Article

PA-LIRS: An Adaptive Page Replacement Algorithm for NAND Flash Memory

Fangjun Wang ^{1,*}, Xianliang Jiang ¹, Jifu Huang ¹ and Fuguang Chen ²

¹ School of Information Science and Engineering, Ningbo University, Ningbo 315211, China; jiangxianliang@nbu.edu.cn (X.J.); huangjifu@nbu.edu.cn (J.H.)

² NingBo Water Meter (Group) Company Limited, Ningbo 315211, China; cfg@chinawatermeter.com

* Correspondence: tracy_wangfj@163.com

Received: 21 October 2020; Accepted: 16 December 2020; Published: 17 December 2020

Abstract: NAND flash memory is increasingly widely used as a storage medium due to its compact size, high reliability, low-power consumption, and high I/O speed. It is important to select a powerful and intelligent page replacement algorithm for NAND flash-based storage systems. However, the features of NAND flash, such as the asymmetric I/O costs and limited erasure lifetime, are not fully taken into account by traditional strategies. In order to address these existing shortcomings, this paper suggests a new page replacement algorithm, called probability-based adjustable algorithm on low inter-reference recency set (PA-LIRS). PA-LIRS completely exploits the “recency” and “frequency” information simultaneously to make a replacement decision. PA-LIRS gives a greater probability to clean pages and a smaller probability to dirty pages when eviction selection happens. In addition, this proposed algorithm dynamically adjusts the parameter based on the workload pattern to further improve the I/O performance of NAND flash memory. Through a series of comparative experiments on various types of synthetic traces, the results show that PA-LIRS outperforms the previous studies in most cases.

Keywords: buffer replacement algorithm; page cache; NAND flash memory

1. Introduction

For several decades, magnetic disks have been overwhelmingly used as storage media in many fields. As the performance of operating systems continuously improves, the gap between the processor and the disk becomes more serious. Compared to magnetic disks, NAND flash memory shows a series of prominent advantages, such as its compact size, high reliability, low power consumption, and high I/O speed. With the increase in capacity and decrease in price, flash memory is dominating the storage industry in enterprise storage applications [1].

NAND flash memory is a type of electronic nonvolatile storage medium organized in blocks, each of which is generally 256 KB to 20 MB in size and consists of a given number of pages. Compared with magnetic disks, NAND flash memory has a shorter random access latency due to its non-mechanical seek movement, which helps to bridge the access speed disparity between the operating system and storage device. The architecture of NAND flash memory allows read and program commands to be executed on a page basis, and erase operations are performed at the level of a block that consists of multiple pages. An entire block must be erased before writing to any page. Generally, the speed of the write operation cannot keep up with the read operation, and the latency of the write operation is approximately seven times greater than that of the read operation. Apart from this, the erase operation requires a longer time than the write operation, and the considerable speed difference between them is usually on the order of magnitude. The lifetime of NAND flash memory is limited owing to its relatively small number of erase operations, typically between 10,000 and 100,000 cycles [2–4].

The flash translation layer (FTL) is the core software driver running on NAND flash memory, and it is used to make an optimal tradeoff between the operating system and the NAND flash memory. However, when an application performs rewriting or in-place update operations, FTL will cause new data to be written to different physical pages on the flash memory, even different physical blocks [5,6]. Therefore, decreasing the number of write operations can further improve access performance and extend the lifetime of NAND flash memory.

Setting caches for data pages between the operating system and the NAND flash memory can greatly promote the performance of the database. When the data page requested by the I/O operation is exactly in the cache, there is no need to access it from the NAND flash memory. Generally, static random-access memory (SRAM), whose operation speed is the same order of magnitude as the operating system, is often used for the cache. However, the fatal fact is that SRAM is expensive and has difficulty reaching high integration, and its capacity is much smaller than that of NAND flash memory.

With the purpose of achieving better I/O performance of storage devices, great progress has been made in cache replacement algorithms [7–11]. All these algorithms are based on the principle that the upcoming access track can be predicted by historical information. However, the historical information is not fully explored by the existing strategies. Some traditional policies are dwelling on the promotion of the cache hit ratio to reduce the access count, but they discard the inherent feature of asymmetric costs on read and write operations. In NAND flash memory, leaving dirty pages in the cache is more profitable than keeping clean pages. However, in some traditional ways, many dirty pages are often removed, which will result in increasing unnecessary I/O costs and deteriorating the database performance of NAND flash memory.

To minimize the cost of writing dirty pages back to NAND flash memory, various algorithms have focused on reasonably increasing the read count to decrease the write count while avoiding a severe drop in the hit ratio [12–15]. LRU (Least Recently Used)-based algorithms focus on the last access time of the data page, while LFU (Least Frequently Used)-based algorithms emphasize the access frequency. However, state-of-the-art algorithms have not completely explored the “recency” and “frequency” information in the access history simultaneously, and they have failed to achieve excellent behavior. Some current studies are interested in giving priority to clean pages for replacement but place less attention on their potential hot access frequencies [16–18].

