Flash-Friendly Buffer Replacement Algorithm for Improving Performance and Lifetime of NAND Flash Storages [*]

Dong Hyun Kang[†], Changwoo Min[†*], Young Ik Eom[†]
[†]*Sungkyunkwan University, Korea*
[*]*Samsung Electronics, Korea*
{kkangsu, multics69, yieom}@skku.edu

1 Introduction

Buffer replacement algorithms have been actively researched for decades because it receives I/O requests directly from applications and transforms the requests into desirable I/O patterns for storage devices. Traditional replacement algorithms such as LRU and CLOCK exploit the *temporal locality* to hide disk-seek latency and evict page which are most unlikely to be accessed in the future, minimizing the number of slow I/O operations. However, traditional algorithms are inappropriate for NAND flash storages, such as eMMCs, microSD cards, and SSDs, because performance characteristics of the NAND flash storages are quite different to those of HDDs. First, *limited program/erase (P/E) cycles*: P/E cycles of multi-level cell (MLC) is roughly 3K. Second, *asymmetric read and write cost*: write operations are much slower than read operations. Third, *no in-place update*: FTL takes log-structured approaches since a flash block must be erased before writing a page that belong to the block and it involves garbage collection (GC) operations to move valid pages to new blocks. Finally, *performance variability in write patterns*: random write patterns are significantly slower than sequential write patterns because random write patterns cause more fragmentation in FTL and thus it drops both performance and lifetime of the NAND flash storage by increasing write amplification factor (WAF).

Previous flash-aware buffer replacement algorithms [2, 3, 4] can be largely classified into two: First, CFLRU [4] exploits the *asymmetric read and write cost* and it prefers to evict clean pages over dirty pages to reduce more expensive write operations. Second, some other schemes [2, 3] exploit the *performance variability in write patterns*. FAB [2] selects a block including the largest number of pages and evicts all pages that belong to the block for generating sequential write patterns. Recently proposed Sp.Clock [3] modifies Clock algorithm to maintain pages by their sector numbers rather than recency order and evicts pages in the order of their sector numbers.

[*]This work was supported by the IT R&D program of MKE/KEIT. [10041244, SmartTV 2.0 Software Platform]

We propose a novel buffer replacement algorithm, called TS-CLOCK (Temporal and Spatial locality-aware CLOCK) to improve the performance and lifetime of NAND flash storages.

2 Our Approach: TS-CLOCK

To improve the performance and lifetime of NAND flash storages, we designed TS-CLOCK algorithm to reduce the GC cost caused by the fragmentation in FTL. We propose three techniques exploiting temporal and spatial locality:

Cache hit ratio: we extends Clock algorithm, which exploits *temporal locality*, for high cache hit ratio. Though Sp.Clock [3] shows comparable cache hit ratio to Clock algorithm in mobile workloads, it shows low hit ratio in server workloads because Sp.Clock distorts the recency by sorting pages in order of their sector numbers (Figure 1a). In contrast, since TS-CLOCK maintains pages in recency order, it shows high cache hit ratio under various workloads.

Clean-first eviction: TS-CLOCK prefers to evict clean pages over dirty pages since evicting dirty pages generates slow write operations and hurts the lifetime of NAND flash storages. CFLRU [4] also adopts clean-first policy. However, it can cause the fragmentation in FTL and it leads to performance degradation because it randomly evicts dirty pages according to LRU policy.

Flash-friendly eviction: To solve the fragmentation problem, we shape evicted dirty pages to *flash-friendly write patterns*. FAB [2] and Sp.Clock [3] also mitigate the fragmentation. However, FAB [2] causes unnecessary operations because it performs eviction at the granularity of a block and Sp.Clock [3] suffers from high GC cost for workloads with wide I/O ranges. TS-CLOCK selects a block with the largest number of dirty pages, which are least likely to be accessed, and then sequentially evicts pages in that block, generating *flash-friendly write patterns*.

