An Efficient Buffer Replacement Algorithm for NAND Flash Storage Devices

Dong Hyun Kang, Changwoo Min, Young Ik Eom
Sungkyunkwan University
Suwon, Korea
Email: {kkangsu, multics69, yieom}@skku.edu

***Abstract*—NAND flash storage devices are now revolutionizing storage stacks. As their capacities are rapidly increasing, they are now being adopted virtually in all classes of computing devices, including mobile devices, desktop computers, and cloud servers. However, there are two remaining problems: first, as flash density increases, the lifetime of the storage medium decreases rapidly. Second, random writes significantly decrease performance and lifetime since they generate more *hidden writes* inside NAND flash storage devices. In this paper, we propose a novel buffer replacement algorithm called TS-CLOCK, to address those two problems. TS-CLOCK exploits temporal locality to keep the cache hit ratio high and also exploits spatial locality to maintain evicted writes *flash-friendly*. The key idea of our flash-friendly eviction is that, when evicting a dirty page, TS-CLOCK first selects a flash block with the largest number of dirty pages that are least likely to be accessed and then sequentially evicts pages in the block. Since it generates *pseudo sequential writes* for a flash block, it significantly increases performance and lifetime at once by reducing the number of hidden writes. We have implemented TS-CLOCK and compared it with seven replacement algorithms, including traditional and flash-aware ones, for several real workloads. Our experimental results show that TS-CLOCK outperforms the state-of-the-art replacement algorithm, Sp.Clock, by 30% on the NAND flash storage devices and extends the lifetime by 53%.**

***Keywords*-NAND flash storage; buffer replacement; temporal and spatial locality**

I. Introduction

NAND flash storage devices are rapidly replacing hard disk drives (HDDs) and they are now prevalent in all classes of computing devices including mobile devices, laptops, and servers. Manufacturers of NAND flash storage devices are driving their technologies and products toward higher densities at lower cost to compete with HDD products. To enable high densities, they scale down NAND flash chips and also leverage technologies to store multiple bits in each cell. Retail products that use single-level cell (SLC, 1 bit/cell), multi-level cell (MLC, 2 bits/cell), or triple-level cell (TLC, 3 bits/cell) technologies are already available in the market [1], [2]. Recently, technologies that enable 6-bits-per-cell have been demonstrated [3]. Unfortunately, increasing flash capacity by storing additional bits per cell reduces the flash chip's lifetime by orders of magnitude [4]. In 2x nm cells in the market, SLC, MLC, and TLC have roughly 100K, 3K, and 1K program/erase (P/E) cycles respectively [5]. In addition to the cell types, write patterns also heavily affect lifetime: random writes induce more *hidden writes* in a storage device than sequential writes do, and thus they significantly reduce performance and lifetime [6]–[8].

Over the last few decades, the primary focus of storage researchers has been to hide I/O latency, and many studies are based on the (frequently unwritten) assumption that the performance of HDDs is limited by their mechanical operations including disk arm movement and disk platter rotation. However, NAND flash storage devices are built on semiconductor chips with no mechanical parts, so their performance characteristics are quite different to those of HDDs: first, read operations have potentially uniform random access time. Second, I/O latencies among read and write operations are asymmetric; write operations are much slower than read operations. Finally, write performance is highly dependent on *write patterns* and eventually determined by the garbage collection cost of the Flash Transaction Layer (FTL) [6]–[8]: sequential writes are several times faster than random writes. Not surprisingly, existing storage stacks including file systems [8]–[11], buffer replacement algorithms [12]–[18], and IO schedulers [19], [20] have been carefully revisited.

The OS buffer cache receives I/O requests directly from applications and transforms the requests into a suitable I/O request stream for storage devices. We focus on the buffer replacement algorithm used by the OS buffer cache due to the algorithm's potential to shape I/O requests. The primary goal of the buffer replacement algorithm is to ensure a high hit-ratio and to reduce the amount of slow I/O operations. The traditional, but flash agnostic, approaches such as Least Recently Used (LRU), CLOCK [21], and Linux2Q [22] exploit *temporal locality* to predict future accesses of the pages and evict the pages which are the most unlikely to be accessed in the future. Though the majority of buffer replacement schemes are based on temporal locality, there are several schemes which exploit *spatial locality* in the context of HDDs [23], [24] and NAND flash storage devices [12]–[18]. In flash aware schemes, while much attention has been focused on improving performance, lifespans improvement has not been given much attention, unfortunately. CFLRU [12], LRU-WSR [14], and FOR [15] exploit asymmetric read and write operation times and evict clean pages over dirty pages to reduce the more expensive write operations. However, since they do not consider the write patterns of the evicted pages, the generated write patterns could induce a large number of hidden writes inside NAND flash storage devices depending on workload characteristics.

In contrast, FAB [13], BPLRU [17], and LB-CLOCK [18] give more focus to generating flash-friendly write patterns in order to reduce garbage collection (GC) costs and hidden writes inside NAND flash storage devices. However, since they evict multiple pages in a unit of a flash block, they suffer from the *early eviction problem* [25] – their coarse-grained policies evict pages with high locality and such pages are re-accessed and re-evicted after all– and the lower hit ratio results in more I/O operations and decreased performance and lifetime. Recently, Kim et al. [16] proposed Sp.Clock, which is a variant of the CLOCK algorithm and manages page frames by logical sector number rather than recency order, in a manner similar to WOW [24] in HDDs. The sorted eviction scheme could contribute to lower hidden writes in NAND flash storage devices when the I/O range is narrow. However, it suffers from high GC cost for workloads with wide I/O ranges, because the evicted write patterns become random rather than sequential. Moreover, Sp.Clock does not exploit asymmetric read and write operation times to select victim pages.

In this paper, we propose a novel buffer placement algorithm, which we named TS-CLOCK (Temporal and Spatial locality aware CLOCK), for NAND flash storage devices. We aim to improve performance and expand lifetime by reducing the number of write operations and generating flash-friendly pseudo sequential write patterns, which minimize hidden writes inside NAND flash storage devices. To this end, TS-CLOCK makes use of *asymmetric read and write operation times* as well as *logical flash block utilization*, which is the number of dirty pages in a logical flash block. Throughout this paper, we use the term *block* for logical flash block, which is calculated from dividing a logical sector number by the number of pages per flash block, for brevity.

Key features of the TS-CLOCK algorithm are derived from the characterization of write performance varying write patterns in NAND flash storage devices, which is our first contribution. Though several replacement schemes consider write patterns, they largely adopt relatively simple policies such as block-level eviction [13], [17], [18] or sorted eviction [16]. In order to characterize write performance to write patterns, we perform trace-driven simulations for various synthetic write patterns. We observe that, when writing dirty pages in a block by the sector number, writing a fewer number of blocks with more dirty pages generates fewer hidden writes. Therefore, *flash-friendly write patterns*, which reduce the GC cost and thus improve performance and lifetime, are *pseudo sequential patterns* that have high block utilization. To our knowledge, this is the first study that experimentally analyzes the relationship between write performance and write patterns in NAND flash storage devices. We expect that our observations can be applicable in other areas for optimizing write performance and lifetime in NAND flash storage devices.

