International Journal of Applied Engineering Research, ISSN 0973-4562 Volume 13, Number 5 (2018) pp. 3128–3138
© Research India Publications, http://www.ripublication.com

AFAPRA: Adaptive Flash Aware Page Replacement Algorithm

Lezwon Castellino
Department of Computer Science, Christ University, Bengaluru, Karnataka, India.

Joy Paulose
Department of Computer Science, Christ University, Bengaluru, Karnataka, India.

Abstract: NAND devices are being used in almost all portable devices today. In the recent past, there has been a tremendous increase in the adoption of NAND devices as a storage medium. Page replacement plays a crucial role within NAND devices as they contribute towards the performance of the system. Numerous algorithms have been proposed over the years to tackle the issues related to NAND memory based flash devices. Issues such as asymmetric read write speeds and erase before write operations, have been addressed by algorithms such as Ghost buffer Assisted and Self-tuning Algorithm (GASA), Smart Evicted Page List (SEPL), and Flash Aware Page Replacement Algorithm (FAPRA) and various others. However, the performance can be further enhanced by combining certain features within these algorithms. In this paper, a new algorithm called Adaptive Flash Aware Page Replacement Algorithm (AFPRA) is proposed. AFAPRA reduces writes to the flash devices by providing higher priority to write intensive pages. Also, this algorithm reduces cache pollution by increasing the priority of scattered pages. Long term pages are given higher preference within this algorithm as they might be referenced repeatedly after a certain period of time. This is done using an adaptive ghost list, which caches the metadata of recently evicted pages. Extensive experiments have been conducted on this algorithm and the results demonstrate that it outperforms the current state of the art algorithms in terms of hit ratio and the write count.

Keywords: Flash memory, Buffer cache, Page replacement, Paging, NAND.

INTRODUCTION

Page Replacement algorithms have been discussed by many researchers and numerous algorithms have been proposed over the years to increase the efficiency of the paging mechanism. Popular algorithms such as Least Recently Used (LRU), Least Frequently Used (LFU), 2Q [1] and Clock [2] schemes are developed for traditional magnetic storage devices. However, due the evolution of storage technology these algorithms are no longer considered to be efficient. The rise in the usage of micro Secure Digital (SD) and NAND storage technologies in smartphones have proved these old algorithms to be inefficient and slow [3]. To counter these problems, Flash Aware Page Replacement Algorithms (FAPRA) are introduced.

Data in magnetic disks could easily be over written and updated as per requirement. However, the technology in NAND storage devices works differently. NAND supports out-of-place update [4]. In other words, a page cannot be directly overwritten and updated in the SD card. Flash memory supports three types of operations: read, write and erase. The main drawback of NAND devices is the update operation wherein a page needs to be updated. The unit of operation to read and write a page is at the page level while the erase operation is done at the block level as depicted in Figure 1. Therefore, whenever a page is to be updated, its parent block has to be erased and overwritten, which may take a considerable amount of time. Due to this reason, write rates are comparatively slower in SD card devices.

In NAND devices, the cost of the erase operation is the highest. The cost for write operation is comparatively lower

