plays an important role in improving storage performances, it may cause data inconsistency if a sudden system crash occurs while data in buffer cache are being written to storage. To prevent such situations, a lightweight journaling library, which periodically writes buffer cache data to storage, is usually used in mobile devices. This paper presents a new buffer cache management policy that aggressively evicts useless journal data from buffer cache. To do so, our policy first classifies single-access data like journal data, and evicts them aggressively from the buffer cache. Experimental results with various mobile applications show that the proposed policy improves the cache hit ratio significantly.

Keywords: Buffer cache; single-access; journaling; storage.

1. Introduction

To buffer the speed gap between DRAM memory and secondary storage, buffer cache is used in file systems and database [1], [2], [3], [4]. Buffer cache maintains requested file data in a certain part of DRAM memory, thereby serving subsequent requests on the same data directly without accessing slow storage. However, data inconsistency may happen in case that a sudden system crash occurs while data in the buffer cache are being written to storage.

To alleviate the data inconsistency situation, a lightweight journaling library such as SQLite is usually used in mobile devices [5], [6]. To modify file data in a journaling library, a copy is first made, which will be used for the recovery of corrupted file data [5]. However, this incurs inefficiency of buffer cache as an additional data copy wastes the cache space. For example, if a file data B needs to be updated, a backup data B_{old} is first created and then written to storage. Once B has been updated, it is now the most recent version, and thus all subsequent requests are served by B . Nevertheless, B_{old} is still preserved in the buffer cache as file systems cannot recognize it as an old version of data that will never be used. Although B_{old} is needed when a system crash happens, we use the copy in storage, not the cached copy. This useless copy in memory excessively wastes the buffer cache space.

In this paper, we analyze file access characteristics of various applications in mobile devices and observe that a large proportion of file data incur a single-access. The main reason behind this is the journaling library, which creates an old copy of file data before updating. As this old copy has a different file name from the updated data, it is classified as a single-access data, which wastes the buffer cache space significantly.

In order to alleviate this inefficiency, we present a new buffer cache management policy that considers non-reusability of journal data. Specifically, our policy classifies single-access journal data and evicts them from the buffer cache when they are being written to storage. This allows more useful data to be maintained in the buffer cache, improving the storage performances. Experimental results with a variety of applications show that the proposed buffer cache management policy improves the cache hit ratio by 20% on average compared to the systems that do not use it.

The remainder of this paper is organized as follows. Section 2 analyzes file access characteristics of mobile applications and presents the motivation of this research. Section 3 presents a

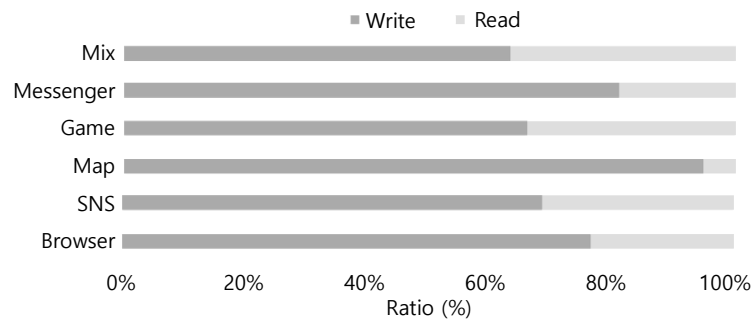


Fig. 1. Operation ratio of mobile application's file accesses.

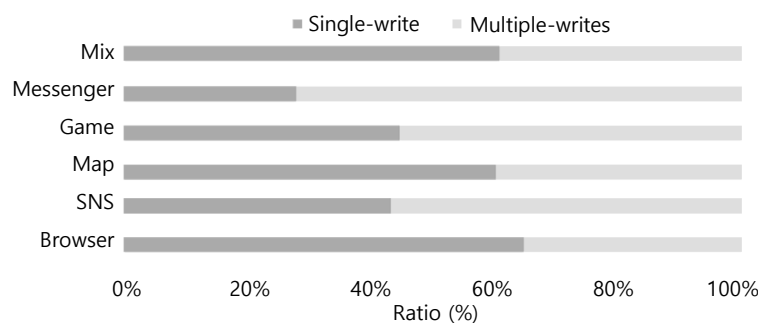


Fig. 2. Analysis of write operation's ratio in mobile application's file accesses.

new buffer cache management policy considering the single-access journal data in file accesses. Section 4 presents the performance evaluation results to assess the effectiveness of the proposed policy. Finally, Section 5 concludes this paper.