Our other contribution is the extensive evaluation of TS-CLOCK, and its comparison with seven state-of-the-art replacement algorithms including flash agnostic and flash aware schemes. To measure the performance of each algorithm, we performed our experiments on three commercial NAND flash storage devices. In addition, to analyze how each algorithm affects lifetime, we performed our experiments on the two most widely used FTL schemes. As a result, TS-CLOCK outperforms the other algorithms by up to 90% and expands lifetime by up to 53%.

The rest of this paper is organized as follows. Section II is an overview of the characteristics of NAND flash storage devices and introduces flash-friendly write patterns. Section III describes the TS-CLOCK algorithm in detail. Then, Section IV presents the results of our evaluation. In Section V, we briefly survey previous related work and then we conclude this paper in Section VI.

II. What Are Flash-Friendly Write Patterns?

In this section, we investigate flash-friendly write patterns in order to design buffer replacement algorithms that improve performance and lifetime. We first overview NAND flash storage devices and then discuss our experimental results in order to characterize write performance varying write patterns.

A. NAND Flash Storage Devices

NAND flash storage devices (such as eMMC, microSD, and SSD) are composed of host interface logic, an array of NAND flash memory, and a controller.

In NAND flash memory, *read* and *write* operations are performed at the unit of a *page* (e.g., 4KB or 8KB) and write operations take typically about 10 times longer than read operations. Before overwriting any page, a whole *block* (composed of 64 - 128 pages) must be erased, and erase operations also take about 10 times longer than write operations. Also, each flash memory cell has limited program/erase (P/E) cycles and the number of maximum P/E cycles sharply drops as the density of the cell increases [4], [5].

In order to hide peculiarities of NAND flash memory, a NAND flash controller runs a *flash translation layer* (FTL). Due to having no in-place overwrites allowed in NAND flash memory, a FTL takes a log-structured approach: rather than modifying existing data, the previous data is invalidated and a new copy is written. A FTL manages a *mapping table* from a logical sector number to a physical page address in NAND flash memories and performs *garbage collection* (GC) to recycle invalidated physical pages. According to their mapping granularity, FTLs can be largely classified into *block-level FTL* [26], *page-level FTL* [27], [28], and *hybrid FTL* [29], [30]. Block-level FTL and page-level FTL are self-explanatory. Hybrid FTL logically partition blocks into data blocks and log blocks and manage data blocks in block-level mapping and log blocks in page-level mapping. There is a trade-off between the required RAM size for the mapping tables and garbage collection overhead. Low-end flash storage devices such as microSD cards and eMMCs adopt hybrid FTL schemes due to their smaller RAM requirements. In contrast, high-end flash storage devices such as SSDs adopt page-level FTL schemes due to their higher performance with lower garbage collection overhead.

B. *Flash-Friendly Write Patterns*

Garbage collection (GC) is universal to all FTL schemes that perform out-of-place writes. Since GC involves moving valid pages in victim blocks to new locations, it generates hidden writes and fundamentally limits write performance in NAND flash storage devices [6]–[8], [31], [32]. The efficiency of GC is generally measured in terms of the write amplification factor (WAF) – the ratio of total internal writes including copying induced by GC to externally requested writes. Since higher WAF means that more hidden writes are generated during GC, it negatively impacts performance and lifetime.

Modeling flash write performance is non-trivial because various factors including write pattern, I/O range, GC policy, and the amount of free space interact with each other non-linearly. To characterize flash write performance, previous studies use various techniques: black-box testing using heuristically generated patterns [6], [33], [34], mathematical modeling [7], [31], [32], and statistical machine learning [35]. Unfortunately, they mostly focus on relatively simple patterns such as completely sequential writes, uniform random writes and stripe writes. Thus, to optimize flash write performance, system designers rely on heuristics such that (1) sequential writes are much faster than random writes, (2) random write performance converges with sequential write performance when requests are aligned in block size and their sizes are also in block size [8], [34], and (3) the performance of random writes in a narrow I/O range is higher than that in a wide I/O range [36].

Buffer replacement algorithms are responsible for shaping write requests from applications into desirable write patterns for NAND flash storage devices. Therefore, we need to understand *"what flash-friendly write patterns are"* and evict dirty pages for them to form such flash-friendly write patterns. To do this, we formulate a hypothesis that, *when writing dirty pages in a block by the sector number, the WAF becomes smaller as the block utilization (i.e., the ratio of dirty pages in a block) becomes higher*. Since writing more pages to a block is likely to invalidate more pages in that block, GC can easily select a victim block with few valid pages and thus can minimize the overhead of copying the valid pages. To confirm the hypothesis, we performed trace-driven simulations for synthetic traces with different block utilizations. The synthetic trace with $x\%$ block utilization is where we randomly select $x\%$ of pages in a randomly selected block and then issue write operations for the selected pages by the sector number. In addition, to investigate how I/O ranges affect the WAF in our traces, we generated synthetic traces with different I/O ranges. The traces were run on an FTL simulator with two representative FTL schemes (FAST FTL [29] as a representative hybrid FTL scheme and page-level FTL [27]).

Our experimental results in Figure 1 clearly show two things regardless of the FTL scheme used:

- As the block utilization of a trace increases, the WAF decreases. In all cases, traces with 100% block utilization have an ideal WAF, 1, with no hidden write overhead. In contrast, for traces with 25% block utilization, the WAFs significantly increase: 4.9 for FAST FTL and 3.5 for page-level FTL.
- As the I/O range of a trace decreases, the WAF decreases. For traces with 25% block utilization and 1 GB range, the WAFs of FAST FTL and page-level FTL are 2.3 and 1.02, respectively.

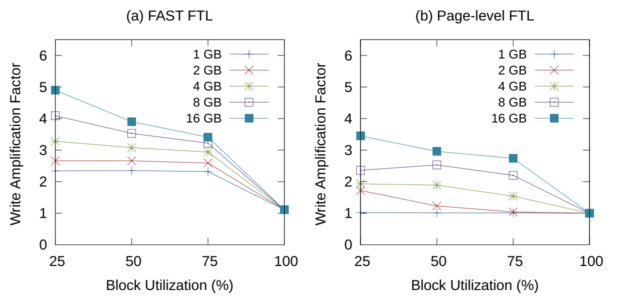


Fig. 1: Write amplification for synthetic traces with different block utilizations and different I/O ranges. In our FTL simulation, the capacity of the devices is 16 GB and the over-provisioning space is 2.4 GB (15%). See Section IV-E for a detailed description of the FTL simulator.