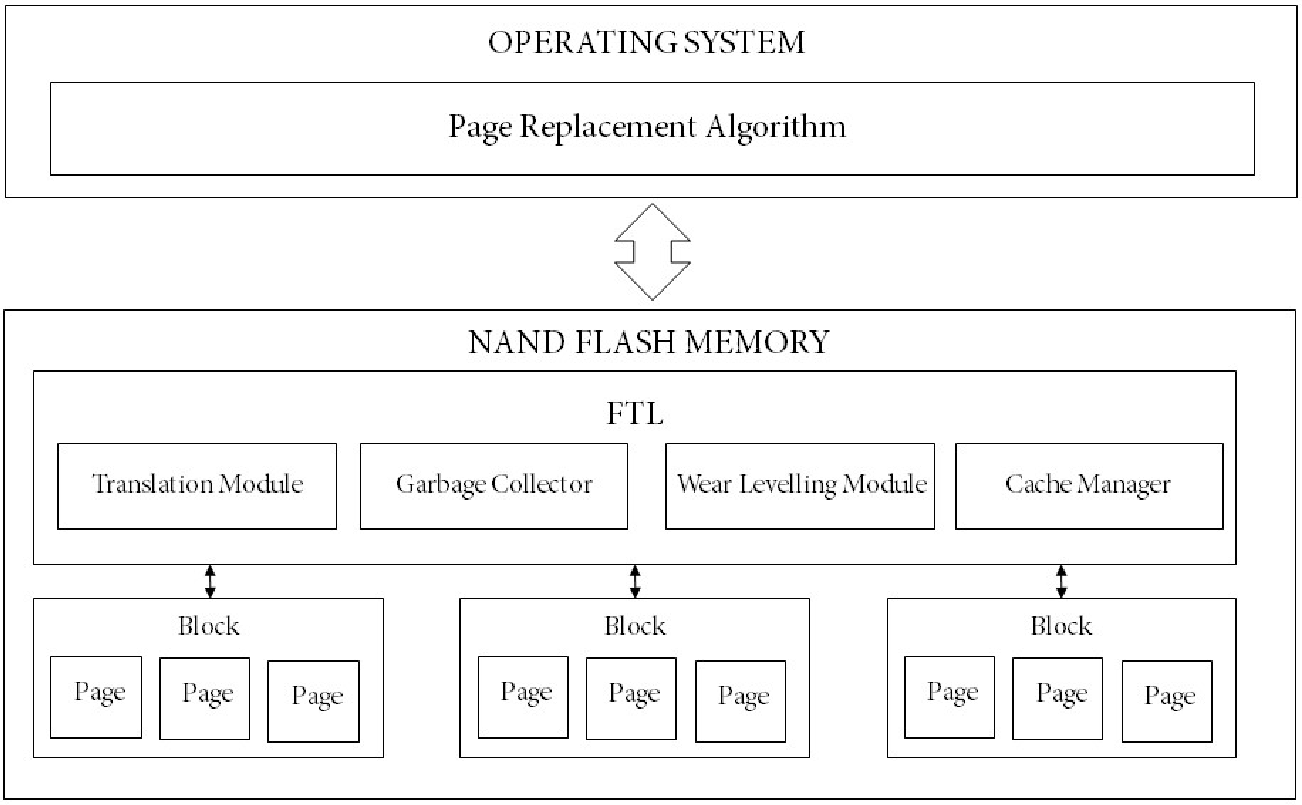

FIGURE 1. Architecture of NAND flash memory

Table 1. Characteristics of NAND Flash Memory

Operation	Granularity	Access Time
Read	2 KB	20 μs
Write	2 KB	200 μs
Erase	128 KB	1.5 ms

and the cost for the read operation is the lowest as shown in the Table 1. These devices follow the erase before write constraint where data needs to be erased at block level so that new data can be placed at the location. NAND devices invalidate the block within which the data has been updated and write the data to a new location. It maintains a table which maps the logical address of the pages to the physical address of the pages within the memory. The garbage collector is responsible for finding invalidated blocks and erasing them, for later reuse.

NAND devices make use of the Flash Translation Layer [5] (FTL) as an interface between the kernel and the storage devices. The flash devices are built and operate in a radically different way as compared to magnetic disks. However, due to the presence of the FTL layer, the kernel does not need to modify the file system interface to interact with flash devices. The FTL emulates the traditional magnetic disks to allow the system to interact with the device in a manner similar to that of magnetic disks. FTL also manages the garbage collection function within the flash device. The FTL is also built to deal with power failure, as it may occur at any moment and cause data corruption on the flash device.

Even with the added functionalities provided by Flash Translation Layer, the traditional page replacement algorithms cannot efficiently function on flash devices owing to its modern architecture. By considering all these drawbacks, it is quite evident that the traditional page swapping algorithms are not suitable for NAND devices. Hence, to cater to the needs of NAND devices, several page replacement algorithms such as Clean-First LRU (CFLRU) [3], LRU-Write Sequence Reordering (LRU-WSR) [6], Cold Clean First LRU (CCF-LRU) [7], Flash Aware Replacement Strategy (FARS) [8], Spatial-CLOCK [2], Smart Evicted Page List (SEPL) [9], Flash Aware Page Replacement Algorithm (FAPRA) [10] and Ghost buffer Assisted and Self-tuning Algorithm (GASA) [11] have been proposed. In the following section the popular SEPL [9], FAPRA [10] and GASA [11] algorithms are analyzed.

RELATED WORK

In this section, we shall review three page flash based replacement algorithms for flash devices. We shall note their advantages and drawbacks and see how the performance can be improved.

Ghost buffer Assisted and Self-tuning Algorithm (GASA)

GASA [11] is a recently introduced page replacement algorithm for Flash based storage devices that tries to reduce write operations to flash devices while maintaining a steady hit ratio. GASA does this by evicting cold clean pages before the others. The GASA algorithm also employs three lists: Mixed list (ML), Cold Clean List (CCL) and the Ghost List (GL) as shown in Figure 2. The Mixed list stores the hot clean, hot dirty and cold dirty pages. The Cold clean list stores the cold clean pages. The Ghost List or ghost buffer, stores the metadata of evicted pages. The algorithm automatically adjusts the size of the ghost buffer depending on the workload.

GASA classifies pages into four states: Hot clean, Cold Clean, Hot Dirty and Cold Dirty. Similar to CCF-LRU, GASA prefers the eviction of cold clean pages in order to reduce writes to the flash device. However, while CCF-LRU might reduce the hit count by evicting newly referenced pages, the GASA algorithm stores a reference of evicted pages within the ghost buffer for future reference [11]. If the pages in the ghost buffer are referenced again, GASA moves the content of the page directly to the buffer and marks it as hot. The hot pages are identified with this technique, thus increasing the hit rate.

During eviction, the cold clean list is checked first. If the list contains pages, then the page at the LRU position is selected as the victim page. Before it is replaced, its metadata is stored within the Ghost Buffer. If the CCL is empty, the algorithm starts scanning the ML for cold dirty pages, from the LRU position. When a cold dirty page is found, its contents are flushed to the disk and its metadata is recorded and inserted at the MRU position in the Ghost Buffer, before being replaced. If a Hot clean page is encountered, it is converted to a cold clean page and moved to the MRU of the Cold Clean List. As for Hot dirty pages, these are marked as Cold dirty pages and moved to the MRU of the Mixed List (ML). This enables the hot dirty pages to stay in the buffer and hence reduce the write frequency.

While the ghost buffer can help in identifying hot pages, it's size can be a matter of concern. For instance, if the size of the ghost buffer is too large, infrequent cold pages will have a higher chance of entering the MRU position of the Mixed List, which can increase the frequency of buffer misses. Similarly, if the ghost buffer size is too small, the buffer might fail to identify hot pages at a significant rate. Therefore, the ghost buffer size plays a very important role in the performance of this algorithm.