2. Analyzing File Access Characteristics

We analyze file access characteristics of various mobile applications. Fig. 1 shows the file operation ratio of various Android applications. As the figure shows, applications exhibit write-dominated access patterns. In some applications, the ratio of writes is over 90%. Such high ratio of writes happens as mobile applications use the journaling library in file writes. Note that this is different from traditional desktop applications in which reads account for a large proportion of file accesses [7].

Fig. 2 classifies write accesses of file data into single-access and multiple-accesses. As shown in the figure, the ratio of single-access data is very large. Note that these large proportion of single-access data will not be used again although we preserve them in buffer cache. Thus, we can conclude that single-access journal data are not useful in enhancing file system performances but pollute the buffer cache.

3. A New Buffer Cache Management Policy

Buffer cache manages file data by a unit of block. There are two types of blocks; a clean block that has not been modified since it was loaded into the buffer cache, and an updated block that has been modified after entering the buffer cache. As a clean block is the same as the original block in storage, it can simply be eliminated when it is removed from the buffer cache. On the contrary, an updated block needs to be written to storage before it is removed.

To preserve a system from sudden crashes, file systems periodically write updated blocks to storage rather than delaying the writes until their eviction time. For example, Ext4 writes updated blocks to storage every 5-30 seconds. After writing to storage, an updated block changes its state to a clean block. Fig. 3 shows an example of periodic writing that occurs every 5 seconds. In this example,

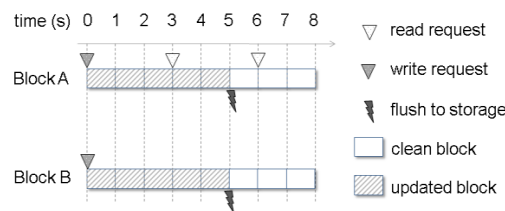


Fig.3. Buffer cache state with the legacy system.

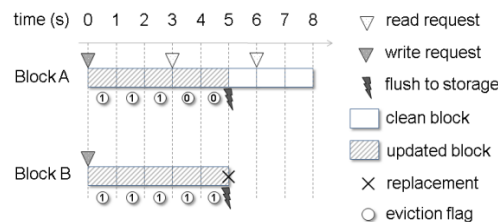


Fig.4. Buffer cache state with the proposed policy.

blocks A and B are updated blocks at 5 seconds as they have been written at 0 second. After periodic writing at 5 seconds, blocks A and B become clean blocks as their updated versions have been written to storage.

Due to the journaling library, we showed in the previous section that mobile applications cause a large proportion of single-accesses. To perform the early eviction of these useless data, we aggressively evict a single-access block from the buffer cache when a periodic write operation on that block happens. For example, block B in Fig.3 is the target of eviction at 5 seconds in our policy. We do not evict block A as it has also caused an additional read operation. Note that a journal data will not be accessed again and thus block A is not classified as a non-reusable journal data. Blocks other than targets of early eviction are managed by the conventional replacement policy like the LRU (Least Recently Used) policy. Thus, they will be removed from the buffer cache if they are not accessed for a long time.

Fig. 4 shows an example situation of the proposed policy with the same read/write requests given in Fig. 3. Our policy writes block B at 5 seconds and then removes it from the buffer cache as there was a single-write on that block. Although block A also incurs a single-write, it is not evicted from the cache as there was a read operation on that block. To check whether a block should be evicted upon a periodic writing operation, our policy maintains an eviction flag for each cached block. The eviction flag of one indicates that the block should be evicted when it is written to storage. A block with the eviction flag of zero is not evicted even after writing to storage.

Although a single-access block was evicted, it may return to the buffer cache upon its further accesses. In that case, as the write count history of the block does not exist, the block may be classified as a single-access block again and is certain to be evicted repeatedly. To alleviate this situation, a small metadata buffer that maintains the metadata of evicted blocks for a while is used to see whether there was a recent write history on that block.

If a block enters the cache whose write history already exists in the metadata buffer, the eviction flag is reset to zero upon its next write. We set the size of the metadata buffer to the original cache size. Maintaining this size of metadata buffer has low overhead because a metadata buffer only maintains 12 bytes of information including inode and a block identifier whereas each of an actual block contains 4KB of data [8], [9], [10], [11].

4. Performance Evaluation Results

Trace-driven simulations are performed to validate the effectiveness of the proposed buffer cache management policy. We run various applications on an Android mobile device, which consists

of Quad core ARM Cortex-A9 with Android 5.0 Lollipop (Linux kernel 3.4.0) installed. Ext4 file system in a metadata journaling mode is used for file system journaling and SQLite 3.7.10 is used as a journaling library. We set the size of a buffer cache block to 4KB as usual operating systems [12].

