GASA: A New Page Replacement Algorithm for NAND Flash Memory

Chu Li, Dan Feng, Yu Hua, Wen Xia, Fang Wang*
Wuhan National Lab for Optoelectronics
School of Computer, Huazhong University of Science and Technology
Wuhan, Hubei, P. R. China, 430074
{lichu, dfeng, csyhua, xia, wangfang}@hust.edu.cn

***Abstract*—NAND flash memory has been widely used as storage medium in diverse environments due to its light weight, low power consumption, and high I/O performance. Page replacement is an important operation in NAND flash-based storage systems. However, traditional replacement algorithms designed for magnetic disks fail to meet the needs of NAND flash memory due to inherent features such as asymmetric I/O latencies and erase-before-write. In order to address this problem, this paper proposes a new page replacement algorithm, called Ghost buffer Assisted and Self-tuning Algorithm (GASA). GASA reduces expensive flash write operations by evicting cold clean pages preferentially and maintains reasonable buffer hit ratios via a ghost buffer. In addition, GASA is self-tuning due to the use of a simple learning scheme, thus adaptively matching different workloads and buffer sizes. Experimental results based on real-world OLTP traces demonstrate that GASA offers a good trade-off between the hit ratio and the flash write count, thus achieving better I/O performance than the state-of-the-art page replacement algorithms.**

Keywords—NAND flash memory; Buffer cache; Page replacement algorithm; Performance.

I. INTRODUCTION

Nowadays, NAND flash memory is widely used as a storage medium in mobile devices like PDAs and laptops due to its superior features such as light weight, low power consumption, and high I/O performance [1]-[3]. With the increasing density and decreasing price, NAND flash memory based solid state drives (SSDs) are also used in desktop and server applications [4], [5]. The performance of the storage system is critical in different types of environments. For example, data-intensive server applications such as OLTP (Online Transaction Processing) normally demand high-performance underlying storage systems [6]. When it comes to mobile systems, some recent works suggest that storage performance has a growing impact on end-user experience [3], [7]. Therefore, optimizing the performance of flash storage systems has been considered extremely important.

NAND flash memory has much lower random access latency than magnetic disks due to no mechanical parts involved. Hence, they help alleviate I/O bottlenecks in some critical data-intensive applications [8]. However, several hardware limitations exist in NAND flash memories. There are three types of operations in flash memory, i.e., read, write, and erase. The latency of write operation is significantly higher than that of the read operation, and the erase operation is much slower than the write operation. This is very different from magnetic disks which have similar performance for read/write operations. Moreover, flash memory fails to support in-place update. Blocks have to be erased before they are rewritten. Flash memory also suffers from endurance problems because blocks can only be erased a limited number of times, typically between 10,000 and 100,000 cycles [9].

Buffer caching plays an important role in improving the storage I/O performance. For decades, many page replacement algorithms have been proposed [10]-[13]. However, traditional algorithms are designed for magnetic disks without considering the properties of flash memory. Specifically, those algorithms aim to improve buffer hit ratio while overlooking the asymmetric I/O performances for read and write operations in flash memory. As a result, they fail to perform well for NAND flash memory based storage systems.

Various algorithms have been proposed to improve the efficiency of the buffer cache for flash memory based storage systems. CFLRU [14] and LRU-WSR [15] algorithms enhance the LRU (Least Recently Used) algorithm by reducing flash write operations while avoiding serious decrease of hit ratios. However, they fail to fully consider the access frequency of pages in the buffer cache. Hence, flash writes cannot be reduced significantly due to cache pollution by cold clean pages, which limits the improvement of overall I/O performance. The CCF-LRU [16] and the recently proposed PT-LRU [17] algorithms classify pages into four states and evict cold clean pages first. They can significantly reduce the write count and are more effective for traces with higher write locality [15], [16]. However, they are not intelligent enough to differentiate cold and hot clean pages. As a result, they suffer from severe degradation of buffer hit ratios for workloads with strong read locality, which even decreases the overall I/O performance.

In this paper, a new page replacement algorithm, called GASA (Ghost buffer Assisted and Self-tuning Algorithm), is proposed for NAND flash memory. GASA strives to improve the overall I/O performance of the flash memory based storage systems by striking a suitable trade-off between the hit ratio and the flash write count. More specifically, GASA separates cold clean pages with other pages using two LRU lists and evicts the cold clean pages preferentially in order to reduce

Corresponding Author: Fang Wang (wangfang@hust.edu.cn).

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

expensive flash writes. Unlike CCF-LRU [16], the unique feature of GASA is using a ghost buffer that effectively identifies potential hot pages. As a result, GASA avoids severe degradation of buffer hit ratios that CCF-LRU suffers from. We use a simple yet effective policy to tune the size of the ghost buffer dynamically eliminating the performance issues of the fixed-sized ghost buffer [11], [18]. The ghost buffer is continuously adjusted according to several buffering events, striving to achieve high I/O performance under various buffer sizes and workloads.

To show the effectiveness of the proposed GASA, we conducted extensive simulation experiments using several real-world traces with different read/write ratios. Experimental results show that GASA achieves better I/O performance than the state-of-the-art algorithms under various public OLTP traces. Specifically, GASA reduces the flash writes by up to 38.5% while maintaining reasonable (even higher) hit ratios compared with LRU. It has fewer flash writes than all the competitors except for CCF-LRU and PT-LRU. In terms of overall runtime for I/O operations, GASA improves the performance of LRU by up to 16.1% and outperforms all the existing algorithms designed for flash memory.

The rest of the paper is organized as follows. Section II presents the overview of the flash memory and the existing page replacement algorithms designed for NAND flash memory based storage. Section III describes the design of the proposed algorithm GASA. In Section IV, the experimental evaluation of GASA compared with previous algorithms is provided and discussed. Finally, Section V summarizes and concludes this paper.