Since the I/O range is the characteristics of workloads, which replacement algorithms cannot control, we use the block utilization as our main design rationale behind the TS-CLOCK algorithm. For brevity, we use the term *pseudo sequential write patterns* or *flash-friendly write patterns*, which are write patterns with high block utilization. While we omitted experimental results on real NAND flash storage devices due to space limitations, throughputs on the real devices were inversely proportional to the WAFs in Figure 1.

III. The TS-CLOCK Algorithm

In this section, we present a novel replacement algorithm called TS-CLOCK for NAND flash storage devices. Guided by the unique performance characteristics of NAND flash storage devices described in Section II, the design objectives of the TS-CLOCK are stated as follows:

- Maximize cache hit ratio by exploiting temporal locality. To achieve high performance, it is important to reduce the number of I/O requests issued to NAND flash storage devices.
- Prefer to evict clean pages over dirty pages. Since write operations are slow and hurt the lifetime of a device, it is important to evict less dirty pages for higher performance and longer lifetime.
- Shape evicted dirty pages to flash-friendly write patterns which have high block utilization. The flash-friendly pseudo sequential patterns can minimize hidden writes inside NAND flash storage devices and thus improve performance and lifetime.

Now, let us explain the details of the TS-CLOCK algorithm based on the pseudo-code depicted in Figure 2. To exploit temporal locality and thus keep hit ratio high, TS-CLOCK extends a well-known CLOCK replacement algorithm [21]. CLOCK is an LRU approximation algorithm which maintains a reference bit. While at every cache hit LRU needs to move a page to the most recently used (MRU) position in the list, CLOCK only monitors whether a page has been recently referenced or not. When a page is accessed, the page unit hardware sets the reference bit of the page to 1. Whenever free page frames are needed, the *clock-hand*, which points to the last evicted page, scans pages in a circular list until a page with a reference bit of zero is found, and that page is then replaced. In the course of the scan, CLOCK clears a reference bit in every page to zero. After all, CLOCK replaces such pages that have not been referenced upon one rotation of the *clock-hand*. To grant one full life of a new page, CLOCK inserts the new page into the position of the evicted page.

TS-CLOCK maintains a *reference count* for each page rather than a reference bit to give each page a different level of opportunity to stay in the buffer cache. Similar to CLOCK, a reference count for each page is set to R when that page is accessed (Line 17, 21 - 22), and it is decremented by one when the *t-hand* (i.e., the *clock-hand* in CLOCK) sweeps the page (Line 45). A page with a reference count of R is granted to stay in the buffer cache for R times the rotation of the *t-hand*. Though a reference count of every clean page is always set to one (Line 16 - 17), that of every dirty page is determined by its update likelihood (Line 21 - 22). We approximately calculate the update likelihood of a page from the update likelihood of the block that belongs to it. The update likelihood of a block is calculated as the ratio of dirty pages, such that they belong to that block and their reference count is not zero, to the number of pages per block. If a page is likely to be updated, its reference count is set to large number for that page to stay longer in the buffer cache. For four equally divided ranges of update likelihood, 0 - 25%, 25 - 50%, 50 - 75 %, and 75 - 100%, we set a reference count to 1, 2, 3, and 4, respectively (Line 20 - 22, 46 - 47). Since a reference count of a dirty page can be set to larger than one, TS-CLOCK prefers to evict clean pages over dirty pages and, among dirty pages, it prefers to evict dirty pages that are unlikely to be updated. If a selected page with a reference count of zero is clean, it is immediately evicted (Line 29 - 30). Otherwise, TS-CLOCK re-selects a victim page using the *s-hand*, as we will explain in next paragraph. Note that our reference counting scheme can be efficiently implemented using paging unit hardware: when the *t-hand* scans pages in the circular list, TS-CLOCK resets a reference count of a page whose reference bit is one.

In order to shape evicted dirty pages to flash-friendly pseudo sequential patterns with high block utilization, TS-CLOCK manages the *s-hand* in addition to the *t-hand*. The *s-hand* scans dirty pages, which belong to the same block, by their sector numbers to maintain spatial locality of evicted dirty pages (Line 13 - 14, 31 - 43). TS-CLOCK replaces a dirty page via a two-step process: the *t-hand* selects a block which is unlikely to be updated (Line 32 - 33, 36 - 38) and then the *s-hand* finally selects a dirty victim page in the block (Line 35, 41 - 42). Let's consider that the *t-hand* selects a page with a reference counter of zero. If the selected page is clean, it is immediately evicted as we explained (Line 29 - 30). But, if it is dirty, the *s-hand* scans dirty pages by their sector numbers until a page with a reference count of zero is found (Line 13 - 14, 31 - 43). When the *s-hand* points to *null* (its initial value) or the last dirty page in a block, TS-CLOCK sets the *s-hand* to the dirty page that belongs to a block pointed by the *t-hand* and that is the first page in that block (Line 31 - 33, 36 - 38). In contrast to the *t-hand*, while the *s-hand* scans dirty pages, it does not decrement a reference count of each page. That is because a reference count is designed to reflect temporal locality. After evicting a page, TS-CLOCK inserts a

```
1   page *t−hand = null, *s−hand = null;
2
3   void TS−CLOCK(page *p) {
4     if (p is not in the buffer cache) {
5       if (the buffer cache is full) {
6         page *v = choose_victim();
7         evict v;
8       }
9       if (t−hand is null)
10        t−hand = p;
11      else
12        insert p into the position of t−hand;
13      if (p.status is dirty)
14        insert p into the sorted list of p.block;
15    }
16    if (p.status is clean)
17      p.reference_count = 1;
18    else {
19      if (p.reference_count is 0)
20        p.block.non_zero++;
21      p.reference_count = ceil(
22        4 * p.block.non_zero / number of pages per block);
23    }
24  }
25
26  page *choose_victim() {
27    while (true) {
28      if (t−hand.reference_count is 0) {
29        if (t−hand.status is clean)
30          return t−hand;
31        if (s−hand is null)
32          s−hand = the first page
33            in the sorted list of t−hand.block;
34        while (true) {
35          page *s = s−hand;
36          if (s−hand is the last of the block)
37            s−hand = the first page
38              in the sorted list of t−hand.block;
39          else
40            s−hand = the next page in the sorted list of the block;
41          if (s.reference_count is 0)
42            return s;
43        }
44      }
45      t−hand.reference_count−−;
46      if (t−hand.reference_count is 0)
47        p.block.non_zero−−;
48      t−hand = the next page of t−hand in the circular list;
49    }
50  }
```