GASA addresses this issue by implementing a mechanism which automatically adjusts the size of the ghost buffer depending on the workload. GASA attaches a Ghost flag to the pages which is used to decide whether the buffer size has to be increased or decreased. A requested page, when found in the ghost list, is moved to the MRU position of the mixed list. A ghost flag is attached to the page to mark it as a page referenced from the ghost list. If the page is referenced again in future from the mixed list, the ghost flag will help us to note that the ghost list is successfully identifying hot pages. The size of the ghost buffer will therefore be increased to accommodate more hot pages. Once the list is enlarged, the ghost flag is cleared from the page. Similarly, during eviction, if the victim page exists in the ML with its ghost flag set, then it shows that the page hasn't been frequently accessed. The ghost buffer size is therefore reduced to avoid the entry of cold pages. GASA implements a strategy for ghost buffer which can benefit the hit ratio within an algorithm in the long run. This strategy is implemented within the proposed AFAPRA algorithm to increase its total hit ratio.

Smart Evicted Page List (SEPL)

SEPL [9] stands for Smart Evicted Page List. SEPL algorithm tries to overcome the few drawbacks of CFLRU and FARS by providing more priority towards write intensive and long term pages. SEPL contains a EPL (Evicted Page List) list which stores the metadata of recently evicted pages. The page list also contains a size adaptive window buffer which adjusts its size automatically according to the algorithm. Each page is attached with a SEPL factor that acts as a basis for evicting pages with the SEPL list as depicted in Figure 3.

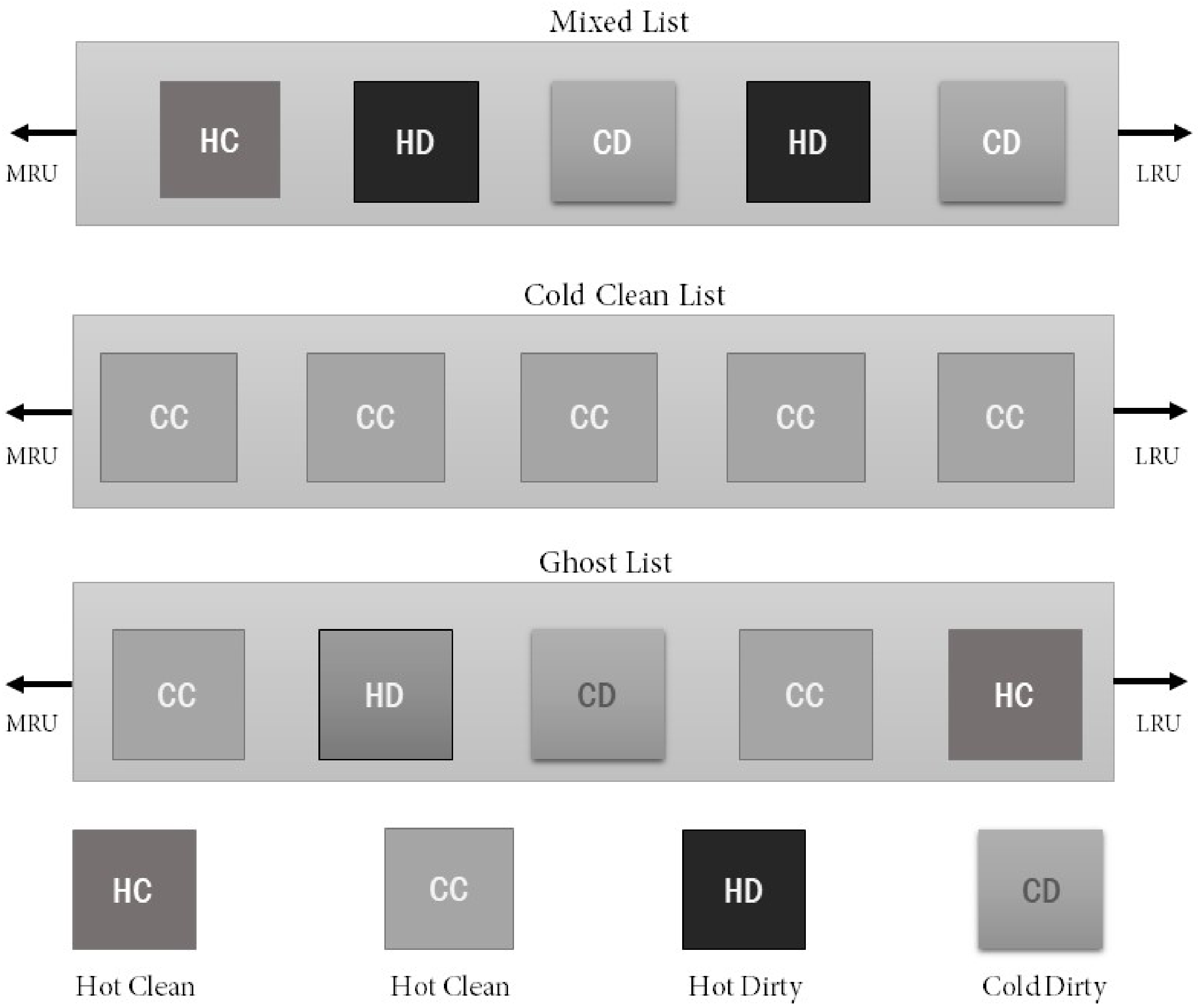

FIGURE 2. Example of GASA Algorithm

The SEPL algorithm initializes two constants ReadFactor and WriteFactor to 1 and 8 respectively [9]. When a page is requested, if it is found within the page list. It is moved to the MRU position of the list. If it is a write request, the SEPL factor for the page is increased by the WriteFactor(8) else by the ReadFactor(1). This is because the cost of the write operation is at least 8 times more than that of a read operation. If there is a miss, the EPL ghost list is scanned. If found, the page is moved to the MRU of the page list and the SEPL factor is increased by WriteFactor(8). This is done so as to reduce the eviction of long term pages. The page hits in the EPL list also provide a perspective on the eviction of page types. A clean page means that too many clean pages are being evicted, and therefore the window size is reduced. Similarly, if the page is dirty the window size is increased. If the page is not found in the EPL, the algorithm checks the flash device for the page. If the page is more than 32 sectors away, the page factor is increased by 8 so that randomly accessed pages stay longer in memory. A timer is set for every 1000 requests so as to reduce the priority of pages in the list.