II. Background and Related Work

A. Flash Memory

NAND flash memory can be subdivided into two categories, Single-Level Cell (SLC) and Multi-Level Cell (MLC), according to the number of bits stored in one cell [1], [9]. Three operations are provided in flash memory, including read, write, and erase. The read and write operations are performed in a page unit, which is typically 4,096-16,384 bytes in size. The erase operation is performed in a block unit, which typically consists of 64-256 pages. For a typical SLC NAND flash memory, the write time is about 8 times longer than the read time and the erase time is about an order of magnitude larger than the write time. The performance asymmetry is even heavier for MLC flash. In addition, NAND flash memory does not support in-place update. A page must be erased before being rewritten, which is called erase-before-write. Erasing is relatively a time-consuming operation and the number of erase operations is proportional to the number of write operations, which leads to the more expensive costs of flash write operations compared with read operations. Furthermore, NAND flash also suffers from endurance problems due to only offering a limited number of program/erase cycles, beyond which they will become unreliable.

Typically, a flash translation layer (FTL) is used to emulate the flash memory as a normal block device providing read/write operations [19], as shown in Figure 1. A buffer cache located between the host system and the FTL absorbs host I/O requests. Various memory types such as DRAM and non-volatile RAM (NVRAM) can be used for buffer caching because they have much lower access latency than flash. However, the buffer cache has quite a limited capacity for cost efficiency. As a result, the inherent properties of flash memory are considered in GASA to improve the effectiveness of the buffer cache.

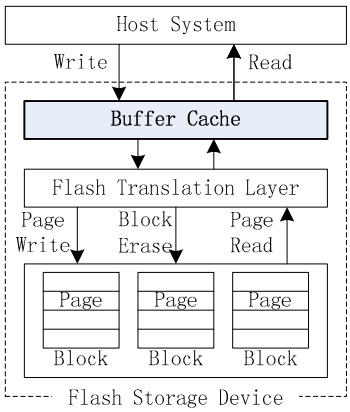


Fig. 1 The architecture of NAND flash storage system

B. Page Replacement Algorithms for Flash Memory

For decades, page replacement algorithms have been widely used to improve the overall I/O performance of storage systems. Traditional replacement algorithms [10]-[13] merely focus on reducing the miss count without distinguishing the different types of I/O operations. However, while the write operation is much slower than the read operation in flash memory, the I/O performance is related to not only the hit ratio but also the distribution of read, write operations sent to the flash memory. To address this problem, many algorithms [14]-[17] are proposed for flash memory based storage systems over the past few years.

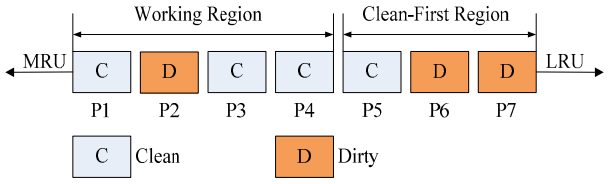


Fig. 2. An example of CF-LRU.

CFLRU (Clean First LRU) is the first algorithm designed for flash memory [14]. The basic idea of CFLRU is to reduce the average replacement cost by reducing the number of flash writes while alleviating severe decrease of hit ratios. To achieve this goal, CFLRU enhances the LRU algorithm by splitting the LRU list into the working region and the clean-first region. When finding a victim page, CFLRU selects a clean page in the clean-first region in LRU order. If there are no clean pages in this region, a dirty page at the LRU position is evicted. As shown in Figure 2, P5, instead of P7, will be chosen as the victim.

CFLRU is able to reduce the write operations by preferentially evicting clean pages, and maintain the hit ratio

through choosing an appropriate window size for the clean-first region. However, it has some drawbacks. First, the access frequencies of the buffered pages are not carefully considered in CFLRU, which can result in poor I/O performance due to cache pollution by cold pages. Second, it is nontrivial to find an appropriate window size under various kinds of workloads and different buffer sizes.

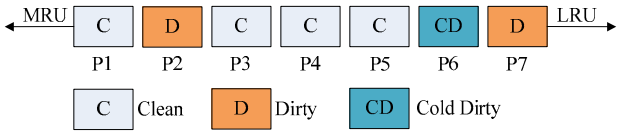


Fig. 3. An example of LRU-WSR.

Jung et al. proposed an enhanced LRU algorithm, called LRU-WSR, which seamlessly integrated LRU with the introduced WSR (Write Sequence Reordering) strategy [15]. In LRU-WSR, a cold-flag bit is attached with each page. The first referenced pages are inserted to the cache without setting the cold-flag bit. During the eviction procedure, the clean pages and the cold dirty pages are preferentially chosen as victim pages while the hot dirty pages are moved to the MRU (Most Recent Used) position with the cold-flag bit being set. When a dirty page in the cache is referenced again, it is moved to the MRU position with its cold-flag cleared. For example, as shown in Figure 3, in an eviction, P7 will be moved to the MRU position with setting its cold-flag, and P6 will be selected as the victim because its cold-flag is set.

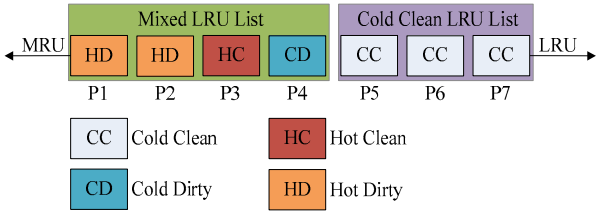


Fig. 4. An example of CCF-LRU.

LRU-WSR considers the access frequency of dirty pages and tries to first evict cold dirty pages and clean pages. Consequently, it reduces the flash write counts and the hit ratios of LRU-WSR approximate the traditional LRU algorithm. However, the overall performance may be adversely affected, because the access frequency of clean pages is not considered. Dirty pages may be evicted while many cold clean pages are residing in the cache, leading to more costs for flushing buffered pages. On the other hand, a hot clean page may be evicted even if there are many cold dirty pages in the buffer, which results in extra read operations.