In this paper, we observe that there is a strong correlation between the access pattern and cache replacement management. A new buffer replacement algorithm, namely PA-LIRS (Probability-based Adjustable algorithm on Low Inter-reference Recency Set), is designed for NAND flash memory. PA-LIRS makes a distinction between the read and write latencies and strives to reduce the number of the write operations while still maintaining a suitable hit ratio. For this sake, the algorithm gives a greater probability to clean pages and a smaller probability to dirty pages when an evict operation occurs. In addition, the algorithm adopts a new mechanism to dynamically adjust a parameter with the workload to attain the utmost overall performance.

The rest of this paper is organized as follows: Section 2 describes some of the existing page replacement algorithms. Section 3 presents the background and detailed implementation of the proposed buffer replacement algorithm. Section 4 highlights the experimental results of PA-LIRS and compares them with conventional cache replacement algorithms. Section 5 concludes the whole study.

2. Related Works

There have been several approaches for cache replacement to fully exploit the advantages of NAND flash memory. Based on the asymmetry of I/O latencies, CFLRU (Clean First LRU) is the first enhanced algorithm that replaces the clean pages preferentially to reduce write and erase operations [19]. There are two LRU lists in the cache, called working list and clean-first list. The former maintains the recently accessed pages so as to increase the hit ratio. On the other hand, the latter with a window size of w maintains the candidate pages to be evicted, which are considered to have no further reference during their lifetimes. If there is no free space for incoming pages, the clean page closest to the LRU

location in the clean-first list will be selected to be replaced. The dirty page with the earliest access time will be driven out if there is no clean page within the list window. Replacing clean pages first is able to reduce the access cost of the memory and choosing an appropriate window size w is helpful to improve the hit ratio. However, dynamically adjusting the window size to make it suitable for the different access patterns is not easy. The buffer hit ratio will decrease if the window is too large, and extra flash access will be generated if the window is too small. In addition, the algorithm replaces clean pages prior to dirty pages regardless of their access frequencies, which will result in cache pollution by the dirty pages.

As an improvement of CFLRU, LRU-WSR (LRU and Writes Sequence Recording) discriminates the cold pages with a cold-detection method [20]. When the replacement occurs, a clean page or a dirty page with a cold flag may be selected according to the LRU order, while the hot dirty page should be inserted at the MRU (Most Recently Used) location and be assigned with a cold-flag to delay being evicted. When the cold, dirty pages in the buffer are re-referenced, they will be moved to the MRU location and marked as hot pages with cold-flags cleared. LRU-WSR makes sense of the dirty pages' frequencies and prevents them from occupying the cache for a long time. Consequently, LRU-WSR reduces the write count without serious degradation of the hit ratio. However, evicting a clean page regardless of its access frequency will incur extra access overheads. If there are many dirty pages with a long period of no reference residing in the cache, a hot clean page may be evicted the moment after it is read in, which will result in a greater cost of flushing buffered pages.

Compared with the two traditional algorithms described above, AD-LRU (Adaptive Double LRU) shows further improvements in performance [21]. AD-LRU pays attention to the page reference frequency as well as recency and tries to change the size of the cold page region to prevent a clean page from being evicted immediately after being referenced. First, AD-LRU divides the cache into two LRU lists, namely cold list and hotlist. Second, the sizes of the two lists are adjusted dynamically. If a page in the cold list is referenced again, then the algorithm will enlarge the hotlist and shrink the cold list. If a hot page is chosen as the driven page or a new referenced page is moved to the cold list, then the range of the hot region will be reduced. Third, the lowest limit of the cold list size lim_lc means that when the length of the cold list reaches lim_lc , it will select a victim page from the hotlist rather than the cold list. Fourth, during the evict procedure, AD-LRU will give priority to the least recently referenced clean page to be selected as the victim. If the pages residing in the list all have dirty flags, then a second-chance policy will be used for victim selection. Through an adaptive mechanism, AD-LRU adjusts the size of the double regions to adapt to different access patterns. However, this algorithm always replaces clean pages first, which will result in dirty pages staying in the cache for a longtime. In addition, it is difficult to select a proper lim_lc suitable for all workloads.