SEPL focuses on three main issues. (1) SEPL attempts to reduce cache pollution by providing higher priority to pages that are scattered away from one another. (2) SEPL stores write intensive pages in buffer for a longer duration by providing them a high SEPL factor. (3) The algorithm provides higher priority to long term pages through the use of a ghost buffer. These are very beneficial solutions

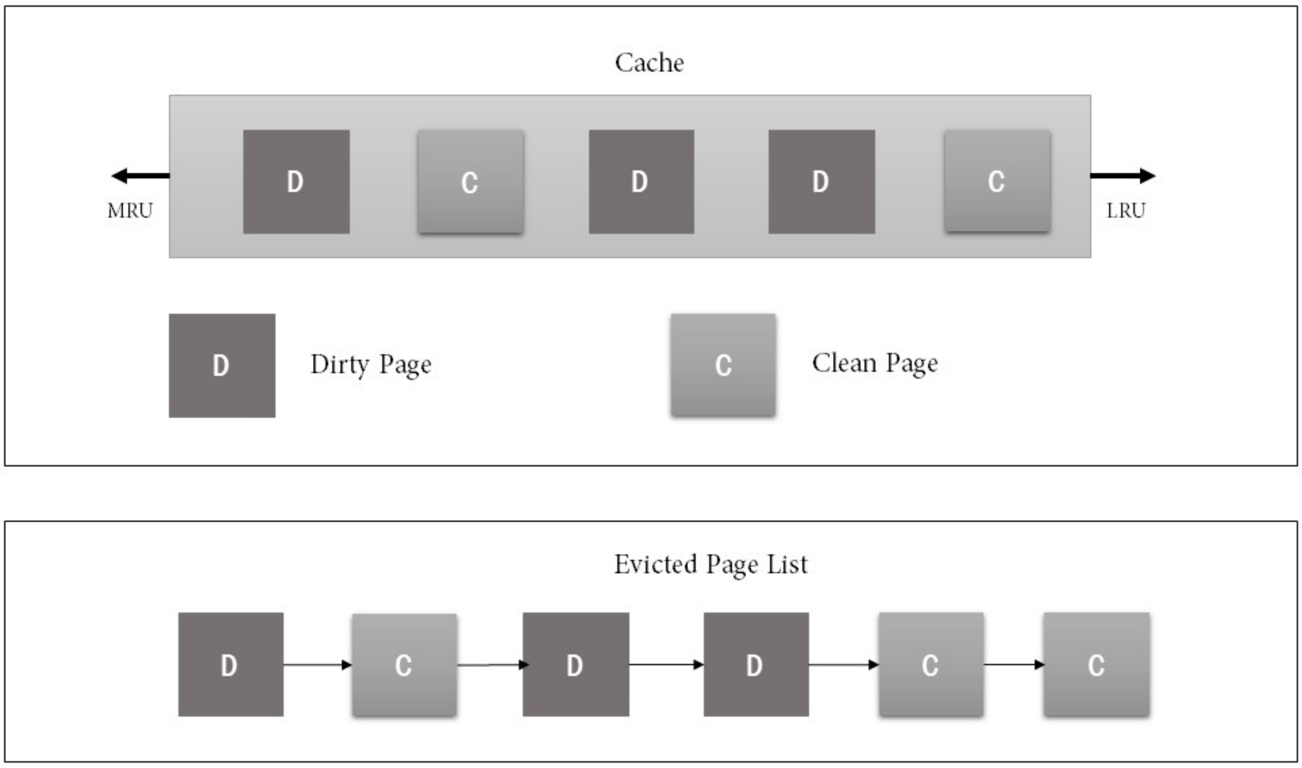

FIGURE 3. Example of SEPL Algorithm

implemented within the algorithm to increase its efficiency. The proposed algorithm, AFAPRA, utilizes these solutions to increase its hit ratio.

Flash Aware Page Replacement Algorithm (FAPRA)

FAPRA [10] focuses on reducing page writes to the disk and to reduce degradation of page hit ratio. FAPRA uses the access frequency of dirty pages to divide the pages into hot and cold ones. FAPRA also reduces the degradation on flash memory by reducing the number of writes to used blocks. The algorithm takes into account the access frequency and access recency to select victim pages. It utilizes a sub paging technique wherein instead of the entire page being written back to memory, only the dirty subpages are considered as shown in Figure 4.

This algorithm consists of two lists. The mixed list and the full dirty page list. The mixed list contains the clean pages and the partial dirty pages. A partial dirty page is a page where in not all subpages of the page are dirty. The dirty page list consists of the full dirty pages. These pages have no clean subpages and are therefore considered fully dirty. FAPRA introduces a benefit to cost ratio for evicting a victim page.

The Benefit-to-Cost Ratio (BCR) considers the free space that is saved and cost for reading the whole victim page candidate. The BCR for evicting a clean page is 1 as it has no dirty subpages while for evicting a full dirty page the BCR is 0. The combined recency and frequency values are also taken into account for devising the victim selection value using the BCE formula.