CCF-LRU (Cold-Clean-First LRU) [16] enhances the CFLRU and LRU-WSR algorithms by further differentiating clean pages into cold and hot ones. It uses the cold-detection mechanism of LRU-WSR to classify the buffered pages into four classes, including cold clean pages, hot clean pages, cold dirty pages, and hot dirty pages. As illustrated in Fig. 4, the cold clean pages are stored in a cold clean LRU list while other types of pages are held in the mixed LRU list. When a replacement occurs, CCF-LRU always evicts cold clean pages first. If there are no cold clean pages, cold dirty pages are preferentially chosen as the victim and the eviction of hot dirty pages are delayed as long as possible.

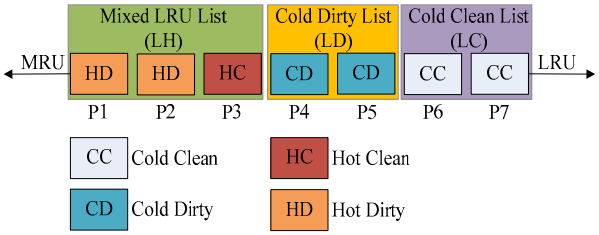


Fig. 5. An example of PT-LRU.

Recently, Cui et al. [17] proposed a new page replacement algorithm, called PT-LRU (Probabilistic Triplicate LRU). As shown in Figure 5, PT-LRU manages the buffer using three LRU lists, including the cold clean list (LC), the cold dirty list (LD), and the mixed LRU list (LH). The first referenced page is inserted to LC, and will be moved to LH when it is referenced again or become dirty. When there is no free page slot for a new access page, the policy evicts the LRU page of LC preferentially. If the list is empty, the cold dirty pages are evicted at a higher probability. Whenever choosing a victim from LH, hot clean pages are evicted first and a modest number of hot dirty pages are moved to LD.

CCF-LRU and PT-LRU can significantly reduce large amounts of flash write operations by evicting cold clean pages first. However, hot clean pages cannot be identified and admitted into the mixed LRU lists effectively due to frequent evictions of cold clean pages. By design, these algorithms are more effective for traces with higher write locality [15]. However, many real-world traces also show strong read locality, in which case they suffer from severe decrease of buffer hit ratios.

III. GASA

GASA strives to improve the I/O performance of the flash-based storage by reducing the write operations to flash while preventing dramatic decrease of hit ratios. To achieve this goal, GASA evicts cold clean pages first to reduce flash writes efficiently. A Ghost buffer that has been employed in lots of page replacement algorithms with various goals [8], [10], [17] is used in GASA to deal with the problems faced by CCF-LRU and PT-LRU. In addition, GASA adjusts the size of the ghost buffer dynamically under different workloads and buffer sizes. The details of the proposed algorithm GASA are described in the following subsections.

A. The Basic Policy

Pages in GASA are classified into four states, including cold clean, cold dirty, hot clean, and hot dirty. Like CCF-LRU, GASA evicts cold clean pages preferentially to reduce the number of write operations to flash memory. However, the key problem of CCF-LRU is that it may suffer from severe

decrease in buffer hit ratios due to the eager evictions of newly referenced pages. To address this problem, GASA chooses victims considering the page states and introduces a ghost buffer which stores the metadata of the evicted pages. Whenever a referenced page hits in the ghost buffer, it will be fetched into the buffer cache and regarded as hot. In this way, potential hot pages can be efficiently identified and have more chances to enter the buffer cache in GASA, which leads to high buffer hit ratios. Note that the memory overhead of the ghost buffer is minimal due to only recording the logical addresses of the evicted pages, not the actual data. The work flow of GASA is described as follows.

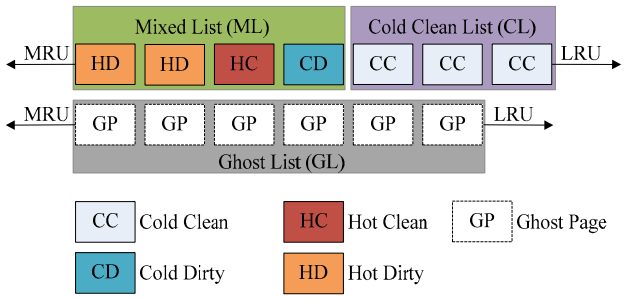


Fig. 6. The three LRU Lists used in GASA.

As shown in Figure 6, GASA maintains three LRU lists, i.e., the cold clean list (CL), the mixed list (ML), and the ghost list (GL). The CL list maintains the cold clean pages while the ML list consists of the hot clean pages, the cold dirty pages, and the hot dirty pages. The ghost buffer is managed by the GL list. When reading a page P, GASA queries the P in all three lists. If found, it will be marked to be hot and moved to the MRU (Most Recently Used) position of the ML list. Otherwise, the page will be inserted into the MRU position of the CL list. If the referenced page is found in the GL list, it still needs to fetch the actual data from the flash memory. The process of writing a page is similar. The only difference is, the buffer page will be marked as dirty and moved to the MRU position of the ML list. Whenever allocating for a new page in the buffer cache, if the buffer has no free page slots, it will trigger the eviction procedure which will be presented next.

