FS-LRU: A Page Cache Algorithm for Eliminating `fsync` Write on Mobile Devices

Dong Hyun Kang† and Young Ik Eom†
†Sungkyunkwan University, South Korea
{kkangsu,yieom}@skku.edu

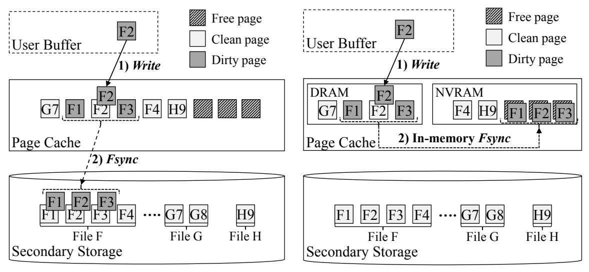
***Abstract*—In mobile environments, frequent `fsync` system calls negatively impact both performance and endurance of NAND-based storage. In this paper, we propose a novel page cache algorithm, called FS-LRU, that reduces the negative effect of `fsync` call by using hybrid DRAM and NVRAM memory architecture. Our experimental results show that FS-LRU outperforms the traditional LRU by up to 39.5%.**

I. Introduction

Unfortunately, mobile devices (e.g., smart phones and tablets) suffer from excessive write operations because applications frequently call `fsync` system call to update their database files with crash consistency. Accordingly, many engineers and researchers focused on mitigating the overhead of `fsync` call. Early work investigated the synchronization cost of I/O path on smart phones [1], [2] and a recent work has focused on the modifications of SQLite [3] considering the following issues. First, most mobile applications employ SQLite as their database. Second, SQLite negatively impacts both performance and endurance of NAND storage (e.g., eMMC) when it operates in *rollback journal mode* since SQLite invokes three `fsync` calls for each transaction: (i) `fsync` followed by a write for rollback journal, (ii) `fsync` followed by a write for journal header, and (iii) `fsync` followed by a write for database.

Our work is motivated to answer the simple question: *can we reduce the negative effects of* `fsync` *call on the page cache layer?* In this paper, we present a novel page cache algorithm, FS-LRU (`fsync`-aware LRU), that adopts a hybrid DRAM and NVRAM memory architecture to reduce the explicit synchronization cost with `fsync` calls. Unsurprisingly, some researchers exploited NVRAM as main memory to gain its benefits (e.g., non-volatility, byte-addressability, low power consumption, and high density) and they proposed a page cache algorithm for mitigating journaling overhead [4]. They improve overall performance by providing journaling functionality on a page cache layer. However, this approach has its limitations in mobile environments since frequent `fsync` calls incur a huge Copy-On-Write (COW) overhead. Our algorithm is different in that FS-LRU is complementary to existing journaling mechanisms, such as those of Ext3, Ext4, and XFS, since it leaves the journaling mechanism. For trace-driven simulation, we have implemented a prototype of our algorithm and compared its performance against two other algorithms. Our experimental results indicate that FS-LRU shows comparable hit ratio and significantly reduces the number of write operations issued to storage by up to 39.5% for real mobile workloads.

This investigation was financially supported by Semiconductor Industry Collaborative Project between Sungkyunkwan University and Samsung Electronics Co. Ltd. Young Ik Eom is the corresponding author of this paper.



(a) Traditional `fsync` process (b) In-memory sync process

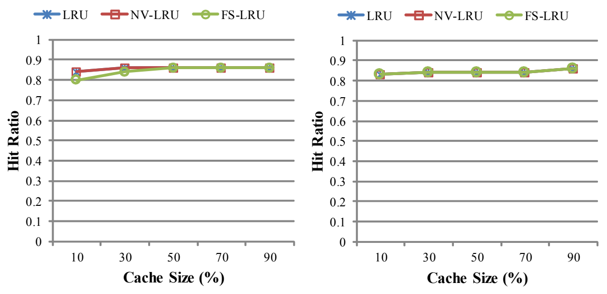
Fig. 1: Comparison between the `fsync` and in-memory sync

II. How do `fsync` impact the performance on mobile devices?

In this section, we study the journaling mechanism of SQLite for supporting the atomicity of transactions and the overhead of `fsync` system call. The *rollback journal mode* of SQLite is commonly used to update database files on mobile devices. However, this mode significantly degrades not only overall performance but also endurance of NAND-based storage because it issues buffered pages to storage whenever `fsync` call is triggered by an application. The first `fsync` is triggered after copying original data into the journal file (`.db-journal`), and then the second `fsync` is immediately triggered to reflect journal header into the journal file (`.db-journal`). Finally, the third `fsync` is triggered after recording up-to-date data into the database file (`.db`). To better understand the overhead of `fsync` call, we have collected real mobile workloads on a Google Nexus 7 tablet and observed that the second `fsync` of rollback journal mode generates significant amount of extra writes to storage even though only a small number of bytes are modified for journal header (e.g., 12 bytes). This observation strongly motivates us to propose *in-memory delta sync* scheme.