CF-ABR (Clean First Adaptive Buffer Replacement) is a new page replacement algorithm proposed in 2019. CF-ABR maintains four LRU lists, the first referenced page list L1 and frequently referenced page list L2 are in the buffer, and the two replaced page lists H1 and H2 are in the upper layer of the NAND flash memory [22]. The lengths of the above four lists are adjusted dynamically according to the variable named reference, which is arranged for each page to count the hit number. The clean pages in the LRU position of L1 or the clean pages with the reference of zero in L2 will be selected first for replacement. In the absence of a clean page in the cache, the dirty page in the LRU position of L1 or the dirty page with reference to zero in L2 will be replaced. H1 holds the pages evicted from L1, and H2 holds the pages evicted from L2. CF-ABR pays attention to the asymmetric I/O performance and always evicts clean pages first to achieve good access performance to some extent. Furthermore, the algorithm manages the frequency and recency of the page efficiently to enhance the hit ratio. However, as we can see, it is a nonnegligible cost for the algorithm to find the page with zero references in L2. In addition, the algorithm is not efficient for some workloads because it evicts clean pages in L1 with absolute priority.

3. PA-LIRS Algorithm

3.1. Background

The algorithm proposed in this paper is built on the base version of the LIRS (Low Inter-reference Recency Set) algorithm [23], and this section introduces the LIRS algorithm briefly.

LIRS employs a parameter named IRR (Inter-Reference Recency) to identify the reference locality of the pages [23]. IRR means the number of other unique pages accessed between two consecutive references to the same page. Figure 1 shows that the IRR of page 9 should be 2, as page 1 and page 2 are referenced between the last page 9 and the penultimate page 9. It is certain that a page with a big IRR will not be frequently used and should be replaced prior to the pages with a small IRR. On the other hand, LIRS uses a variable called R to quantify the recency of pages [23]. R (Recency) of a page refers to the number of unique pages referenced from the last access of this page to the current access of the flash memory. As shown in Figure 1, the R of page 9 is 2. LRU-based algorithms lack the consideration of other history access information in addition to recency, and they simply regard that the pages with large R values are impossible to be used soon. LIRS effectively uses multiple sources of history access information, responsively changes the status of all the referenced pages, and improves the I/O performance of the storage device [24].

Figure 1. Example of the page parameters IRR (Inter-Reference Recency) and R (Recency).

In the LIRS algorithm, all the accessed pages are classified into two sets: the high IRR (HIR) set and the low IRR (LIR) set. There are few spaces for the HIR set, and the pages located in the HIR set will be evicted soon. As can be seen in Figure 2, LIRS keeps two LRU queues, namely Q queue and S queue, which are used to register the R and IRR of the pages, respectively. The S queue holds all the LIR pages and HIR pages (there are two types of HIR pages, resident HIR pages mean the pages whose page data and metadata are all stored in the cache, and nonresident HIR pages mean the pages that only store metadata in the cache) whose R is smaller than the largest R of the LIR pages; the Q queue contains all the resident HIR pages. When a new page absent in the S queue is referenced, it will be set with the HIR state and placed at the top of the Q queue. When the page that is in the Q queue but not in the S queue is operated, it will be promoted to the top of the Q queue and retain its HIR state. When the page in the S queue is re-referenced, it will be promoted to the top of the queue with its state set to LIR. Once the evict process is executed, the HIR page at the bottom of the Q queue will be removed, and its status will be changed to a nonresident if the page is also in the S queue.

3.2. Base LIRS Policy

To better illustrate the replacement algorithm, LIRS consists of three sections: the insertion policy, the promotion policy and the victim selection policy.

3.2.1. The Insertion Policy

When a new page or a page without access history records is accessed, it will be placed at the top of the Q queue and S queue and marked as an HIR page. Other than that, when a nonresident HIR page with a long period of no-reference is accessed, it should be inserted at the top of the S queue, and its state is changed to LIR.

Figure 2. Architecture of LIRS (Low Inter-reference Recency Set) (including the S queue and Q queue).

3.2.2. The Promotion Policy

Upon a hit, the LIR page will be promoted to the top of the S queue. The HIR page that leaves the history mark in the S queue will be promoted to the top of the queue and marked as LIR. On the other hand, an HIR page that is not in the S queue will be promoted to the top of the Q queue without changing its state.

3.2.3. The Victim Selection Policy

The page at the bottom of the Q queue is deemed as the victim to be replaced, and its status should be changed to a nonresident if it is also in the S queue.

3.3. Proposed Policy

In the LIRS algorithm, dirty pages are considered to have the same probability of being replaced as clean pages when free space is needed. However, evicting dirty pages will result in more write and erase operations as well as the overall running cost. The goal of LIRS is to obtain a high buffer hit ratio regardless of the different I/O latencies. As a result, LIRS shows poor I/O performance in flash-based database operations in some cases. For the purpose of fully adapting to the asymmetry of the read and write operations while maintaining a high hit ratio, this paper first introduces an algorithm called probability-based LIRS (P-LIRS). P-LIRS is enhanced in the following ways: first, by evicting the clean page that is least recently and least frequently used to reduce the write operations; second, by leaving the dirty pages in the buffer until being selected as the victim candidates for the second time to avoid a serious drop in the buffer hit ratio during the replacement.