When a real page needs to be evicted out of the buffer cache for a new page insertion, the CL list will be checked first. If it is not empty, the LRU page will be selected as the victim and its metadata will be recorded in the MRU position of the GL list. Otherwise, GASA scans the ML list beginning from the LRU end to choose the first cold dirty page as the victim page. The metadata of that page will also be inserted into the MRU position of the GL list after its eviction. During the scanning procedure, hot clean pages are changed into cold clean pages and moved to the MRU position of the CL list. On the other hand, hot dirty pages are changed into cold dirty pages and moved to the MRU position of the ML list. In this way, hot dirty pages tend to be kept in the buffer cache longer to reduce the slow write operations to flash memory. If it fails to get a victim page after these steps, the above procedure will be repeated again. Note that a victim page can be successfully returned during the second scanning, because in that case either the CL is not empty or the ML list has cold dirty pages.

```
Algorithm      GASA_Access (Page p,  Operation op)
Data queues:   ML = Mixed LRU list;
               CL = Cold Clean LRU list;
               GL = Ghost LRU list;
Out: The reference to the requested data page p in buffer

  Initialize: glflag=0; GS=GSMIN;
  if p is in ML or p is in CL    //buffer hit
    set hot-flag of p;
    move p to MRU position of ML;
    if ghost-flag of p is set    //adjust ghost buffer size
      GS = min(GS+1, GSMAX);
      clear ghost-flag of p;
  else
    if p is in GL
      remove p from GL;
      glflag=1;   //indicating it is a ghost buffer hit
    p = GASA_Alloc();
    if glflag!=1 and op is read
      move p to MRU position of CL;
    else
      if glflag==1
        set the ghost-flag and hot-flag of p;
      move p to MRU position of ML;
  return the reference to p
```

Fig. 7. The page management policy of GASA.

```
Algorithm   GASA_Alloc()
Out: The reference to the allocated data page p in buffer

  Initialize: victim = NULL;
  if there is no free space in the buffer
    while victim == NULL
      if CL is not empty
        victim = the LRU page in CL;
      else
        for each page p from LRU position of ML
          if the hot-flag of p is not set
            victim = p;
            break;
          else
            clear the hot-flag of p;
            if p is clean
              move p to MRU position of CL;
            else if p is dirty
              move p to MRU position of ML;
  else
    victim = get a free page from the buffer
  if the ghost-flag of victim is set
    GS = max(GSMIN, GS – GSMAX/(GSMAX-GS+1));
  allocate a ghost page for victim and insert it to MRU
  position of GL;
  adjust the size of GL to GS;
  if victim is dirty
    flush victim to flash memory;
  return victim;
```

Fig. 8. The page allocation algorithm of GASA.

B. The Learning Scheme

The ghost buffer is used to identify potential hot pages, which is important due to the following two reasons. First, although the ghost buffer only contains metadata with low memory overhead, too many ghost pages can still result in smaller data cache which may decrease the performance of the buffer cache. Second, the GL list also has an impact on the trade-off between the hit ratio and the write count, which is explained as follows. Specifically, if the GL list is too short, it may fail to avoid frequent evictions of newly referenced clean pages. On the other hand, if the GL list is too large, previously evicted cold clean pages will have more chances to enter the MRU position of the ML list, which may evict more dirty pages and increase flash writes.

Based on the above analysis, GASA uses a simple learning scheme to manage the ghost buffer automatically. A bit flag called *ghost-flag* is introduced to allow GASA to decide to either shrink or enlarge the ghost buffer, which is presented as follows. Initially, the *ghost-flag* bit of each page in the buffer cache is not set. When a referenced page is found in the GL list, it is moved to the MRU position of the ML list and the page's *ghost-flag* is set. If the page is referenced again before it is evicted from the ML list, it indicates that the ghost buffer successfully identifies the hot pages. Consequently, the target size of the buffer cache will be increased in order to achieve higher hit ratios. When a page is chosen as the victim, its *ghost-flag* is also checked. If the *ghost-flag* is still set, it means the page has not been accessed ever since it was identified to be hot. Therefore, the ghost buffer should be reduced in this case. In this way, the ghost buffer is continually adjusted to different workloads and buffer sizes.

C. The Proposed GASA

The pseudo-codes of GASA are shown in Figure 7 and 8. A *hot-flag* is attached with each page for distinguishing hot pages from cold pages in the buffer cache. When accessing a page, its *hot-flag* will be set to 1 whenever it is found in any of the three LRU lists (i.e., ML, CL and GL). The parameters GS, GSMIN, and GSMAX are used for the learning scheme. GS is a variable which indicates the target size of the GL list. GSMIN and GSMAX represent the lower and upper bound of GS, which are empirically set to 10% and 100% of the actual buffer size in terms of the total number of buffer pages in ML and CL, respectively, in this paper. When the ghost buffer should be enlarged, GS is increased by 1. On the contrary, when the ghost buffer should be decreased, GS is reduced by GSMAX/(GSMAX-GS+1), which is larger than 1. The larger the current size of the ghost buffer is, the quicker the shrinking of GS will be. The reason is that GASA keeps the GL list shorter to avoid increasing more flash writes, as mentioned in the previous subsection.

IV. Performance Evaluation

In this section, we conduct trace-driven experiments to evaluate the performance of the proposed algorithm GASA compared with the traditional LRU policy as well as the recently proposed algorithms for NAND flash memory, including CFLRU, LRU-WSR, CCF-LRU, and PT-LRU. The evaluation metrics, including buffer hit ratio, flash write count, and overall runtime, are used for the performance comparisons. In order to facilitate better understanding of the algorithms, we also present some detailed metrics such as the split buffer hit rates on different kinds of buffered pages.

A. Experimental Setup

We built a trace-driven buffer cache simulator and integrated it with FlashSim [19], which has the interfaces with DiskSim [20]. We implemented six page replacement algorithms, including LRU, CFLRU, LRU-WSR, CCF-LRU, PT-LRU, and our proposed GASA in the buffer simulator. FlashSim is a well-known and validated flash system simulator, and it allows simulating various flash chips. In this paper, a typical kind of NAND flash memory is simulated for the performance evaluations of different page replacement algorithms. The detailed parameter settings which are well-recognized [21] are shown in Table I.