When a free frame is needed, the FAPRA algorithm is triggered. The algorithm first evicts a page from the mixed list, provided it is not empty. The page with the biggest Selecting Index Value (SIV) is evicted. However, if the mixed list is empty, the algorithm evicts a full dirty page with the biggest SIV. FAPRA checks for partial dirty subpages. If a page is found with clean subpages, FAPRA only writes the dirty subpages back to memory. In order to reduce the degree of wear levelling, the pages are written to a free block with the least erase count.

PROPOSED METHOD

In this section we shall present the idea and workflow of our algorithm AFAPRA.

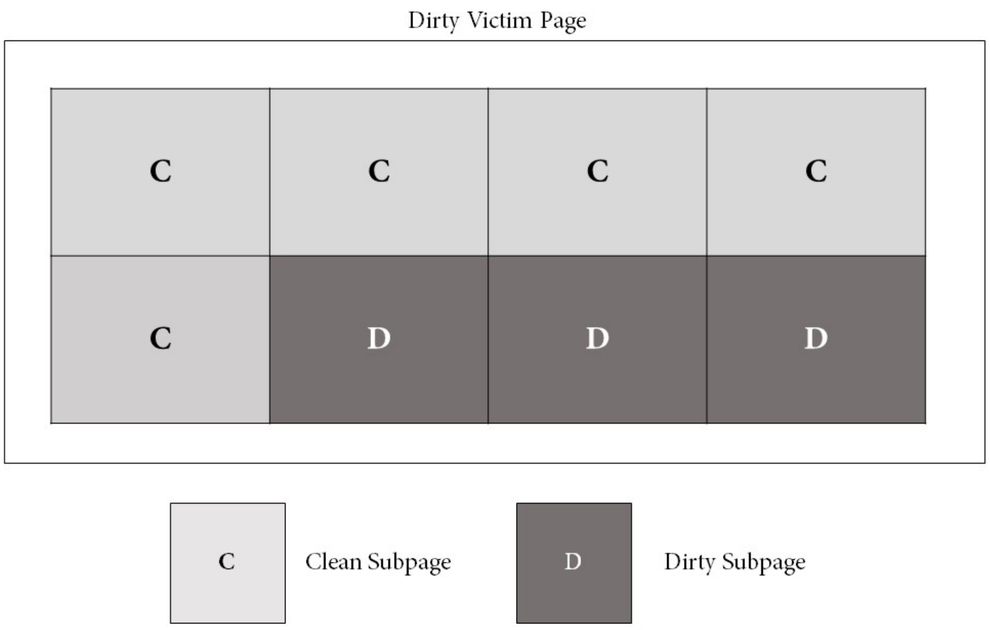

FIGURE 4. Example of a Dirty page in FAPRA

Structure

In order to improve the performance of these algorithms (GASA, FAPRA, SEPL), a new algorithm named Adaptive Flash Aware Page Replacement Algorithm (AFAPRA) is proposed. Similar to SEPL, AFAPRA will have a ghost list which will store the metadata of the pages evicted. The ghost list will be adaptive in nature. The algorithm will consist of three lists: the mixed list, the cold clean list and the ghost list. The mixed list consists of an eviction window. The window will automatically adjust later according to the page eviction scheme. The window size is initialized to 0.5 (half the size of the mixed list) at the start. The range the window size is permitted to vary from window/2 to window/5. The pages shall be differentiated on the concept proposed by Zhi Li, et al. [6]. The pages are classified into hot clean, hot dirty, cold clean and cold dirty. Figure 5 demonstrates the architecture of the AFAPRA algorithm. A hot clean page is a page that is frequently accessed and is not modified within the memory. It is the same copy as that in the storage device. A hot dirty page is a frequently accessed page that has been modified within memory. These pages should be given the highest priority to stay within memory. A cold clean page is a less frequently accessed page that is not modified within the memory while a cold dirty page is one that has been modified.

The cold clean list consists of all cold clean pages. The mixed list consists of cold dirty, hot clean and hot dirty pages. The Ghost list only consists of an evicted page's metadata. Each page contains two flag variables. The hot flag variable denotes whether the page in memory is a hot page (frequently accessed page) or not. The ghost flag denotes if the page in memory was retrieved from the ghost list. This will help us to adjust the ghost list size accordingly. The page will also contain a eFactor. The eFactor variable will act as the basis for the eviction criteria of the algorithm.

Algorithm

Page Request

In the proposed algorithm when a page is requested, the algorithm first checks the mixed list to see if the page exists. If the page is found and its hot flag set then it is moved to the MRU position of the list. The eFactor of the page is increased by 1 so as to increase its priority. If the ghost flag of the page is set, it means that the ghost list is helping to restore long term pages back to memory. This in turn causes the size of the ghost list to be increased by 1 and the ghost flag on the page is cleared.

If the page is not found in the mixed list, the algorithm will search the page within the cold clean list. If the page is found within the cold clean list, it is moved to the MRU position of the mixed list. If the page is required for a write operation, its eFactor is set to 8 and it is then moved to