Based on these experimental setup, file access traces were captured while executing various applications. With these extracted traces, simulations are executed by replaying the traces. Since the traces were captured at the upper layer of buffer cache, they collect all file access requests including those of buffer cache hits as well as actual storage accesses. Our traces have the physical time of each file access request, and hence one can reproduce the results with the exact time of each request. Note that these experiments provide fair evaluation compared with the direct running of real applications each time. The reason is that applications are not possible to be executed with the same user interaction and buffer cache state for each application run, making fair comparison difficult. However, simulations with actual buffer cache traces can reproduce the real workloads with the same conditions.

Six application traces are used in our experiments: five of them are single application traces collected while executing browser, SNS, game, map, messenger, and one of them is a mixed trace collected while executing multiple applications together. The duration of the trace capturing period is 15-20 minutes.

Fig. 5 shows the cache hit ratio of the proposed policy in comparison with the original system that does not evict single-access blocks earlier than their eviction time. Note that the original system selects eviction blocks by the least recently used (LRU) cache replacement policy and writes updated blocks to storage every 5 seconds as usual Android environments with the Ext4 file system. Our policy also uses the same default algorithm but it removes a single-access block when it is written to storage. As shown in Fig. 5, the proposed policy significantly improves the cache hit ratio in all cases. The enhancement becomes significant as the cache size is small.

Buffer cache replacement policies are interesting research topic, which have been extensively studied. However, cache replacement is totally orthogonal to our policy since the main focus of cache replacement is on the eviction of cache blocks when free buffer cache space is exhausted while the focus of our research is on performing an early eviction of cache blocks upon periodic write operations. Actually, regardless of any cache replacement policies being adopted, periodic write operations are performed independently. Thus, our work can be incorporated into any cache replacement policies, and then evicts data that has incurred single-write from the cache upon periodic write operations.

As DRAM memory cells should be consistently refreshed although there is no read/write request on that cells, systems with a large buffer cache consume considerable battery power. Our experimental results show that reducing the buffer cache size would be possible without performance degradations when our policy is incorporated into the buffer cache system. Specifically, we can observe that the size of the buffer cache can be reduced to 1/2 to 1/4 without performance degradations of the buffer cache.

5. Conclusion

In this paper, we found out that a large proportion of mobile application's file accesses exhibit single-write characteristics. Based on this observation, we presented a new buffer cache management policy that classifies single-write blocks and evicts them from the buffer cache aggressively when the blocks are being written to storage. This has the effect of extending the effective buffer cache space, leading to significant improvement of the cache hit ratio. Simulation results with various real application traces showed that the proposed buffer cache management policy enhances the performance of mobile system's buffer cache by 20% on average. We also showed that our policy can reduce the buffer cache size significantly without performance degradations, which would allow for the reduction of energy consumption in mobile systems significantly.

6. Acknowledgment

This work was supported by the Basic Science Re-search program through the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2016R1A2B4015750).