TABLE I. PARAMETERS OF THE NAND FLASH MEMORY

Parameter	Value
Page Size	4 KB
Block Size	64 pages
Page Read Latency	25 μs
Page Write Latency	200 μs
Block Erase Latency	1.5 ms

Several existing algorithms have adjustable parameters, which actually can affect the results of the performance evaluation. In the experiments, the parameters are chosen carefully according to previous studies for fair comparisons. For example, the *window size* of CFLRU is set to 0.1, which is the same as the value used in the LRU-WSR [15] and the CCF-LRU [16]. The parameter *pro* (i.e., the probability of evicting a cold dirty page) in PT-LRU is set to 0.8 which is the best set according to the original paper [17].

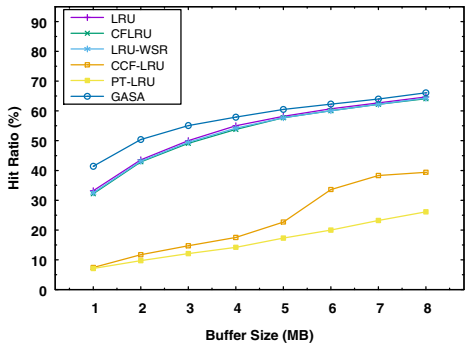
TABLE II. THE OLTP TRACES USED IN SIMULATIONS

Traces	Total Requests	Unique Pages	Read Ratio	Read Locality	Write Locality
T1	1,347,536	61,727	85%	13% / 87%	31% / 69%
T2	911,657	46,627	87%	17% / 83%	16% / 84%
T3	766,673	74,760	34%	28% / 72%	11% / 89%

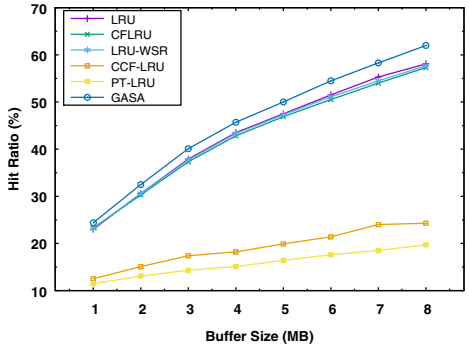
Various real-world traces are used in the experiments. While the traces used in recent work [16], [17] are not released, this paper employs public and widely-used traces from OLTP applications running at a financial institution made available by the Storage Performance Council (SPC) [22]. Several representative sub-traces denoted by T1-T3 are chosen according to the "Application Specific Unit (ASU)" field of the traces like previous studies [23], [24]. The characteristics of the traces are summarized in Table II. The locality expressions (p% / a%) means %p of total number of unique pages are accessed by a% of total number of accesses. As shown in Table II, among the various workloads, T1 and T2 are read-dominant whereas T3 is write-dominant. And the workloads have different characteristics in terms of read/write locality.

B. *Experimental Results*

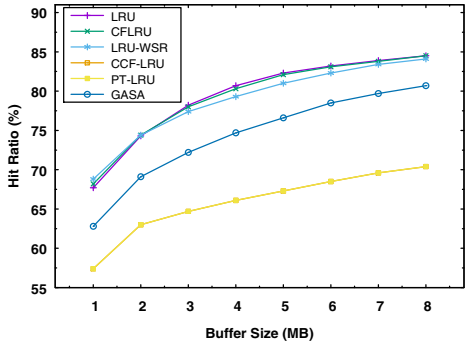
Experimental results of the six page replacement algorithms under several real-world workloads are shown in the following subsections. Since the public OLTP workloads have small working sets, the size of the buffer cache in our tests ranges from 1MB to 8MB in order to demonstrate performance differences among the different cache algorithms. The cache-line size of the buffer cache is set to one physical page of the flash memory (i.e., 4KB).



(a) T1



(b) T2

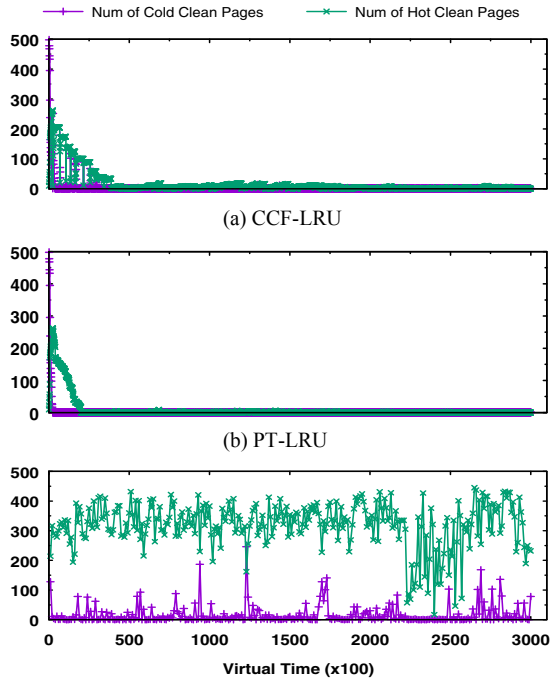


(c) T3

Fig. 9. Hit ratios for the traces under various buffer sizes.

1) *Buffer Hit Ratios*

Figure 9 shows the buffer hit ratios of various page replacement algorithms under the OLTP workloads. While CFLRU and LRU-WSR are modified from LRU and do not fully consider the page access frequencies, their hit ratios are approximate to those of LRU. GASA achieves the highest hit ratios under the read-dominant traces T1 and T2, which demonstrates that GASA can effectively identify the potential hot pages. Under trace T3, the hit ratios of GASA are lower than those of other three algorithms (i.e., LRU, CFLRU, and LRU-WSR) but still higher than those of CCF-LRU and PT-LRU. Note that the ghost buffer hit rates are not counted in for obtaining the buffer hit ratios in GASA because the ghost buffer only has metadata and real page replacements are still required in those cases. Both CCF-LRU and PT-LRU have the lowest hit ratios under the various traces, which are quite different from the results in previous studies [16], [17]. For better understanding of the results, we plot detailed metrics such as the number of hot/cold clean pages and detailed buffer hit ratios in Figure 10 and 11 under T1 with a 2MB buffer. Simulations under other traces and buffer sizes are also performed, and similar trends are obtained.