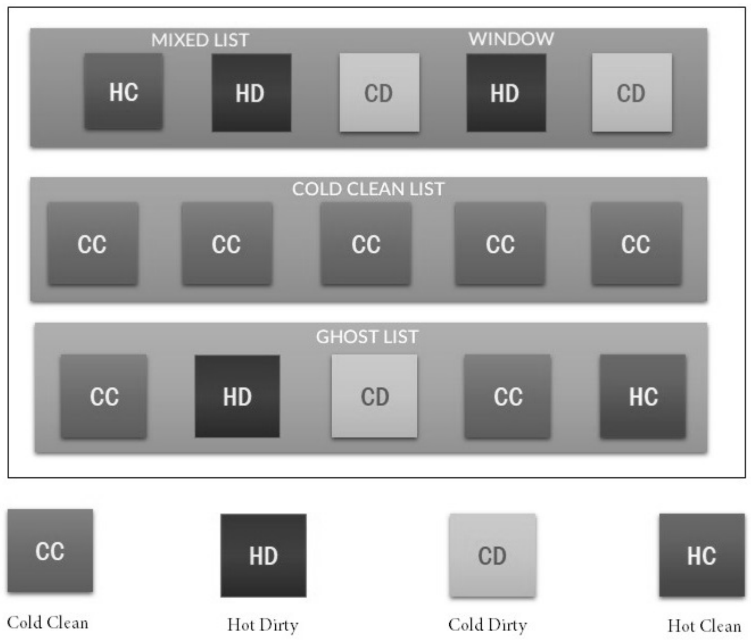

FIGURE 5. Architecture of AFAPRA Algorithm

the Mixed List, else 1 is added to the existing eFactor. The count of 8 is given so as to increase the priority for write intensive pages and to help them to stay in memory for a longer duration.

If there is no hit within the mixed list and the cold clean list, the algorithm will refer to the ghost list to check if the metadata of the page exists within it. If the page metadata is found within the ghost list, its data is loaded within a page, and its ghost flag is set. If the ghost page is hot when it is evicted, then its eFactor is set to 8, before moving it to the MRU of the Mixed List. This is done so as to retain long term pages within memory and to give them higher priority over short term pages. If the page was cold, an eFactor of 4 is set and the page is moved to the Cold Clean list. The count of 4 is chosen so as to keep a balance between the long term factor and cold factor. The page might be long term, but if its infrequently accessed, then it might hinder the allotment of new pages by remaining in memory for too long a time.

The algorithm monitors the clean factor of the page. If the page is clean, it means that too many clean pages are being evicted. The window size is therefore reduced by 1. Similarly, if the page was dirty, the window size is increased by 1 page so as to accommodate more dirty pages.

If the page is not found within any of the lists, then the data is retrieved directly from the flash device. If the page is required for a read operation then the eFactor of the page is set to 1. If it is required for a write operation, the eFactor of the page is set to 4. The location of the page is also vital. If the page is 32 sectors away from the last retrieved page, it means that the pages are being randomly accessed. Therefore, the eFactor of the page is increased by 4, to give priority to randomly accessed pages. The window size and ghost list size is adjusted after every page request. The page request algorithm is depicted in Algorithm 1.

Page Allocation

The page allocation procedure works as illustrated in Algorithm 2. If the cold clean list is not empty, the victim is selected from the LRU position of the cold clean list. If the Cold Clean list is empty, then the algorithm scans the window within the mixed list. The algorithm first scans the page at the LRU position. If its eFactor is 0, then it is chosen as the victim, else its eFactor is decremented by 1 and the algorithm scans the next page in the list. The algorithm iteratively scans every page within the window until a page with eFactor of 0 is found. Note that a page with an eFactor of 0 will eventually be found as the eFactor of a page is decremented whenever it is iterated over. The victim page is then removed form the list.

If the victim page has the ghost flag set, it means that too many cold pages are being retrieved from the ghost

Algorithm 1 Request

```
1: Initialize:
       $ML, CCL, GL$
2: Inputs:
       Page p, Operation op
3: if Page p is in ML then
4:     set hot-flag on p
5:     p.eFactor += 1
6:     move p to MRU of ML;
7:     if p.ghost-flag is set then
8:         ghostlist-size += 1;
9:         clear ghost flag on p;
10: else if Page p is in CCL then
11:     move page to MRU of ML
12:     if  op == write  then
13:         set p.eFactor = 8
14:     else
15:         p.eFactor += 1
        Page p is in GL
16:     p.data = read data from device
17:     set ghost-flag on p
18:     if p.dirty-flag is set then
19:         window-size += 1
20:         clear dirty-flag on p
21:     else
22:         window-size -= 1
23:     if p.hot-flag is set then
24:         set p.eFactor = 8
25:         move to MRU of ML
26:     else
27:         set p.eFactor = 4
28:         move to MRU of CCL
29: else
30:     p = alloc()
31:     read data from device into p
32:     if op == read then
33:         p.eFactor = 1
34:     else
35:         p.eFactor = 4
36:     if p is 32 sectors away then
37:         p.eFactor+=4
38:     move p to MRU of CCL
39:     adjustWindowSize()
40: return $p$
```

Algorithm 2 Alloc

```
1: if no free space in buffer then
2:     if CCL is not empty then
3:         victim = LRU page of CCL
4:     else
5:         while Page p.eFactor != 0 in window do
6:             p.eFactor -= 1
7:             continue
8:         victim = p
9:     if Page victim.ghost-flag is set then
10:        ghostlist-size -= 1;
11:    if Page victim.dirty-flag is set then
12:        flush victim.data to device;
13:        discard victim.data
14:        add victim to MRU of ghost list
15: else
16:     victim = get free page from buffer
17: return victim
```

list, therefore the size of the ghost list is decreased by 1. Once the page is removed from a list, its data, if clean, is discarded. Else if the page is dirty, it is flushed to the NAND device and then discarded. The metadata on the other hand is stored within the ghost list. If the ghost list is full, the page at the LRU position is discarded, and the new page is added at the MRU position of the ghost list. The ghost list size is adjusted after the eviction process

PERFORMANCE EVALUATION

In this section the performance of the proposed algorithm AFAPRA, is analyzed using a flash device simulation utility named Flash-DBSim [12]. Flash-DBSim is a popular tool used to simulate the functioning of a NAND disk. It provides the ability to conduct trace driven performance evaluations of various buffer cache algorithms. Flash-DBSim has been used by multiple researchers to measure the performance of their algorithms [4], [7], [13]. The evaluation is done using pre-recorded traces available within the simulation tool. The performance of the proposed algorithm will be compared to other popular algorithms such as CFLRU, LRU-WSR and CCF-LRU.