Fig. 2: The pseudo-code of the TS-CLOCK algorithm

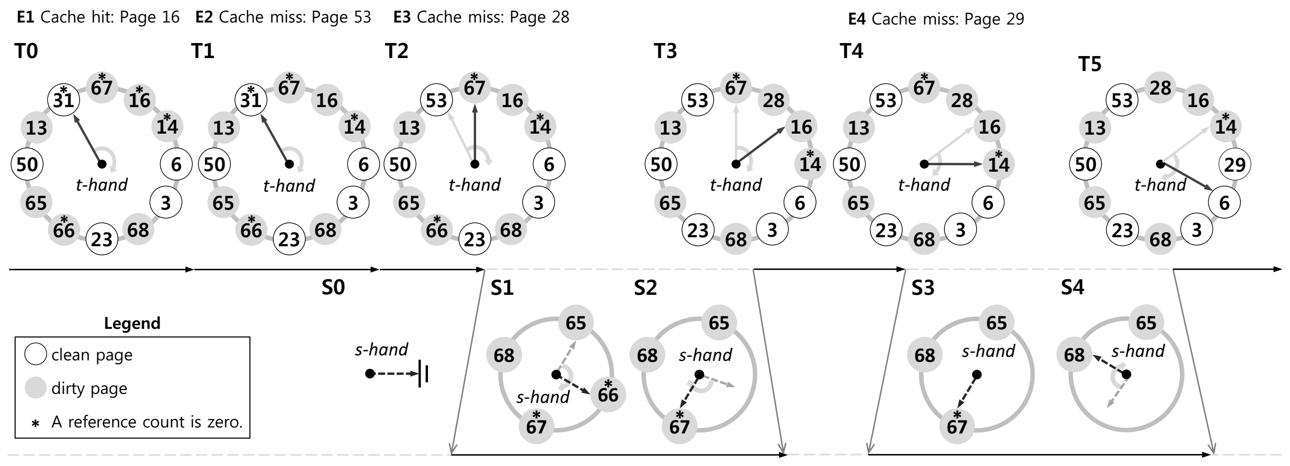


Fig. 3: An example of TS-CLOCK operations showing how TS-CLOCK handles events E1 - E4 from the initial status of T0 and S0. In this example, the size of a block is four pages.

new page into the position of the *t-hand* to grant at least one full life to the page (Line 12). In this way, TS-CLOCK can generate flash-friendly pseudo sequential write patterns with high block utilization and thus minimize hidden writes inside NAND flash storage devices.

Figure 3 illustrates an example of TS-CLOCK operations which show how TS-CLOCK handles events E1 - E4 from the initial status of T0 and S0. In this example, the size of a block is four pages. While accessing Page 16 (E1) at T0, a cache hit occurs and the reference counter of Page 16 is set to 2 (T1). Then, a cache miss occurs while accessing Page 53 (E2). Page 31, which is clean and is being pointed to by the *t-hand*, is replaced by a new page, Page 31 and now the *t-hand* moves to the next page in the circular list (T2). Next, a cache miss occurs while accessing Page 28 (E3). The *t-hand* selects Page 67, which is dirty, and then TS-CLOCK starts to scan the *s-hand* of Block 16, which Page 67 belongs to. Since the *s-hand* is initially *null* (S0), the *s-hand* is set to the smallest page in Block 16, which is Page 65 (S1). TS-CLOCKS sweeps the *s-hand* until a page with a reference count of zero is found. As a result, Page 66 is evicted (S2). Then, the new page, Page 28, is inserted at the position of the *t-hand* and TS-CLOCK moves the *t-hand* one step forward (T3). Finally, a cache miss occurs while accessing Page 29 (E4), and TS-CLOCK sweeps the *t-hand* to select a victim page and then selects Page 14 (T4). Since the selected Page 14 is dirty, TS-CLOCK uses the *s-hand* to select a dirty victim page. Since the *s-hand* already points to pages in Block 16 (S3), TS-CLOCK finds a victim page, Page 67, by sweeping the *s-hand*. The victim page pointed by the *s-hand* is evicted (S4), the new page, Page 29, is inserted (T5).

IV. Evaluation

In this section, we present the performance evaluation and analysis to assess the effectiveness of TS-CLOCK. We first compare the performance results of TS-CLOCK with those of well-known buffer replacement algorithms: LRU, CLOCK, Linux2Q, CFLRU, FAB, LB-CLOCK, and Sp.Clock.

	Storage-A	Storage-B	Storage-C
Manufacturer	Patriot	Adata	Samsung
Type	microSD card	microSD card	SSD
Capacity	16 GB	16 GB	120 GB
Erase Block Size	4MB	4MB	24MB
Flash Memory	MLC	MLC	TLC

TABLE I: Specification of the NAND flash storage devices

Workload	Read (MB)	Write (MB)	Write Range (MB)
Video Streaming	0.2	1514.4	266.6
Mixed Applications	526.9	413.2	133.0
File Server	7252.8	5880.1	14568.2
TPC-C	1129.2	2357.6	15124.4

TABLE II: Summary statistics of our test traces

A. Experimental Setup

For evaluation, we follow the evaluation methodology used by Kim et al. [16]. Our evaluation proceeds in three steps: first, we collect traces of read/write operations to the buffer cache from mobile and server workloads. We call this the *before-cache traces*. Second, we use the collected traces as the input of a simulator that implements the seven buffer replacement algorithms. The simulator produces a *cache hit ratio* and a storage access trace evicted by a buffer replacement algorithm, which we call the *after-cache trace*. Finally, the after-cache traces are replayed on top of real devices or on a FTL simulator. To measure the performance of each replacement algorithm, we replayed the after-cache traces on real devices in Table I with the `O_DIRECT` option. We also enable the Native Command Queuing (NCQ) to exploit the internal parallelism of the devices. To broadly exercise the replacement algorithms, we choose two microSD cards and a SSD such that their manufactures are different and their flash memory types include MLC and TLC. We measure the size of