(a) CCF-LRU

(b) PT-LRU

(c) GASA

Fig. 10. Number of hot/cold clean pages in buffer for T1 under 2MB buffer.

We believe that the poor performance of CCF-LRU and PT-LRU in terms of buffer hit ratios is mainly due to the lack of intelligence of identifying hot clean pages. As shown in Figure 10, the x-axis is the virtual time which ticks at each page reference and the y-axis is the number of buffered hot/cold clean pages. It clearly shows that both CCF-LRU and PT-LRU evict almost all clean pages soon. The reason is that they always evict the cold clean pages first, and thus the cold clean lists tend to be empty over time, in which case newly referenced clean pages will be evicted immediately and can hardly become hot. As a result, few clean pages can be kept in the buffer cache as the system runs. On the contrary, GASA avoids this problem due to its efficient scheme to protect potential hot pages.

GASA has reasonable and better buffer hit ratios than other algorithms under the OLTP workloads and various buffer sizes. To show the reason, Figure 11 plots the detailed buffer hit ratios splitting read and write operations, which are further split according to the status of the buffered pages. For example, the

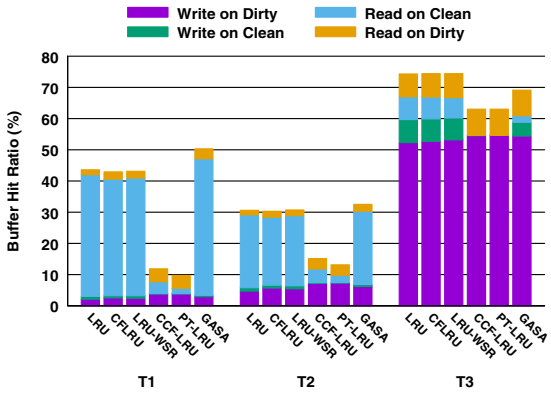
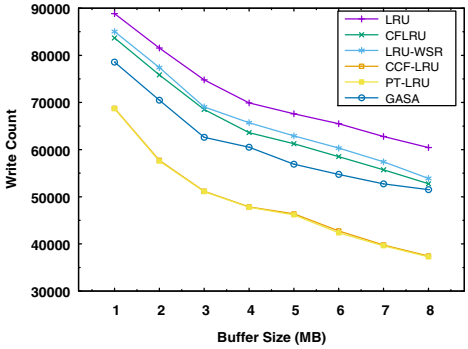


Fig. 11. Detailed hit ratios for the traces under 2MB buffer.

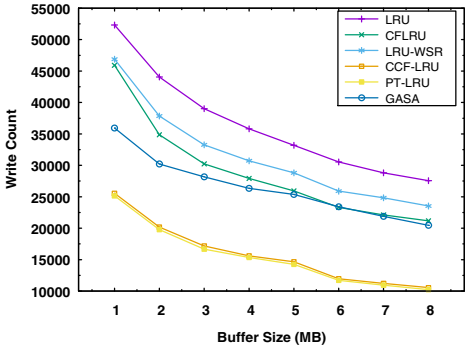
"Write on Dirty" indicates the ratio of write hits on dirty pages to the total number of requests. From Figure 11, across the three traces, one observation is, compared with LRU, all other algorithms have more write hits on dirty pages, which enable them to reduce the slow write operations to flash memory as will be described in the next subsection. This is because all the these algorithms proposed for flash memory try to keep dirty pages in the buffer longer. Another observation is, compared with LRU, CFLRU, and LRU-WSR, GASA has more hit rates on clean pages for T1/T2 and less hit rates on clean pages for T3. The reason is explained as follows. While T1/T2 are read-intensive with high read locality, GASA identifies more hot clean pages and admitted them into the ML list, thus achieving higher hit ratios. The T3 trace is write-intensive and shows higher write locality, in which case, GASA prefers to keep more dirty pages while evicting more clean pages, suffering from lower hit ratios of clean pages. The last observation is that the severe degradations of hit ratios for CCF-LRU and PT-LRU are attributable to few buffer hits on clean pages in buffer cache. This is mainly due to the inefficiency for detecting hot clean pages of these two algorithms and the strong read locality of the OLTP traces.

2) Flash Writes

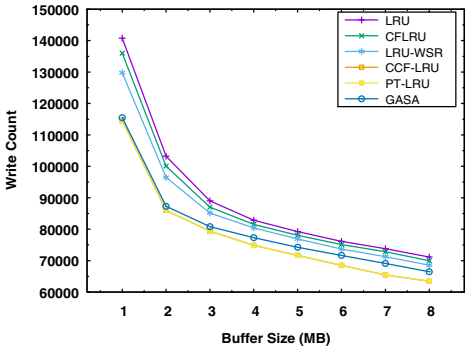
We obtain the number of flash write operations by counting page writes whenever a page replacement occurs, and adding the number of dirty pages in the buffer cache at the end of the simulation [15]. The flash write counts in different buffer policies under various buffer sizes are shown in Figure 12. The statistics are captured between the buffer and the FTL layer. All the algorithms proposed for flash memory have less flash writes than LRU. Across the three traces, the reductions of flash writes are more significant for the read-intensive traces T1 and T2 than the write-intensive trace T3. While CCF-LRU and PT-LRU evict clean pages eagerly as mentioned earlier, they have the least number of flash writes among all the traces under different buffer sizes. Excluding CCF-LRU and PT-LRU, GASA outperforms all the other competitors in terms of flash write count. Particularly, compared with LRU, GASA reduces the flash writes by up to 19.9%, 38.5%, and 18.0% under the T1, T2, and T3, respectively.