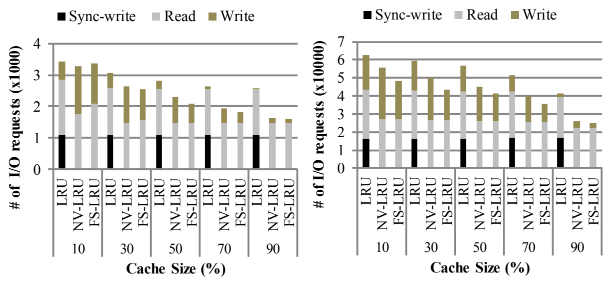
III. Design of FS-LRU

Most modern file systems provide `fsync` system call to immediately reflect in-memory data to storage. However, frequent `fsync` calls cannot benefit from buffered I/O because SQLite must wait until all dirty pages, which are related to the target file of the `fsync` call, are flushed to storage (called `fsync` write). In this paper, we propose a `fsync`-aware page cache algorithm, called FS-LRU, that efficiently eliminates some `fsync` writes by employing NVRAM as both page cache and persistent storage media. Figure 1 demonstrates the traditional `fsync` call process with that of the FS-LRU. In this figure, the traditional `fsync` simultaneously writes three dirty pages (F1, F2, and F3) related with the file F to the storage. On the other hand, FS-LRU simply copies those pages on DRAM into NVRAM



(a) Game (b) Mixed

Fig. 2: Cache hit ratios



(a) Game (b) Mixed

Fig. 3: The amount of generated read/write requests

to make them persistent, and then we clear the clean bits of the copied pages (F1, F2, and F3) and set the sync bits of them for replacement policy. If a synced page (i.e, the page with sync bit set) on DRAM is written again, we change its state to dirty and record the update information (e.g., the offset and length) for *in-memory sync*. When `fsync` is triggered, FS-LRU performs either *in-memory sync* or *in-memory delta sync* according to the update information of a synced page. If more than half of a page is updated, FS-LRU maintains the page on NVRAM as a log page and copies the up-to-date page on DRAM into NVRAM (called *in-memory sync*). Otherwise, FS-LRU persistently store the original part, which is to be modified, and then reflects the up-to-date data to the original part on NVRAM at byte-level granularity (called *in-memory delta sync*). Consequently, we eliminates the amount of extra writes as mentioned in Section II.

A. Page Replacement Policy

To perform the *in-memory sync*, FS-LRU separately manages the pages in DRAM and NVRAM with two least recently used (LRU) lists. If a page fault occurs, we place the faulted page on DRAM because it generally does not need to be persistently stored. When it is necessary to reclaim a page on DRAM, we first select a victim page with LRU policy and try to migrate the victim page to NVRAM. However, if the victim page has clean state and its sync bit is 1, the page is discarded since the sync bit means that the up-to-date page was persistently stored on NVRAM by previous `fsync` call. After discarding the page on DRAM, when the persistently stored page on NVRAM is written again, FS-LRU copies the page at the most recently used (MRU) position on DRAM to keep its persistency. If a page replacement is necessary on NVRAM, FS-LRU prefers to evict a page which is most unlikely to be synced again, in order to increase the possibility of the *in-memory delta sync*.

IV. Performance Evaluation

To verify our algorithm, we implemented a trace-driven cache simulator and compared its performance with the traditional LRU algorithm. We also compared FS-LRU with NV-LRU, which employs only NVRAM as a page cache and provides only *in-memory sync* at the page-level granularity, to confirm the effectiveness of *in-memory delta sync*. For evaluation, we have modified the Android Kitkat to collect real application traces. The game workload was collected by running the Angry Birds and the mixed workload was collected while running multiple mobile applications, such as Facebook, Chrome, Hang-out, etc. For comparison, we also measured the total memory footprint of each workload. Based on the footprint, we run the cache simulator by varying the cache size from 10% to 90%. Since FS-LRU separately manages pages in DRAM and NVRAM, we set the ratio of the size of DRAM to that of NVRAM to be 1:9 whose configuration can show the best performance of our algorithm. We first compared the cache hit ratio because FS-LRU sometimes violates the page cache philosophy by placing a page on both LRU lists. Figure 2 confirms that FS-LRU shows comparable hit ratios to other algorithms even though it simultaneously places a page on DRAM and NVRAM. This is because mobile workloads have a small working set size (i.e., in the case that the cache size is 10%, most pages can be referenced with a cache hit). To investigate how each algorithm impact the overall performance, we measured the amount of write requests generated from each of the algorithms (Figure 3). It clearly confirms that the amount of write requests from LRU are the largest in all cases because LRU periodically issues write requests caused by `fsync` calls. On the other hand, FS-LRU and NV-LRU show lower write counts than LRU by up to 39.5% due to *in-memory sync*. In addition, since FS-LRU also supports the *in-memory delta sync*, which optimizes the amount of writes, it shows the lowest write counts in most cases. This results show that FS-LRU efficiently eliminates the overhead of `fsync` writes by performing either *in-memory sync* or *in-memory delta sync*.

V. Conclusion

In this paper, we introduced the hybrid memory architecture that can give the benefit of low power consumption because NVRAM consumes only a small amount of energy in idle-state. We also proposed FS-LRU cache algorithm to reduce the negative effects of `fsync` calls. Our experimental results confirm that FS-LRU improves the performance compared to LRU scheme. We also believe that the key idea of FS-LRU can be widely adopted for other environments because SQLite is widely used as embedded database engine.

References

[1] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O Stack Optimization for Smartphones," in *Proc. of the USENIX ATC*, 2013.

[2] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting Storage for Smartphones," in *Proc. of the 10th USENIX FAST*, 2012.

[3] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split," in *Proc. of the 12th USENIX FAST*, 2014.

[4] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory," in *Proc. of the 11th USENIX FAST*, 2013.