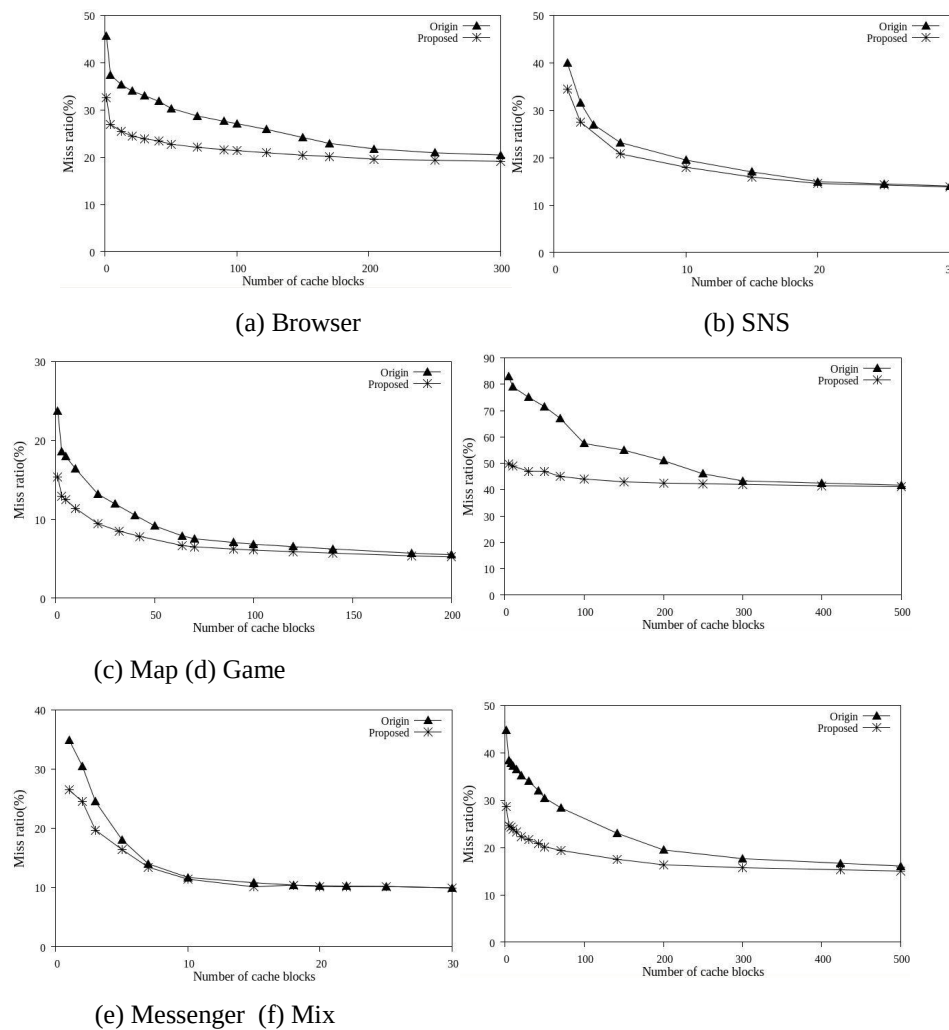


Fig. 5 Cache miss ratio of the original and the proposed policies as a function of the cache size.

References

- [1] Lee, E. and Bahn, H., 2014. Caching strategies for high performance storage media. *ACM Transactions on Storage*, 10(3), pp. 1-22.
- [2] Park, J., Lee, E. and Bahn, H., 2012. DABC-NV: A buffer cache architecture for mobile systems with heterogeneous flash memories. *IEEE Transactions on Consumer Electronics*, 58(4), pp. 1237-1245.
- [3] Lee, S., Bahn, H. and Noh, S.H., 2014. CLOCK-DWF: a write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. *IEEE Transactions on Computers*, 63(9), pp. 2187-2200.
- [4] Lee, E., Bahn, H. and Noh, S.H., 2013. Unioning of the buffer cache and journaling layers with non-volatile memory. In *USENIX Conference on File and Storage Technologies* (pp. 73-80). USENIX.
- [5] SQLite, <https://www.sqlite.org>.

- [6] Oh, G., Kim, S., Lee, S. and Moon, B., 2015. SQLite optimization with phase change memory for mobile applications. *PVLDB*, 8(12), pp. 1454-1465.
- [7] Riska, A. and Riedel, E., 2006. Disk drive level workload characterization. In *USENIX Annual Technical Conference* (pp. 97-102). USENIX.
- [8] Bansal, S. and Modha, D.S., 2004. CAR: clock with adaptive replacement, In *USENIX Conference File and Storage Technologies* (pp. 187-200). USENIX.
- [9] Liu, S., Pattabiraman, K., Moscibroda, T. and Zorn, B., 2011. Flickr: saving DRAM refresh-power through critical data partitioning, In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 213-224). ACM.
- [10] Zhou, Y. and Philbin, J.F., 2001. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference* (pp. 90-104). USENIX.
- [11] Megiddo, N. and Modha, D.S., 2003. ARC: a self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies* (pp. 115-130). USENIX.
- [12] Yoo, S. and Bahn, H., 2016. An efficient page replacement algorithm for PCM-based mobile embedded systems. In *IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* (pp. 183-188). IEEE.

Authors Biography



Hyunkyung Choi received the BS degree in computer science and engineering from Kangwon National University in 2008 and the MS degree in computer science and engineering from Ewha Womans University in 2011, respectively. She is currently a PhD candidate of computer science and engineering at Ewha Womans University, Seoul, Republic of Korea. Her research interests include operating systems, caching algorithms, storage systems, embedded systems, system optimizations, and real-time systems.



Dohee Kim received the BS and PhD degrees in computer science and engineering from Ewha University, Seoul, Republic of Korea, in 2012 and 2017, respectively. She is currently an assistant professor of electrical and computer engineering at Ajou University, Suwon, Republic of Korea. Her research interests include smart devices, operating system, real-time systems, non-volatile memory, and high-performance computing.



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.