(a) T1



(b) T2



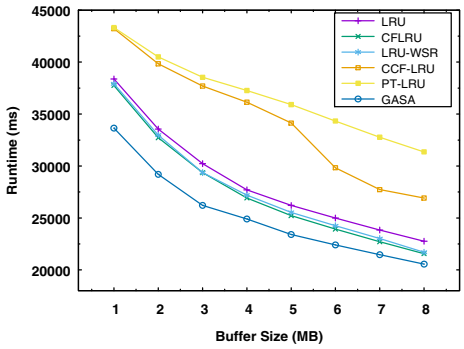
(c) T3

Fig. 12. Flash writes for the traces under various buffer sizes.

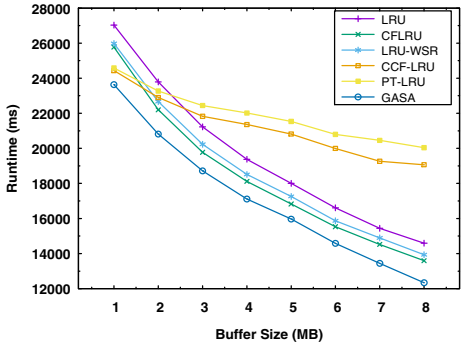
Note that the number of I/O operations sent to the flash memory is related to the buffer hit ratios. The higher the read hit ratios are, the less the read operations sent to the flash; the more the write hits on dirty pages, the less the write operations sent to the flash. The former is obvious while a brief explanation for the latter is shown below. Write hits on clean pages cannot reduce flash write operations because the page will be set dirty and written back to flash memory eventually. On the contrary, overwrites on dirty pages can cancel the write before they are actually flushed to the flash memory, which is called write cancelation.

Since flash writes are much slower than reads, all the algorithms designed for flash memory strive to reduce the slow flash write operations to some extent. Nevertheless, CCF-LRU and PT-LRU are able to write back fewer pages to the flash memory at the cost of decreased hit rates on clean pages as

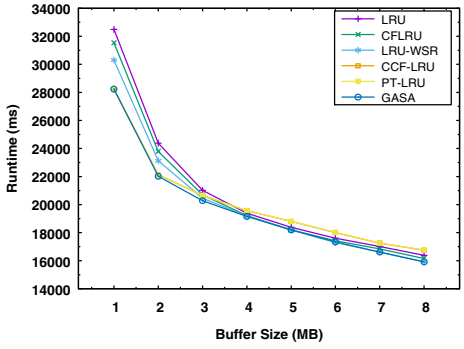
shown in Figure 11, leading to much more flash read operations than other algorithms, which can adversely impact the overall performance. On the other hand, GASA makes a trade-off between the flash writes and the hit ratios, striving to achieve higher I/O performance. Further experimental results will be described in the next subsection.



(a) T1



(b) T2



(c) T3

Fig. 13. Overall runtime for the traces under various buffer sizes.

3) Overall Runtime

Figure 13 shows the overall runtime of each page replacement algorithm under various workloads and buffer sizes. The overall runtime is comprised of the algorithms' running time and the sum of I/O execution time in the flash memory. While the running time of the algorithms is much small compared with the total time consumed by flash I/O operations, the overall runtime is dominated by the time of read/write operations to the flash memory. As shown in Figure 13, CFLRU and LRU-WSR have less runtime than LRU because they reduce flash write counts and maintain the buffer hit ratios comparable to those of LRU. Although CCF-LRU and PT-LRU have the least number of flash writes, they show even more runtime than LRU in most cases due to the severe decrease of read hit ratios as mentioned before. GASA reduces the runtime by up to 14.6%, 16.1%, and 13.1% compared with LRU under T1, T2, and T3, respectively. In addition, GASA outperforms all the other replacement algorithms because it effectively reduces the number of flash writes and maintains even higher hit ratios at the same time.

The results for the traces T2/T3 are interesting. As shown in Figure 13 (b) and (c), when the buffer size is sufficiently small (e.g. 1MB), PT-LRU and CCF-LRU have less overall runtime than CFLRU and LRU-WSR. However, with the increase of the buffer size, the overall runtime of PT-LRU and CCF-LRU become longer than those of other algorithms. To demonstrate the reason clearly, Figure 14 plots the detailed flash read/write time (obtained above the FTL) of the different algorithms under trace T2 with the buffer size set to 1MB, 2MB, 4MB, and 8MB. When enlarging the buffer cache, the costs due to the increasing read operations catch up with the benefits of reducing the write operations for CCF-LRU and PT-LRU. The proposed GASA achieves a better trade-off between the hit ratios and the flash write counts under different buffer sizes due to the utilization of the ghost buffer and the self-tuning mechanism. As a result, GASA outperforms all the other algorithms consistently.

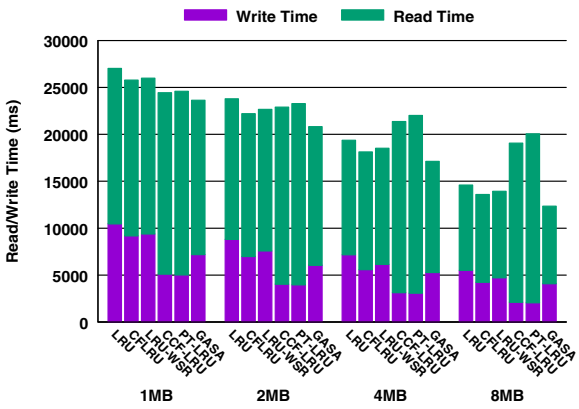


Fig. 14. Detailed read/write time for the trace T2.