P-LIRS maintains two LRU queues similar to the base LIRS. The S queue contains all the LIR pages and HIR pages, and the Q queue contains all resident HIR pages. Different from the traditional policy, all the pages in the proposed algorithm are marked with a read-write state. When the free space is required for a new referenced page, the proposed policy will call the victim selection policy. The main idea of P-LIRS is as follows:

1. Using deep-cold-detection policy to assign a deep-cold flag to the cold pages in the Q queue;
2. Putting off evicting a dirty page that is considered as a non-deep-cold page.

When a page miss happens in the buffer, the eviction will be carried out to bring in free space. In the deep-cold-detection algorithm, a bit named “deep-cold-flag” is assigned to each page to mark whether the page is hot. During the evict procedure, the page at the bottom position of the Q queue will be checked first. If the page is clean, then it will be regarded as the victim and will be driven out to NAND flash memory regardless of its “deep-cold-flag”. On the other hand, if the page is dirty,

then the flag bit will take effect. The dirty page whose “deep-code-flag” is 0 will seize the opportunity to stay in the cache and will be inserted at the top of the queue, and its “deep-cold-flag” will be marked as 1. Alternatively, if the “deep-cold-flag” dirty page is chosen as the victim, then it will be replaced to avoid an excessive decrement of the hit ratio. Upon a hit, the LIR pages in the S queue will be moved to the top, and the pages appearing only in the Q queue will be inserted at the top location of the queue with their “deep-cold-flag” set to 0.

Figure 3 presents an example of the victim selection procedure of P-LIRS. In this example, we suppose that the LIR set is three pages in length, the HIR set is 2 pages in length, and the buffer is full in initial. When a new page reference takes place, the algorithm will manage the buffer, as shown in Figure 3. Taking Figure 3a as an example, when a new page 7 is written, the victim selection policy will first drive clean page 5 out of the cache and leave it as a nonresident HIR page in the S queue. Due to the deep-cold-detection algorithm, dirty page 3 will remain in the cache with its “deep-cold-flag” changed to 1. Second, the insertion policy will set page 7 as a dirty resident HIR page and place it at the top of the Q queue and S queue.

In the P-LIRS algorithm model, L_{hir} is the only parameter and represents the cache size for HIR pages. According to the document [23], the author demonstrates that the LIRS algorithm is not sensitive to L_{hir} and can achieve the optimal hit ratio and I/O performance when L_{hir} is 1% of the cache size. It is worth noting that this result is drawn up on the fact of neglecting the asymmetric I/O runtimes. If we take the asymmetric I/O overheads into account, then we will find that it is not reasonable to have a fixed L_{hir} in the algorithm. In the P-LIRS algorithm, the victim selection policy works in the Q queue with a cache size of L_{hir} , which largely determines the I/O performance of NAND flash memory. In a write-intensive workload, the overall access runtime is changed approximately with the hit ratio, but in a read-intensive workload, it is advisable to have a larger L_{hir} to give a wider work scope for the dirty pages with lower access frequency, which can reduce the write count. Based on the workload history information of R_{rw} (read/write ratio), we add an adjustable L_{hir} to P-LIRS to form an algorithm called PA-LIRS. L'_{hir} is the theoretical target value of L_{hir} in PA-LIRS. The algorithm adopts a self-learning scheme to make L_{hir} gradually reach the target value automatically. L'_{hir} can be calculated as follows:

$$L'_{hir} = \begin{cases} \frac{1+6 \times \ln(3 \times R_{rw}) / \sqrt{R_{rw}}}{10} \times L_{buf}, & R_{rw} > 1/3 \\ \frac{1}{10} \times L_{buf}, & R_{rw} \leq 1/3 \end{cases} \quad (1)$$

L_{buf} is the cache size given in the unit of page. From Equation (1), it can be found that the value of L'_{hir} is no less than 10% of the cache size in any case.

Figure 3. Cont.

Figure 3. Examples of the insertion policy, promotion policy and victim selection policy for probability-based LIRS (P-LIRS). (a) Upon accessing page 7, which is not in the S queue, page 5 is driven out, and page 3 is set with a deep-cold-flag; (b) upon accessing high IRR (HIR) page 3, which is in the S queue, the page's status is changed to low IRR (LIR), and a stack pruning is conducted. Upon accessing LIR page 4, which is in the S queue, the page is inserted at the top of the S queue, and a stack pruning is conducted. (c) Description of the pages that appear in (a) and (b).