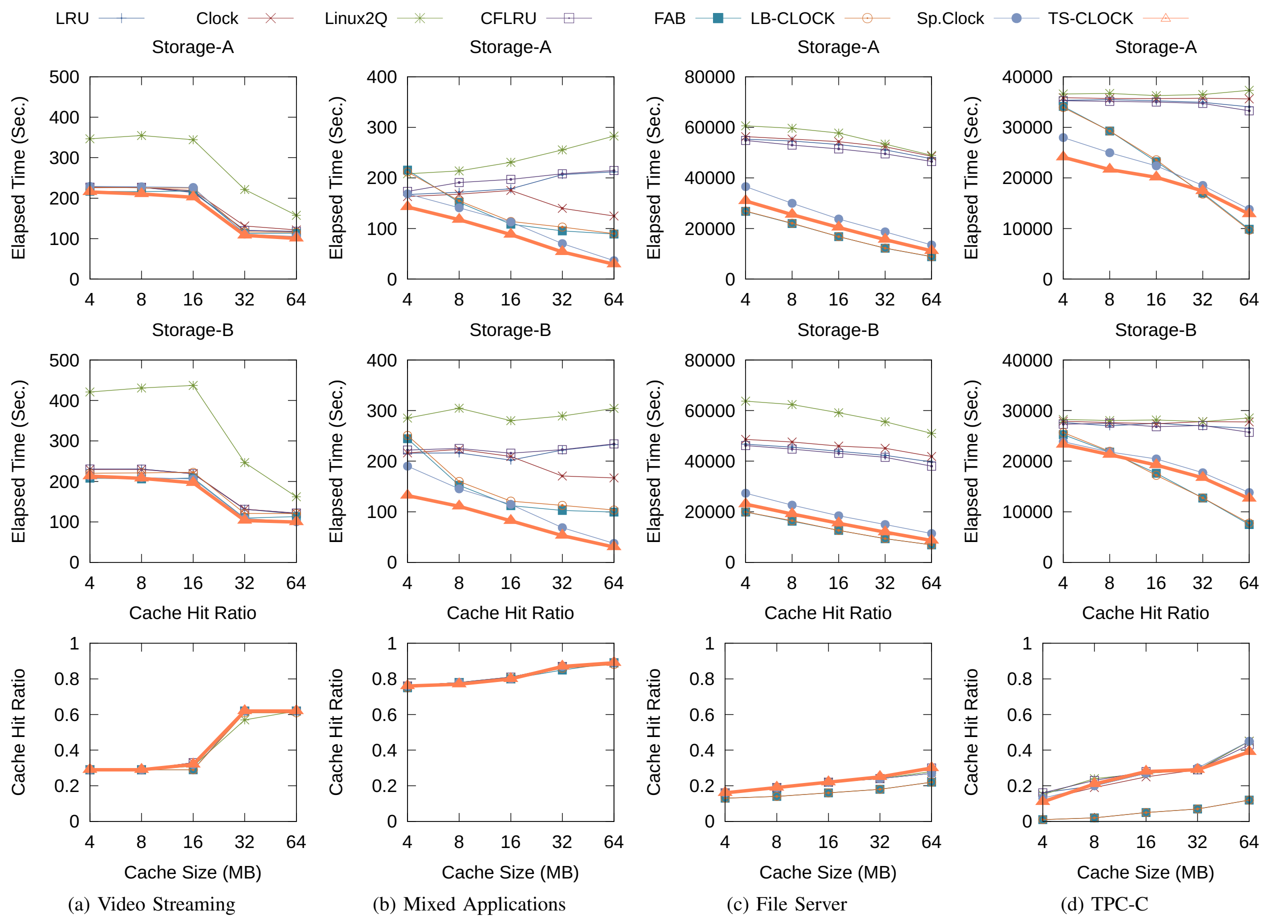


Fig. 4: Elapsed time and hit ratio on the microSD cards

the erase block in each NAND flash storage by following the measurement methodology in Kim et al. [34]. Also, to evaluate how each replacement algorithm affects lifetime, we replayed the after-cache traces on a FTL simulator. Our FTL simulator is trace-driven and supports FAST FTL and page-level FTL.

B. Evaluation Workloads

We used four before-cache traces: the two are workloads on mobile devices and the others are workloads on servers. Table II summarizes their statistics. The mobile workloads – video streaming and the mixed applications – are from Kim et al. [16]. These two workloads are collected from a real Android smart phone: the video streaming is collected while watching YouTube video and the mixed applications workload is collected while running multiple mobile applications, such as Facebook, Maps, Camera, Internet Browser, YouTube, Gallery, etc., for several hours. We modified the Linux kernel (version 3.2.0) to collect server traces. The file server workload is collected by running the Dbench benchmark [37], which simulates file server I/O. The TPC-C workload is collected while running the TPC-C benchmark [38] on a MySQL database system with 50 warehouses.

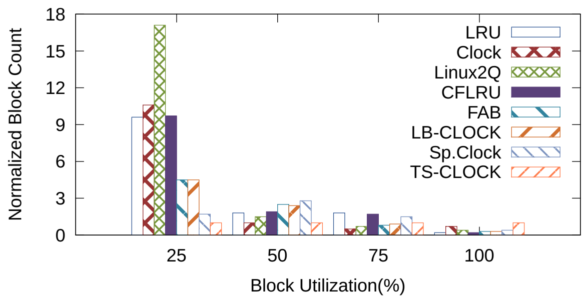
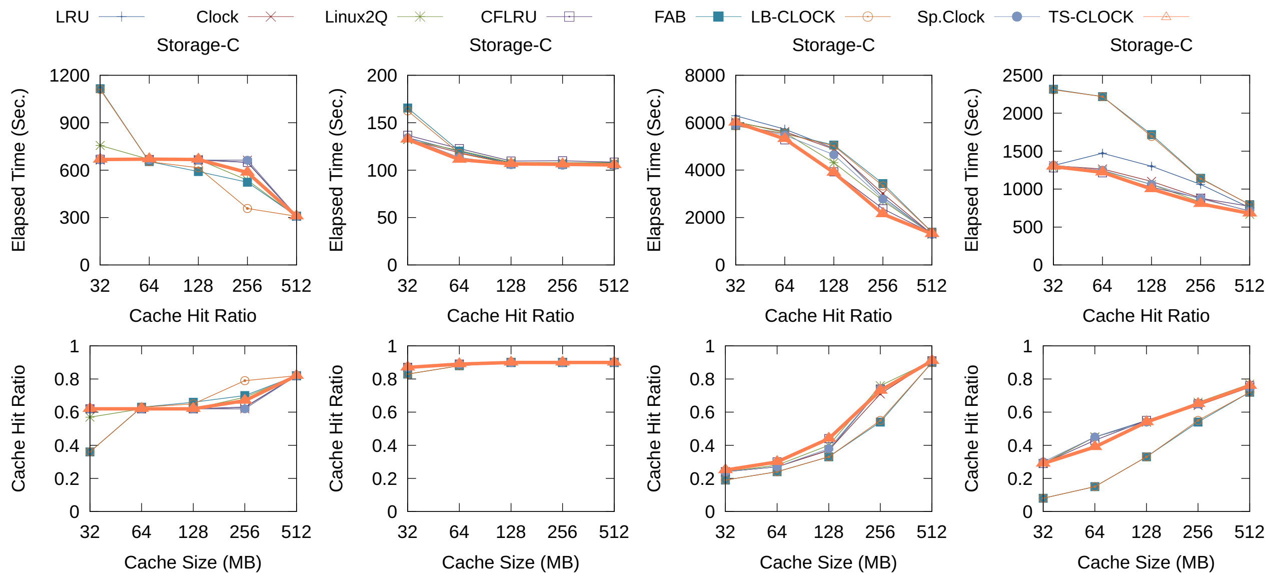


Fig. 5: Distribution of block utilizations for write operations in the mixed applications workload. Cache size and Block size are set to 4 MB.

C. Experiments on the MicroSD Cards

We first evaluate the performance of the eight replacement algorithms on the two microSD cards. Figure 4 presents the elapsed times and hit ratios of each replacement algorithm varying in buffer cache size from 4 MB to 64 MB. According to our measurements in Table I, we configured the block sizes of TS-CLOCK, FAB, and LB-CLOCK to 4 MB. Also, we



(a) Video Streaming (b) Mixed Applications (c) File Server (d) TPC-C

Fig. 6: Elapsed time and hit ratio on the SSD

set the window size of CFLRU as 25%, which is advised by the authors of the paper, for their static algorithm. As we described, we measured elapsed time by replaying a generated *after-cache trace* of each replacement algorithm on the microSD cards. Figure 4 show that TS-CLOCK maintains high cache hit ratios, which are comparable to those of traditional buffer replacement algorithms, in all cases. In addition, TS-CLOCK re-arranges the eviction order of dirty pages to form *pseudo sequential write patterns*, and thus it significantly outperforms the other algorithms. In the rest of this section, we will analyze results in more detail.