4) Discussion and Future Work

According to the experimental results, CFLRU and LRU-WSR avoid severe degradation of hit ratios but only reduce the overall runtime slightly. CCF-LRU and PT-LRU significantly reduce the flash writes but suffer from poor performance under some workloads. Benefit from the ghost buffer and the learning scheme, GASA can notably reduce the write counts while maintaining high hit ratios, outperforming existing algorithms in terms of overall runtime under various workloads. However, there are also limitations to be addressed in the future. For example, it would be an interesting topic to be studied to combine GASA with the FTL caching policy [25], [26]. We also plan to implement GASA in a real platform to evaluate its performance under more workloads.

V. Conclusion

This paper presents an efficient page replacement algorithm called GASA for NAND flash memory based storage systems. While the write operation is much slower than the read operation in flash memory, the buffer caching algorithms should reduce the flash write operations without significantly increasing read operations. To achieve this goal, GASA keeps more hot dirty pages in the buffer cache by evicting cold clean pages preferentially and maintains reasonable buffer hit ratios by identifying and protecting potential hot pages by using a ghost buffer. In addition, a simple learning mechanism is developed to manage the ghost buffer automatically and intelligently. A series of trace-driven evaluations are conducted to show the effectiveness of GASA. Experimental results demonstrate that GASA efficiently improves the overall I/O performance in comparison with the existing algorithms.

Acknowledgment

This work was supported by the National Basic Research 973 Program of China under Grant No. 2011CB302301; 863 Project No. 2013AA013203, No. 2015AA015301, No. 2015AA016701; Fundamental Research Funds for the Central Universities, HUST, under Grant No. 2015MS073; NSFC No. 61502190, No. 61173043, No.61303046, No. 61472153; This work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China.

References

[1] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," ACM Comput. Surv., vol. 37, no. 2, pp. 138–163, Jun. 2005.

[2] G. Lawton, "Improved flash memory grows in popularity," Computer, vol. 39, no. 1, pp. 16–18, Jan 2006.

[3] J. Ren, C.-J. M. Liang, Y. Wu, and T. Moscibroda, "Memory-centric data storage for mobile systems," in Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, ser. USENIX ATC '15. Berkeley, CA, USA: USENIX Association, 2015, pp. 599–611.

[4] L. Wu, N. Xiao, F. Liu, Y. Du, S. Li, and Y. Ou, "Dysource: A high performance and scalable nand flash controller architecture based on source synchronous interface," in Proceedings of the 12th ACM International Conference on Computing Frontiers, ser. CF ' 15. New York, NY, USA: ACM, 2015, pp. 25:1–25:8.

[5] H. Jeon, K. El Maghraoui, and G. B. Kandiraju, "Investigating hybrid ssd ftl schemes for hadoop workloads," in Proceedings of the ACM International Conference on Computing Frontiers, ser. CF '13. New York, NY, USA: ACM, 2013, pp. 20:1–20:10.

[6] T. Xie and J. Koshia, "Boosting random write performance for enterprise flash storage systems," in Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on, 2011, pp. 1–10.

[7] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," Trans. Storage, vol. 8, no. 4, pp. 14:1–14:25, Dec. 2012.

[8] S. Park and K. Shen, "Fios: A fair, efficient flash i/o scheduler," in Proceedings of the 10th USENIX Conference on File and Storage Technologies, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 13–13.

[9] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. Siegel, and J. Wolf, "Characterizing flash memory: Anomalies, observations, and applications," in Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, Dec 2009, pp. 24–33.

[10] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in Proceedings of the 20th International Conference on Very Large Data Bases, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450.

[11] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in Proceedings of the 2Nd USENIX Conference on File and Storage Technologies, ser. FAST '03. Berkeley, CA, USA: USENIX Association, 2003, pp. 115–130.

[12] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," IEEE Trans. Comput., vol. 50, no. 12, pp. 1352–1361, Dec. 2001.

[13] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," SIGMOD Rec., vol. 22, no. 2, pp. 297–306, Jun. 1993.

[14] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: A replacement algorithm for flash memory," in Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 234–241.

[15] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "Lru-wsr: integration of lru and writes sequence reordering for flash memory," Consumer Electronics, IEEE Transactions on, vol. 54, no. 3, pp. 1215–1223, August 2008.

[16] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue, "Ccf-lru: a new buffer replacement algorithm for flash memory," Consumer Electronics, IEEE Transactions on, vol. 55, no. 3, pp. 1351–1359, August 2009.

[17] J. Cui, W. Wu, Y. Wang, and Z. Duan, "Pt-lru: a probabilistic page replacement algorithm for nand flash-based consumer electronics," Consumer Electronics, IEEE Transactions on, vol. 60, no. 4, pp. 614–622, Nov 2014.

[18] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng, "Improving flash-based disk cache with lazy adaptive replacement," in Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on, May 2013, pp. 1–10.

[19] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings," in Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 229–240.

[20] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101)," Parallel Data Laboratory, p. 26, 2008.

[21] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in USENIX 2008 Annual Technical Conference, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70.

[22] OLTP trace from UMass Trace Repository, *http://traces.cs.umass.edu/index.php/Storage/Storage*.

[23] S. Liang, K. Chen, S. Jiang, and X. Zhang, "Cost-aware caching algorithms for distributed storage servers," in Proceedings of the 21st International Conference on Distributed Computing, ser. DISC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 373–387.

[24] J. Boukhobza, I. Khetib, and P. Olivier, "Characterization of oltp i/o workloads for dimensioning embedded write cache for flash memories: A case study," in Proceedings of the First International Conference on Model and Data Engineering, ser. MEDI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 97–109.

[25] R. Chen, Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan, "On-demand block-level address mapping in large-scale nand flash storage systems," IEEE Transactions on Computers, vol. 64, no. 6, pp. 1729–1741, June 2015.

[26] H. Shim, B. K. Seo, J. S. Kim, and S. Maeng, "An adaptive partitioning scheme for dram-based cache in solid state drives," in 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), May 2010, pp. 1–12.