Initially, L_{hir} is 1% of the cache size, which will result in the highest hit ratio. Then, every time we have accessed L_{buf} pages, we should calculate a new L'_{hir} with Equation (1). When accessing a page, the proposed algorithm calls the function `adjust_Lh_process()` to check up on the working parameter L_{hir} , which will decide whether to shrink or enlarge the HIR buffer size. If $L_{hir} > L'_{hir}$, then we will only select the victim page from the Q queue and prevent the LIR page from becoming HIR page for the sake of decreasing L_{hir} . In contrast, when $L_{hir} < L'_{hir}$, we will have an additional operation to turn the LIR page at the bottom of the S queue into an HIR page at the top of the Q queue to enlarge L_{hir} . In this way, the length of the HIR set is continually adjusted under various workloads.

Figures 4 and 5 show the pseudo-codes of PA-LIRS. The function `stack_pruning_process()` shown in Figure 5 performs the stack pruning operation, which is detailed in the study [23], and its purpose is to ensure that the page at the bottom of the S queue is marked with an LIR flag. There is no doubt that we can obtain the replacement page eventually after scanning through the whole cache for the second time because the page at the bottom of the Q queue is either clean or dirty with a deep-cold-flag.

Algorithm: victim_selection_process()
 Out: The victim page in the buffer

```

initialize: victim = NULL;
if there is no free space in the buffer
  while victim == NULL
    for page p from bottom of Q queue
      if the p is clean
        victim = p;
        break;
      else
        if deep-cold-flag of p is set
          victim = p;
          break;
        else
          set the deep-cold-flag of p;
          move p to top of Q queue;
    remove victim from Q queue;
  if victim is dirty
    flush victim to flash memory;
  if victim is in S queue
    clear deep-cold-flag of victim;
    set non-resident HIR to victim;
return victim;
  
```

Figure 4. The pseudo-codes of the victim selection policy of PA-LIRS.

Algorithm PA-LIRS_Access(page p)

Input: the requested page p

Output: The reference to the requested data page p

```

Initialize: the buffer is full
if p is in LIR set //buffer hit
  move p to the top of S queue;
  adjust_Lh_process(); //adjust the length of HIR set gradually
  //reach the target value
  stack_pruning_process(); // stack pruning operation
else
  if p is in HIR set //buffer hit
    clear deep-cold-flag of p;
    if p is in S queue;
      set LIR status to p;
      remove p from Q queue;
      move p to the top of S queue;
      adjust_Lh_process();
      stack_pruning_process(); //stack pruning operation
    else
      move p to the top of Q queue;
      move p to the top of S queue;
      adjust_Lh_process();
      stack_pruning_process(); //stack pruning operation
  else //buffer miss
    victim_selection_process(); //select a victim block
    fetch the data of page p from flash;
    if p is in S queue
      set LIR status to p;
      move p to the top of S queue;
      adjust_Lh_process();
      stack_pruning_process();
    else
      set HIR status to p;
      move p to the top of Q queue;
      move p to the top of S queue;
  if the access count mod cache size == 0
    Lh_com_self_learning_process(); //calculate the target length of HIR set
return the reference to p;
  
```

Figure 5. The pseudo-codes of the page management algorithm of PA-LIRS.

4. Discussion

In this section, we verify the performance of the PA-LIRS algorithm via a simulator. Various types of workload traces are provided to evaluate the characteristics of the algorithm: the cache-hit ratio, write count, and runtime. The significant performance differences of PA-LIRS are illustrated with changes in the model parameters. In addition, in order to validate the proposed algorithm, six well-known cache replacement algorithms, namely LRU, CFLRU, LRU-WSR, AD-LRU, LIRS and CF-ABR, are cited for comparison.

4.1. Experimental Environment

The following flash-based algorithm experiments are all implemented using the simulation platform, named Flash-DBsim [25]. The platform provides a frame work for making performance evaluations of various algorithms, and it can configure the virtual NAND flash memory for different features, such as different read and write costs and different buffer sizes. In this paper, we simulated the NAND flash with 64 data pages, and the size of each page is 2 KB, which is the same as that of the frame in buffer. The detailed characteristics are described in Table 1.

Table 1. Parameters of the NAND flash memory.

Parameter	Value
Page size	2 KB
Block Size	64 pages
Write latency	200 μ s/page
Read latency	25 μ s/page
Erase latency	1.5 ms/block

This paper employs both synthetic traces and real-world traces for performance evaluation. Five types of synthetic traces denoted by T1–T5 are listed in Table 2. Due to the extensive application, all of the five traces are based on the pseudorandom references with temporal locality and spatial locality, and they are generated according to the Zipf distribution. A read/write ratio “25%/75%” represents that in the trace, read and write operations account for 25 and 75 percent each, and a locality “60%/40%” means that 60% of the total references are executed on the 40% of the pages. The real-world trace used in this paper is from on-line transaction processing (OLTP) applications running at a financial institution provided by the Storage Performance Council [26]. The trace is widely used in recent studies, such as work [18]. Table 3 lists the detailed attributes of the OLTP trace.

Table 2. Details of the five synthetic traces.