Mobile Workloads: For the video streaming workload, which has mostly sequential accesses, TS-CLOCK outperforms the other algorithms: on average 12.8% for Storage-A and 15.3% for Storage-B (Figure 4a). For the mixed applications workload, whose significant portions are random accesses, TS-CLOCK significantly outperforms the other algorithms: by up to 89.5% for Storage-A and 90.0% for Storage-B (Figure 4b). To understand why TS-CLOCK performs better, we analyze the distribution of block utilizations from the after-cache traces. We count the number of sorted write operations in the same block. Specifically, Figure 5 shows the distribution of the mixed applications workload. It clearly shows that TS-CLOCK issues significantly less number of write operations and it clusters more write operations for a block to generate flash-friendly write patterns.

Server Workloads: For the file server and TPC-C workloads, TS-CLOCK significantly outperforms the others, excluding FAB and LB-CLOCK, by up to 78.1% for the file server workload and 65.6% for the TPC-C workload (Figure 4c and 4d). That is because the server workloads are mostly random accesses and their I/O ranges are wider than those of the mobile workloads. While FAB and LB-CLOCK are slightly faster than TS-CLOCK, they issue significant write operations than TS-CLOCK does. That is because their cache hit ratios are significantly lower, the lowest among the eight algorithms, due to the *early eviction problem* caused by their coarse-grained buffer management. Since they issue significantly more write operations, they shorten the lifetime of the devices.

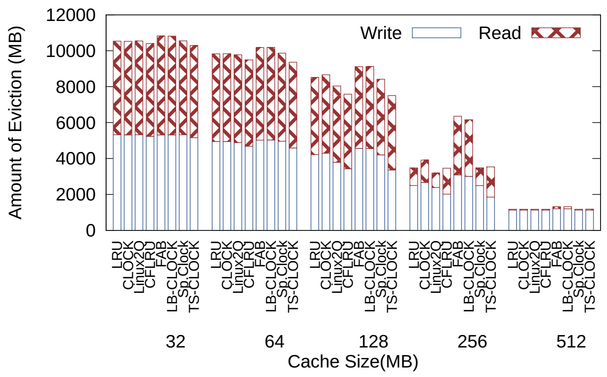


Fig. 7: The amount of generated I/O requests for the file server workload

Our experimental results reveal that write pattern strongly affects performance on low-end NAND flash devices which mostly use hybrid FTLs due to smaller memory requirement. In terms of performance, TS-CLOCK, FAB, and LB-CLOCK perform well since they all consider write patterns. However, in terms of lifetime, the coarse-grained cache management of FAB and LB-CLOCK generate significantly more number of write operations than TS-CLOCK does. As a result, TS-CLOCK improves performance and lifetime at the same time

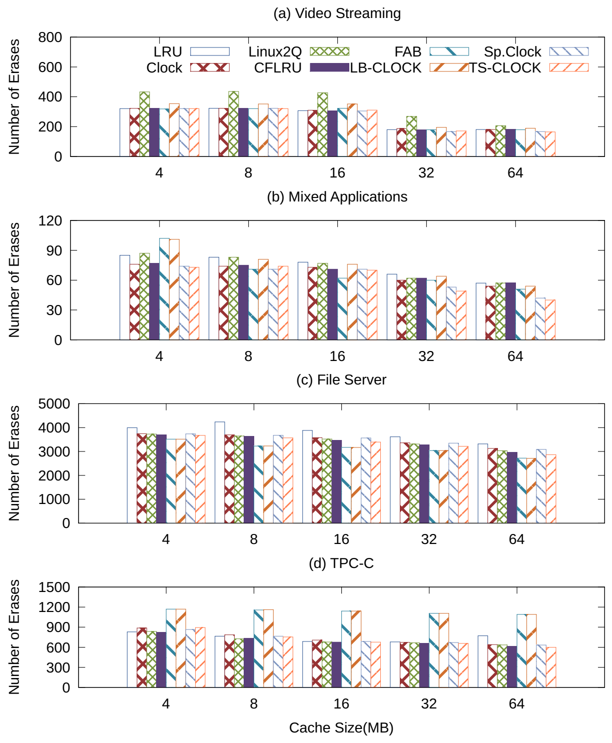


Fig. 8: Comparison of erase count on FAST FTL

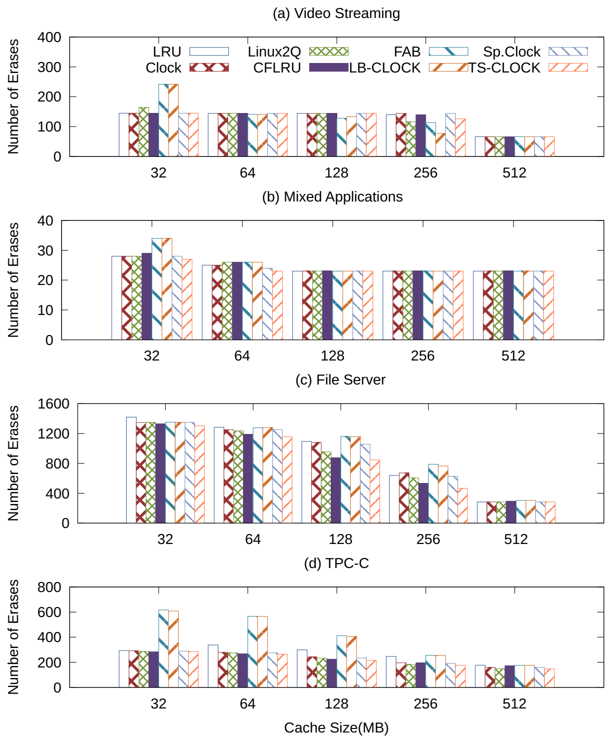


Fig. 9: Comparison of erase count on page-level FTL

by maintaining high hit ratios and generating flash-friendly write patterns.

D. Experiments on the SSD

Now, we investigate our experimental results on our TLC-based SSDs. Performance on SSDs could be different from that on microSD cards, since SSDs typically adopt page-level FTL to achieve high performance and their erase block size is much larger than that of microSD cards to exploit higher degrees of internal parallelism (24 MB in our case). Also, we set cache sizes to more realistic ranges for computing devices with high performance SSDs: from 32 MB to 512 MB.