Experiemental Setup

The experiments are conducted on a 2.5 Ghz Intel Core i5 CPU. The system contains 6 GB of inbuilt RAM and runs Windows 10 with 64-bit support.

Flash-DBSim is initialized with 512MB memory and a page size of 2KB. The block size is 128KB with each block consisting of 64 pages. Flash-DBSim comes with pre-recorded traces of 200,000 and 1,000,000 records each. The experiments are conducted with 200,000 records which can provide a reliable result.

The following three performance metrics are considered in the experiments. The write count ratio, hit count ratio and runtime which is captured at the end of the simulation. The

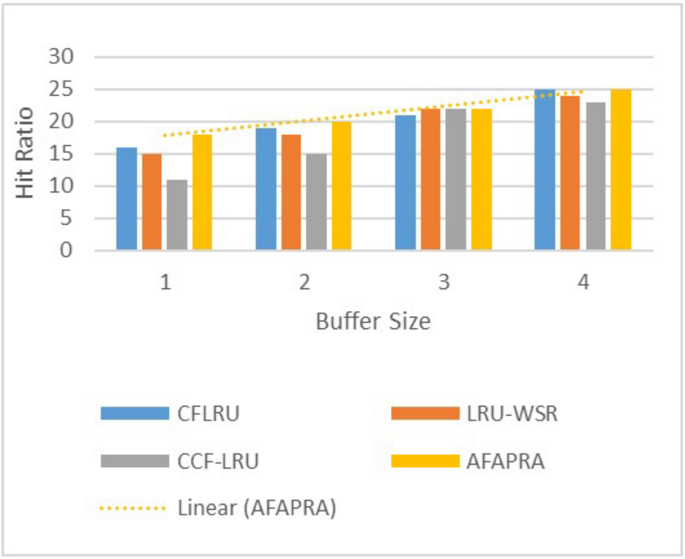

FIGURE 6. Comparison of Hit Ratio

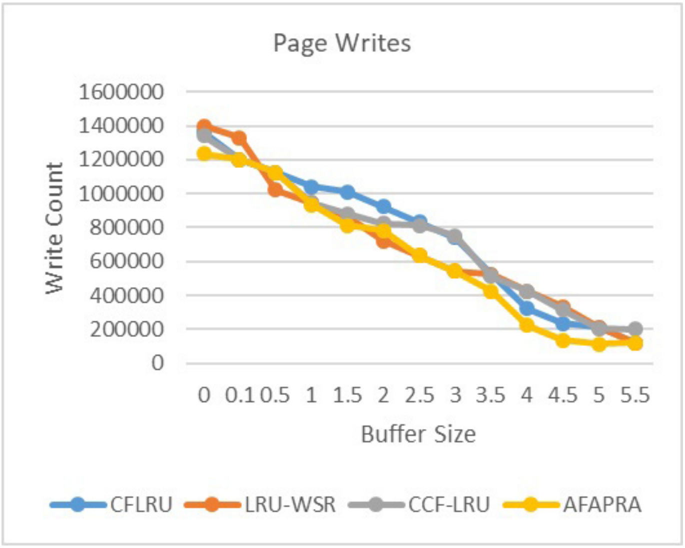

FIGURE 7. Comparison of Page write

window size for this experiment is set to 0.5 and the size of the ghost list is initialized to the same size as that of the ML and CCL list.

Experiemental Results

The simulation of these algorithms is conducted on Flash-DBSim and the results are analyzed. The results generated by the Flash-DBSim simulator are illustrated in Figure 8, Figure 9 and Figure 10, and are discussed in the following sub sections.

Hit Ratio

The Hit Ratio achieved by AFAPRA is comparatively higher compared to other algorithms. This is due to AFAPRA ghost list feature wherein recently evicted pages are stored longer within the memory. CF-LRU provides relative results during the hit ratio comparison of the algorithms. However, AFAPRA cache performance increases over time as it prioritizes the storing of frequently accessed pages within memory for longer durations. Figure 6 shows the hit ratio among the various algorithms.

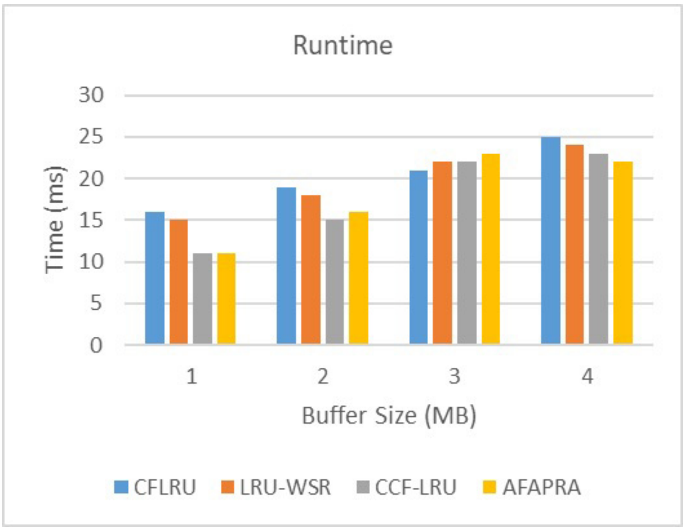

FIGURE 8. Comparison of Runtime performance

Write Count Ratio

AFAPRA performs better than the other algorithms with respect to the write count metric. The write count reduction provided by AFAPRA, is due to the mechanism which delays the eviction of write intensive page and the eFactor priority which stores them longer within memory. The write count is very low compared to that of LRU-WSR as shown in Figure 7. CFLRU and CCFLRU however struggle to match the performance criteria set by AFAPRA and LRU-WSR

Runtime

Figure 8 shows the comparison of runtime performance of AFAPRA with other similar algorithms. The time taken to execute the trace operation using the AFAPRA algorithm is lower than most of the other algorithms except for CCF-LRU. This is due to the complex conditions included in the AFAPRA algorithm to store pages that can be frequently accessed within the memory.