Type	Total Buffer Requests	Unique Pages	Read/Write Ratio	Locality
T1	200,000	10,000	25%/75%	80%/20%
T2	200,000	10,000	98%/2%	60%/40%
T3	200,000	10,000	8%/92%	80%/20%
T4	200,000	10,000	85%/15%	80%/20%
T5	200,000	10,000	85%/15%	60%/40%

Table 3. Details of the on-line transaction processing (OLTP) trace.

Attribute	Value
Total buffer requests	502,775
Read/Write ratio	87%/13%
Unique pages	31,732

4.2. Experiment Results

In the P-LIRS experiment, in order to exhibit the influence of L_{hir} , we change the parameter from 1% of the buffer size to 10%, 30%, 50%, 70% and 90% to show the necessity and validity of L_{hir} adaptive adjustment mentioned above. Figure 6 presents how the three performance metrics, hit ratio, write count and total runtime, vary with various L_{hir} values under five synthetic traces and a 5 MB cache.

(a)

(b)

(c)

Figure 6. (a) Hit ratio, (b) write count and (c) total runtime curves under different L_{hir} values in P-LIRS.

Figure 6 indicates the results of the sensitivity experiment. Flash-DBsim is used to simulate the above five synthetic traces and the buffer hit ratios of P-LIRS with six different L_{hir} values are calculated

under the cache of 5 MB. From the curves shown above, we get the following conclusions. First, for all the workloads, the hit ratios decrease when L_{hir} increases. As demonstrated in the simulation data, when L_{hir} increases from 1% to 90% of the buffer size, the hit ratio is reduced by almost 3%. Second, P-LIRS is not sensitive to the change of L_{hir} . The hit ratio differences are quite small and reasonable to be fully accepted when L_{hir} varies from 1% to 90% of the buffer size. Third, we can find the result that the larger the reference locality the trace has, the higher the hit ratio it will obtain. All the mentioned observations are consistent with the research results of LIRS [23,27].

Write count is the number of the write operations sent to flash memory, and it is got through the following ways: by counting the times of dirty page replacements during the experiment and by calculating the total number of dirty pages in the cache that should be flushed back to flash when the experiment is finished. According to the five traces, when L_{hir} increases gradually, the read-intensive traces T2, T4 and T5 get decreasing write counts, while the write-intensive trace T3 suffers from the increase in a flash write count. The write count of T1 is not sensitive to L_{hir} . However, it is deemed that, in read-intensive traces, when R_{rw} is large enough, the write count is no longer reduced due to the drop in the hit ratio, as we can see that the degradation of the write count is gentler in T2 than in T5.

The total runtime is composed of the execution overhead of the algorithm itself and the sum of the physical runtime of all the operations delivered to flash memory. Because of the asymmetric read and write performance of flash memory, the changing trend in runtime on the various workloads is not consistent with the change of the hit ratio. With the increase of L_{hir} , T2, T4 and T5 have decreasing runtime values, while T3 has the opposite one.

In order to obtain the desired tradeoff between the hit ratio and the total runtime under various workloads, L_{hir} employed in P-LIRS is adjustable to the read/write ratio of the workloads. From the results shown in Figure 6, it is appropriate to calculate the L_{hir} target value by Equation (1), which is implemented in the proposed PA-LIRS algorithm.

In the following sections, we measure three characteristics of PA-LIRS and compare them with recently used page replacement algorithms, namely LRU, CFLRU, LRU-WSR, AD-LRU, LIRS and CF-ABR. According to the application scenario and the previous studies [19–23], we range the buffer cache from 1 MB to 5 MB. This paper exhibits the experimental results of the seven algorithms under five synthetic traces and the OLTP trace.

Figure 7 illustrates the buffer hit ratios when the replacement algorithms are under various workloads and with different buffer sizes. Compared to the other algorithms, LIRS achieves an outstanding hit ratio under all six traces, which means that it can effectively identify the potential hot blocks. Due to the consideration of the I/O asymmetry, the hit ratio of PA-LIRS is slightly lower than that of LIRS, but it is still higher than that of the other five algorithms because PA-LIRS fully takes into account the block access frequency and access time. Although CFLRU, LRU-WSR and AD-LRU are all improved from LRU, they give higher priority to the clean pages to be driven out when there is no free space, which will result in their hit ratios being lower than that of LRU in some circumstances. CF-ABR captures the frequency and recency of the page, and its hit ratio is better than that of LRU, CFLRU, LRU-WSR and AD-LRU in T1, T3 and T4. When the access locality of the trace is approximately 50%/50%, such as in T2 and T5, there is an almost equal probability for cold blocks and hot blocks to be accessed again in the future. CFLRU and CF-ABR evict clean pages first, without much consideration, which will obviously lead to a drop in the hit ratio in T2 and T5. The OLTP trace has a higher temporal locality than the synthetic traces; many hot pages are re-referenced within a small buffer size, resulting in high hit ratios. As a result, when the cache size increases from 1 MB to 5 MB, the increases of hit ratios in OLTP trace are slower than those in the five synthetic traces. As we can see, with different buffer sizes, the hit ratio in PA-LIRS outperforms the other six algorithms in most cases.