Mobile Workloads: For the video streaming workload, which consists mostly of sequential accesses with some meta-data updates, TS-CLOCK performs well in all cache sizes (Figure 6a). However, under large enough cache size, e.g., 256 MB, to keep all metadata in memory, FAB and LB-CLOCK also performs well with high cache hit ratios. In particular, TS-CLOCK outperforms Sp.Clock, which is the best performing replacement algorithm for mobile workloads in the recent literature, by up to 11.8%. This result confirms that our approach to generate *flash-friendly write patterns* is more effective than the sorted eviction of Sp.Clock.

For the mixed applications workload of which significant portions are random accesses, results on the SSDs in Figure 6b show quite different trends than on the microSD devices in Figure 4b; all replacement algorithms show similar cache hit ratios and similar performance. However, these results are not surprising. As we discussed in Figure 1b, since I/O ranges of random writes are small compared to the capacity of a device, the WAF approaches to a value of one. Since the write range of the mixed applications workloads is quite small (133 MB over 120 GB, see Table II and I), performance will be mostly limited by the amount of write operations. Since all replacement algorithms show similar cache hit ratios, they all show similar performance.

Server Workloads: For the server workloads, TS-CLOCK performs best in all cases: on average 9.3% for the file server workload and 14.2% for the TPC-C workload (Figure 6). Interestingly, comparing to performance results on microSD cards, CFLRU performs much better while FAB and LB-CLOCK performs much worse. We found the reason by analyzing the after-cache traces. Figure 7 compares the amount of generated I/O requests from each of the replacement algorithms. It clearly reveals the limitations of FAB and LB-CLOCK. The amount of generated I/O requests from FAB and LB-CLOCK are the largest in all cases due to the coarse-grained buffer management. On the other hands, TS-CLOCK and CFLRU show lower eviction counts, especially in write operations, than the others due to their fine-grained buffer management and an eviction policy that prefers to evict clean pages over dirty pages. In addition, since TS-CLOCK generates pseudo sequential write patterns which minimize GC costs, it performs best in all cases.

In summary, the number of write operations strongly affects the performance of SSDs, which typically adopt page-level FTL with large overprovisioning space. It clearly shows that a buffer replacement algorithm should consider both of the number of generated write operations and their patterns to work well on a wide range of NAND flash storage devices.

E. Effect on Lifetime

In order to investigate how each replacement algorithm affect the lifetime of NAND flash storage devices, we ran the after-traces on our FTL simulator. The FTL simulator is trace-driven and supports FAST FTL and page-level FTL. For FAST FTL, it is configured as typical microSD cards:16 GB capacity, 15% over-provisioned space, 4 KB page, and 4 MB block. For page-level FTL, its block size is configured to 24 MB and the other parameters are the same as FAST FTL. Before running each workload, we run aging traces, which are 16 GB and are mixed with sequential accesses and random accesses.

Figure 8 and Figure 9 compare erase counts. Our experimental results clearly show that TS-CLOCK significantly extends the lifetime of NAND flash storage devices more than the others. More specifically, it can extends the lifetime by up to 29% compared to CFLRU on FAST FTL (Figure 8b) and by up to 53% compared to FAB on page-level FTL (Figure 9d). The reason behind these results is that TS-CLOCK prefers to evict clean pages to minimize write operations as well as it shapes evicted dirty pages in flash-friendly write patterns to minimize hidden write operations.

F. Write Patterns

To intuitively understand how each replacement algorithm behaves, we plot the before-cache traces of the mixed applications workload and the after-cache traces of the five selected algorithms, LRU, CFLRU, FAB, Sp.Clock, and TS-CLOCK. As we expected, LRU and CFLRU generates random I/O patterns since they do not consider write patterns (Figure 10b and Figure 10c). On the other hand, TS-CLOCK seems to evict multiple pages in a unit of a block like FAB although it actually evicts at the granularity of a page (Figure 10f). This is because that TS-CLOCK shapes evicted dirty pages to flash-friendly write patterns which have high block utilization. Especially, we also found that TS-CLOCK performs more sequential write operations compared to FAB since TS-CLOCK maximizes block utilization by clustering page based on reference count. Moreover, as shown in Figure 4 and Figure 6, TS-CLOCK outperforms Sp.Clock in all combinations of workloads and buffer cache sizes, even though Sp.Clock writes almost all pages sequentially (Figure 10e). This reveals that flash-friendly pseudo sequential writes are more effective than the sorted writes of Sp.Clock.

V. Related work

Over the years, the HDD has been considered to be a performance bottleneck because it has poor access latency due to the characteristic of having mechanical movement. To relieve this problem, various replacement and I/O scheduling algorithms have been proposed. Most replacement algorithms [21], [22], [39]–[41] have been designed to reduce the number of I/O operations by exploiting temporal locality. However, this involves

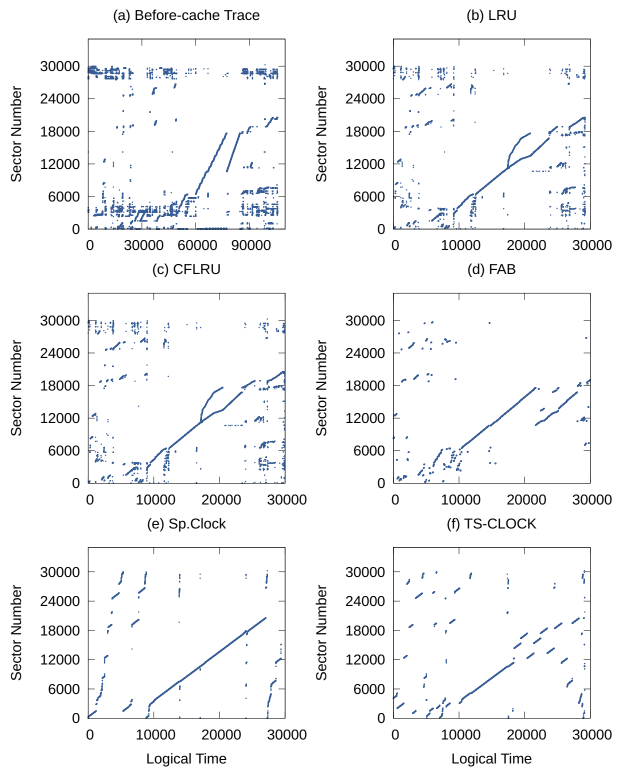


Fig. 10: Comparison among the before-cache trace and the after-cache traces. Only write operations are presented.

seeking and rotating overheads because they do not consider write patterns of evicted pages. In order to mitigate these overheads, I/O scheduling algorithms, such as SSTF, SCAN, VSCAN, and FSCAN, have focused on spatial locality, which can minimize seek and rotate times. DULO [23], WOW [24], and STOW [42] combine the above two approaches to improve I/O performance. However, these algorithms only consider characteristics of the HDD.