CONCLUSION

In this paper, a novel buffer replacement algorithm called AFAPRA is presented to enhance the performance of the flash memory. The buffer is divided into mixed and cold clean list. The algorithm also contains a ghost list to maintain a history of recently evicted pages. AFAPRA attempts to reduce write count by giving higher priority to pages in memory which are scattered at random location on the disk. It also reduces the chances of a frequently used page from being evicted by increasing its eFactor, every time it is used. The pages that are used repeatedly after a certain duration, are noted by the ghost list and is provided higher priority, so that they may stay in the buffer for a longer duration.

The proposed AFAPRA algorithm is tested on the flash simulation platform Flash-DBSim. The results are displayed in the Figure 6, Figure 7 and Figure 8. The results demonstrate that the performance of the AFAPRA algorithm supersedes other algorithms with respect to the performance parameters such as the hit ratio and write count.

In future work, this algorithm shall be tested using better data structures and a sub paging mechanism that may further contribute towards an improvement in its performance.

REFERENCES

[1] T. Johnson, D. Shasha, 2Q: A Low Overhead High Performance Management Replacement Algorithm, Proceedings of the 20th VLDB Conference (1994) 439–450 http://dx.doi.org/1-55860-153-8 doi:1-55860-153-8.

[2] H. Kim, M. Ryu, U. Ramachandran, H. Kim, M. Ryu, U. Ramachandran, http://dl.acm.org/citation.cfm?doid=2254756.2254786, What is a good buffer cache replacement scheme for mobile flash storage?, ACM SIGMETRICS Performance Evaluation Review 40 (1) (2012) 235. http://dx.doi.org/10.1145/

2318857.2254786 doi:10.1145/2318857.2254786. http://dl.acm.org/citation.cfm?doid=2254756.2254786

[3] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, J. Lee, http://portal.acm.org/citation.cfm?id=1176789CFLRU: a replacement algorithm for flash memory, Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (2006) 234–241 http://dx.doi.org/10.1145/1176760.1176789 doi:10.1145/1176760.1176789. http://portal.acm.org/citation.cfm?id=1176789

[4] P. Jin, Y. Ou, T. Härder, Z. Li, AD-LRU: An efficient buffer replacement algorithm for flash-based databases, Data and Knowledge Engineering 72 (2012) 83–102. http://dx.doi.org/10.1016/j.datak.2011.09.007 doi:10.1016/j.datak.2011.09.007.

[5] Intel, Understanding the Flash Translation Layer (FTL) Specification, Viewpoints (December 1998).

[6] H. Jung, H. Shim, S. Park, S. Kang, J. Cha, LRU-WSR: Integration of LRU and writes sequence reordering for flash memory, IEEE Transactions on Consumer Electronics 54 (3) (2008) 1215–1223. http://dx.doi.org/10.1109/TCE.2008.4637609 doi: 10.1109/TCE.2008.4637609.

[7] Z. Li, P. Jin, X. Su, K. Cui, L. Yue, CCF-LRU: A new buffer replacement algorithm for flash memory, IEEE Transactions on Consumer Electronics 55 (3) (2009) 1351–1359. http://dx.doi.org/10.1109/TCE.2009.5277999 doi:10.1109/TCE.2009.5277999.

[8] O. Kwon, H. Bahn, K. Koh, FARS: A page replacement algorithm for NAND flash memory based embedded systems, in: 2008 8th IEEE International Conference on Computer and Information Technology, 2008, pp. 218–223. http://dx.doi.org/10.1109/CIT.2008.4594677 doi:10.1109/CIT.2008.4594677.

[9] X. Jin, S. Jung, Y. H. Song, SEPL: Smart evicted page list buffer for NAND flash storage system, Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010 (Cit) (2010) 1687–1694. http://dx.doi.org/10.1109/CIT.2010.297 doi:10.1109/CIT.2010.297.

[10] G. Xu, L. Ren, Y. Liu, Flash-Aware Page Replacement Algorithm 2014 (1).

[11] C. Li, D. Feng, Y. Hua, W. Xia, F. Wang, Gasa: A new page replacement algorithm for NAND flash memory, 2016 IEEE International Conference on Networking Architecture and Storage, NAS 2016 - Proceedingshttp://dx.doi.org/10.1109/NAS.2016.7549403 doi:10.1109/NAS.2016.7549403.

[12] X. Su, P. Jin, X. Xiang, K. Cui, L. Yue, Flash-DBSim: A simulation tool for evaluating flash-based database algorithms, in: Proceedings - 2009 2nd IEEE International Conference on Computer Science and Information Technology, ICCSIT 2009, 2009, pp. 185–189. http://dx.doi.org/10.1109/ICCSIT.2009.5234967 doi:10.1109/ICCSIT.2009.5234967.

[13] H. Z. , P. J. , P. Y. , L. Y. , BPCLC: An Efficient Write Buffer Management Scheme for Flash-Based Solid State Disks, International Journal of Digital Content Technology and its Applications 4 (6) (2010) 123–133. http://dx.doi.org/10.4156/jdcta.vol4.issue6.15 doi:10.4156/jdcta.vol4.issue6.15.