Figure 7. The buffer hit ratios of the different algorithms under 6traces. (a) T1 trace, (b) T2 trace, (c) T3 trace, (d) T4 trace, (e) T5 trace and (f) OLTP trace.

Figure 8 presents the flash write counts for different workloads and for different buffer sizes. Since the cost of the flash write operation is much higher than that of the read operation, all the strategies focus on reducing flash write operations. From the comparison, we can make the following observations. First, the larger the buffer size, the fewer write operations are performed on flash memory. The hit ratio has a close correlation with the times of access operation on the flash memory, and it decreases according to the shrinking of the cache size. Therefore, the physical write count reaches the largest value under the buffer size of 1 MB. Second, with the comparison of the other provided algorithms, LRU incurs the most write operations because it replaces the least recently used page regardless of whether the page is clean. Therefore, LRU leads to more dirty page operations than those algorithms that first evict clean pages (i.e., CFLRU, LRU-WSR, AD-LRU, CF-ABR, PA-LIRS). LIRS has fewer write operations than LRU because of its higher hit ratio. CFLRU, LRU-WSR, AD-LRU and CF-ABR attempt to reduce write operations at the cost of cutting down the hit ratio of clean blocks. According to the results, PA-LIRS reveals a better performance both in a flash write operation number and buffer hit ratio. It can be observed that PA-LIRS strives to reduce the write traffic without a sharp

increase in flash read count. Third, it is obvious that, under synthetic traces, compared to the other algorithms, PA-LIRS shows fewer write operations, and the reductions in read-intensive traces of T2, T4 and T5 are more significant than those in write-intensive traces of T1 and T3. The reason is that a larger L_{hir} will lead to a larger work scope for the dirty pages residing in the cache, which plays a more visible role in read-intensive workloads. Under OLTP trace, because of the high temporal locality, the working list in CFLRU, the hotlist in AD-LRU and the L2 list in CF-ABR are all filled with hot pages in a short time. In LRU order, the clean pages in the clean-first list (in CFLRU), cold list (in AD-LRU) and L1 list (in CF-ABR) are replaced prior to dirty ones, which makes the write counts under the three algorithms fewer than those under the other algorithms. LRU-WSR and PA-LIRS use the code-detection method to remove the dirty pages to prevent the cache from being occupied by dirty pages for a long time.

Figure 8. The flash write counts of different algorithms under 6traces. (a) T1 trace, (b) T2 trace, (c) T3 trace, (d) T4 trace, (e) T5 trace and (f) OLTP trace.

Because of the negligible execution time of the algorithm, the runtime is dominated by the sum of all the access operations. The total latency of the algorithm is decided by the hit ratio and write count. From Figure 9, we can find that among all seven algorithms, LIRS displays the highest hit ratios. However, in some cases, CFLRU, LRU-WSR, AD-LRU, CF-ABR and PA-LIRS show lower runtime; the reason is that they effectively reduce the number of the write operations while avoiding a severe decrease in the hit ratios. In read-intensive trace, T2, CFLRU and CF-ABR have longer runtimes, although they have fewer write counts than LRU, LRU-WSR, AD-LRU and LIRS algorithms. The

possible reason for this result may be that the reduction in write count is limited in the trace and the hit ratio is the capital factor influencing the runtime. Under OLTP trace, CF-ABR shows longer runtime than the other six algorithms because of its extremely low hit ratio. Above all, under all the types of workload traces and various buffer sizes, PA-LIRS outperforms all the other algorithms in most cases because it fully considers the impact of the buffer hit ratio and write count on the flash performance and achieves a good tradeoff between them.

Figure 9. The overall runtimes of the different replacement algorithms under 6 traces. (a) T1 trace, (b) T2 trace, (c) T3 trace, (d) T4 trace, (e) T5 trace and (f) OLTP trace.

To better exhibit the performance improvement of PA-LIRS compared with the other proposed buffer replacement algorithms, we list the results of T5 under the buffer size of 5 MB in Table 4. We can observe that the hit ratio of PA-LIRS is slightly lower than that of LIRS but higher than that of the other algorithms. The flash write counts reduced by PA-LIRS are up to 62.9%, 44.9%, 56.3%, 45.7%, 54.4% and 52.6% compared to LRU, CFLRU, LRU-WSR, AD-LRU, LIRS and CF-ABR, respectively. The runtimes reduced by PA-LIRS are up to 46.9%, 37.9%, 38.8%, 37.7%, 36.8% and 46.5% compared to LRU, CFLRU, LRU-WSR, AD-LRU, LIRS and CF-ABR, respectively.