Recently, many approaches have been extensively proposed to consider the characteristics of NAND flash storage devices in the buffer replacement layer. Previous flash-aware buffer replacement algorithms can be classified into two: First, CFLRU [12], LRU-WSR [14], and FOR [15] consider the asymmetric read and write cost and they prefer to evict clean pages over dirty pages to reduce the number of more expensive write operations. However, these algorithms cannot guarantee I/O performance, because they generate random write patterns. Second, the other algorithms [13], [16]–[18], [25] focus on write patterns. FAB, BPLRU, and LB-CLOCK select a block including the largest number of pages and evicts all pages in the block for generating sequential write patterns. Sp.Clock manages pages by its logical sector number and uses the sorted eviction scheme. The major drawback of these algorithms is that they slightly ignore the recency by evicting whole pages in the same block and sorting pages in the block by the sector number. BPAC exploits the temporal and spatial locality by using the dual-list. Unfortunately, BPAC does not consider read operations because it has been designed for the write buffer in SSD.

VI. Conclusion

In this paper, we proposed a novel buffer replacement algorithm, called TS-CLOCK, for NAND flash storage devices. TS-CLOCK extends the CLOCK algorithm for temporal locality and exploits spatial locality for generating the *flash-friendly write pattern*. Moreover, it prefers to evict clean pages over dirty pages to minimize expensive write operations. Our experimental results clearly confirm that TS-CLOCK outperforms existing replacement algorithms in terms of performance and lifetime of the devices. In future work, we explore how does TS-CLOCK affect the host system (e.g., CPU or memory) because we believe that the key idea of TS-CLOCK can be widely adopted for other environments to optimize performance and lifetime of storage devices.

Acknowledgments

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT&Future Planning (2010-0020730) and the IT R&D program of MKE/KEIT. [10041244, SmartTV 2.0 Software Platform]

References

[1] V. Kristian. (2012, Sep.) Samsung releases tlc nand based 840 ssd. [Online]. Available: http://www.anandtech.com/show/6329/samsung-releases-tlc-nand-based-840-ssd
[2] A. L. Shimpi. (2013, Jan.) Plextor's dabbles in tlc nand. [Online]. Available: http://www.anandtech.com/show/6577/plextors-dabbles-in-tlc-nand
[3] K.-C. Ho, P.-C. Fang, H.-P. Li, C.-Y. Wang, and H.-C. Chang, "A 45nm 6b/cell charge-trapping flash memory using LDPC-based ECC and drift-immune soft-sensing engine," in *Proc. IEEE ISSCC*, 2013.
[4] L. M. Grupp, J. D. Davis, and S. Swanson, "The Bleak Future of NAND Flash Memory," in *Proc. USENIX FAST*, 2012.
[5] V. Kristian. (2012, Nov.) Samsung ssd 840: Testing the endurance of tlc nand. [Online]. Available: http://www.anandtech.com/show/6459/samsung-ssd-840-testing-the-endurance-of-tlc-nand
[6] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. ACM SIGMETRICS*, 2009.
[7] X.-Y. Hu and R. Haas, "The fundamental limit of flash random write performance: Understanding, analysis and performance modelling," Research Report RZ 3771, IBM Research, 2010.
[8] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random Write Considered Harmful in Solid State Drives," in *Proc. USENIX FAST*, 2012.
[9] D. Woodhouse, "Jffs: The journalling flash file system," in *Proc. IEEE LINUXSYM*, 2001.
[10] YAFFS. Yet another flash file system. [Online]. Available: http://www.yaffs.net/
[11] UBIFS. Unsorted block image file system. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubifs.html
[12] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "CFLRU: a Replacement Algorithm for Flash Memory," in *Proc. ACM CASES*, 2006.
[13] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: flash-aware buffer management policy for portable media players," *IEEE Trans. Consum. Electron.*, vol. 52, no. 2, pp. 485–493, May 2006.
[14] H. Jung, H. Sim, P. Sungmin, S. Kang, and J. Cha, "LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory," *IEEE Trans. Consum. Electron.*, vol. 54, no. 3, pp. 1215–1223, Aug. 2008.
[15] Y. Lv, B. Cui, B. He, and X. Chen, "Operation-aware buffer management in flash-based systems," in *Proc. ACM SIGMOD*, 2011.
[16] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?" in *Proc. ACM SIGMETRICS*, 2012.
[17] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proc. USENIX FAST*, 2008.
[18] B. Debnath, S. Subramanya, D. Du, and D. J. Lilja, "Large Block CLOCK (LB-CLOCK): A write caching algorithm for solid state disks," in *Proc. IEEE MASCOTS*, 2009.
[19] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," in *Proc. USENIX FAST*, 2012.
[20] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-based SSDs," in *Proc. USENIX ATC*, 2013.
[21] A. Bensoussan, C. Clingen, and R. C. Daley, "The Multics virtual memory: concepts and design," *ACM Commun.*, vol. 15, no. 5, pp. 308–318, May 1972.
[22] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, 2005.
[23] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality," in *Proc. USENIX FAST*, 2005.
[24] B. S. Gill and D. S. Modha, "WOW: Wise Ordering for Writes – Combining Spatial and Temporal Locality in Non-Volatile Caches," in *Proc. USENIX FAST*, 2005.
[25] G. Wu, X. He, and B. Eckart, "An Adaptive Write Buffer Management Scheme for Flash-Based SSDs," *ACM Trans. on Storage*, vol. 8, no. 1, p. 1, Feb. 2012.
[26] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE Trans. Consum. Electron.*, vol. 48, pp. 366–375, May 2002.
[27] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-memory Based File System," in *Proc. USENIX TCON*, 1995.
[28] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. ACM ASPLOS*, 2009.
[29] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, p. 18, Jul. 2007.
[30] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 36–42, Oct. 2008.
[31] S. Boboila and P. Desnoyers, "Performance Models of Flash-based Solid-state Drives for Real Workloads," in *Proc. IEEE MSST*, 2011.
[32] P. Desnoyers, "Analytic Modeling of SSD Write Performance," in *Proc. ACM SYSTOR*, 2012.
[33] L. Bouganim, B. Jónsson, and P. Bonnet, "uflip: Understanding flash io patterns," in *Proc. CIDR CIDR*, 2009.
[34] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-Aware I/O Management for Solid State Disks (SSDs)," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 636–649, May 2012.
[35] H. H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance Modeling and Analysis of Flash-based Storage Devices," in *Proc. IEEE MSST*, 2011.
[36] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for Flash-based SSDs with Nameless Writes," in *Proc. USENIX FAST*, 2012.
[37] The DBENCH web pages. [Online]. Available: http://dbench.samba.org/
[38] Transaction Processing Performance Council. TPC Benchmark C. [Online]. Available: http://www.tpc.org/tpcc/spec/tpcc_current.pdf
[39] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proc. USENIX FAST*, 2003.
[40] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proc. ACM SIGMETRICS*, 2002.
[41] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," in *Proc. USENIX ATC*, 2005.
[42] B. S. Gill, M. Ko, B. Debnath, and W. Belluomini, "STOW: A Spatially and Temporally Optimized Write Caching Algorithm," in *Proc. USENIX ATC*, 2009.