TS-CLOCK follows the basic rule of traditional Clock algorithm with three differences. First, TS-CLOCK manages a sorted list of dirty pages for each block number. Second, TS-CLOCK maintains a reference count per page rather than a reference bit. If an accessed page is clean, TS-CLOCK always sets its reference count to one. Otherwise, its reference count is determined by its *update likelihood*. For all dirty pages in the same block, their update likelihood is calculated as the ratio of the number of dirty pages, whose reference count is zero, to the total number of pages per block (i.e., a dirty page which belongs to a block with many dirty pages is considered likely to be updated). As the update likelihood is higher, TS-CLOCK sets larger value to the reference count to keep pages with higher update likelihood in cache longer. For 25%, 50%, 75%, and 100% update likelihoods, TS-CLOCK sets a reference count to 1, 2, 3, and 4, respectively. Third, TS-CLOCK manages two hands, *t-hand* and *s-hand*, to select a victim page. When there is no free space, *t-hand* scans pages in a circular manner and checks the reference count of each page. If the reference count is greater than zero, it is decreased by one. Otherwise, TS-CLOCK considers the page pointed by *t-hand* as a victim candidate. If the victim candidate is clean, it is immediately evicted. Otherwise, *s-hand* scans dirty pages belonging to a block in order of sector number and selects an evicted dirty page instead of a victim candidate for flash-friendly eviction. If *s-hand* is not set or reaches the end of the block during the scan, *s-hand* is set to the first dirty page of the block that includes the victim candidate pointed by *t-hand*. Finally, TS-CLOCK evicts the selected page by *s-hand* and inserts a new page into the position of *t-hand*.

TS-CLOCK can significantly improve performance and lifetime of NAND flash storage without any hardware support by maintaining high cache hit ratio and evicting pages in *flash-friendly write patterns*.

3 Evaluation

For evaluation, we implemented a prototype of TS-CLOCK and four buffer replacement algorithms, Clock, CFLRU [4], FAB [2], and Sp.Clock [3], on Linux. For comparison, we follow the evaluation methodology used in Sp.Clock: (1) We collect the *before-cache traces*, which consist of I/O requests issued to the buffer cache, by running Dbench [1] benchmark that emulates server workloads. (2) For trace-driven simulations, we use the *before-cache traces* as the input of our cache simulator and collect *after-cache traces*, which are I/O requests generated by each replacement algorithm. (3) Finally, we replay the *after-cache traces* on a SSD manufactured by Samsung with the `O_DIRECT` option.

Figure 1 presents the simulated cache hit ratios as well as elapsed times measured on the Samsung SSD. Although TS-CLOCK shapes evicted pages to *flash-friendly write pattern*, cache hit ratios in Figure 1a clearly show that it maintains higher cache hit ratio than other algorithms. On the other hands, as expected,

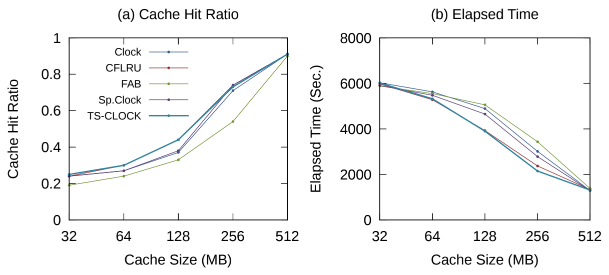

Figure 1: Cache hit ratio and elapsed time on SSD

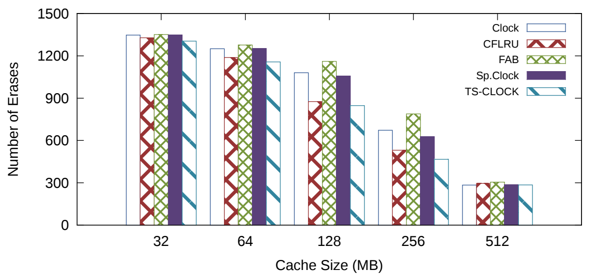

Figure 2: Comparison of erase count on page-level FTL

FAB [2] shows significantly low hit ratio due to its block-level eviction. Figure 1b shows that TS-CLOCK outperforms other algorithms. Especially, TS-CLOCK significantly outperforms the state-of-the-art algorithm, Sp.Clock, by up to 22.7%. That is because TS-CLOCK prefers to evict clean pages to minimize write operations as well as it evicts dirty pages *flash-friendly* to reduce GC cost. We also implemented a FTL simulator, which supports a page-level FTL scheme, to investigate how each replacement policy affects the lifetime of NAND flash storages. Figure 2 compares erase counts of each replacement algorithm. Our experimental result clearly presents that TS-CLOCK can extend the lifetime of NAND flash storages by up to 40.8%.

References

[1] The DBENCH web pages. `http://dbench.samba.org/`.

[2] Jo, H., Kang, J.-U., Park, S.-Y., Kim, J.-S., and Lee, J. FAB: flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics 52*, 2 (May 2006), 485–493.

[3] Kim, H., Ryu, M., and Ramachandran, U. What is a good buffer cache replacement scheme for mobile flash storage? In *Proc. of SIGMETRICS'12* (2012), ACM, pp. 235–246.

[4] Park, S.-Y., Jung, D., Kang, J.-U., Kim, J.-S., and Lee, J. CFLRU: A Replacement Algorithm for Flash Memory. In *Proc. of CASE'06* (2006), ACM, pp. 234–241.