Table 4. Details of the performance comparison under T5.

Algorithm	Hit Ratio (%)	Write Count (10^3)	Runtime (s)
LRU	68.34	20.49	5.47
CFLRU	57.9	13.8	4.67
LRU-WSR	67.89	17.43	4.74
AD-LRU	58.71	14.0	4.66
LIRS	70.81	16.67	4.59
CF-ABR	51.88	16.01	5.42
PA-LIRS	68.81	7.6	2.9

5. Conclusions

In buffer replacement algorithms, because of the asymmetry of the write and read operation overheads sent to flash memory, it is necessary to reduce the write count while avoiding a serious decline in the hit ratio. In this paper, we suggest a systematic algorithm named PA-LIRS for NAND flash memory. Similar to the base version of LIRS, PA-LIRS simultaneously explores recency and frequency information and splits the cache buffer into two queues. The pages in the Q queue are the candidates to be evicted. In PA-LIRS, we give a deep-cold-flag to dirty pages in the Q queue to give them a second chance to stay in the buffer. In order to obtain more performance improvements, the proposed algorithm develops a simple learning mechanism to manage the length of the Q queue automatically to reduce the costly write count and keep the hit ratio at a reasonable level. We perform a series of simulation experiments under various workload traces, and the results demonstrate that the learning mechanism of L_{hir} adjustment is reasonable and intelligent. The proposed algorithm can significantly improve the overall performance of NAND flash memory while effectively extending the lifetime of NAND flash memory.

Author Contributions: Conceptualization, F.W. and J.H.; investigation, F.W.; methodology, F.W. and X.J.; software and experiments, F.W.; validation, F.W., X.J., J.H. and F.C.; writing—original draft preparation, F.W.; writing—review and editing, F.W. and X.J.; project administration, F.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

- Ahn, S.; Hyun, S.; Kim, T.; Bahn, H. A Compressed File System Manager for Flash Memory Based Consumer Electronics Devices. *IEEE Trans. Consum. Electron.* **2013**, *59*, 544–549. [\[CrossRef\]](#)
- AlinezhadChamazcoti, S.; Delavari, Z.; Ghassem, S.G.; Asadi, H. On Endurance and Performance of Erasure Codes in SSD-Based Storage Systems. *Microelectron. Reliab.* **2015**, *55*, 2453–2467. [\[CrossRef\]](#)
- Fukami, A.; Ghose, S.; Luo, Y.; Cai, Y.; Mutlu, O. Improving the Reliability of Chip-Off Forensic Analysis of NAND Flash Memory Devices. *Digit. Investig.* **2017**, *20*, S1–S11. [\[CrossRef\]](#)
- Xie, T.; Koshia, J. Boosting Random Write Performance for Enterprise Flash Storage Systems. In Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), Denver, CO, USA, 23–27 May 2011; pp. 1–10.
- Du, C.; Yao, Y.; Zhou, J.; Xu, X. VBBMS: A Novel Buffer Management Strategy for NAND Flash Storage Devices. *IEEE Trans. Consum. Electron.* **2019**, *65*, 134–141. [\[CrossRef\]](#)
- Yao, Y.; Kong, X.; Zhou, J.; Xu, X.; Feng, W.; Liu, Z. An Advanced Adaptive Least Recently Used Buffer Management Algorithm for SSD. *IEEE Access* **2019**, *7*, 33494–33505. [\[CrossRef\]](#)
- Megiddo, N.; Modha, D.S. ARC: A Self-Tuning, Low Overhead Replacement Cache. In Proceedings of the USENIX File and Storage Technologies Conference (FAST), San Francisco, CA, USA, 31 March 2003; pp. 115–130.
- Bengar, D.A.; Ebrahimnejad, A.; Motameni, H.; Golsorkhtabaramiri, M. A page replacement algorithm based on a fuzzy approach to improve cache memory performance. *Soft Comput.* **2020**, *24*, 955–963. [\[CrossRef\]](#)

9. Zanoon, N.; Abu-Taieh, E.; Abu-Hamatta, H.S. A Novel Approach to Improve LRU Page Replace Algorithm. *J. Appl. Sci. Eng.* **2018**, *13*, 478–483.
10. Anwar, U.; Paik, J.Y.; Jin, R.; Chung, T.S. Log-buffer Aware Cache Replacement Policy for Flash Storage Devices. *IEEE Trans. Consum. Electron.* **2017**, *63*, 77–84. [[CrossRef](#)]
11. Man, D.P.; Lu, Q.; Wang, Y.; Wu, Y.; Du, X.J.; Guizani, M. An Adaptive Cache Management Approach in ICN with Pre-filter. Queues. *Comput. Commun.* **2020**, *153*, 250–263. [[CrossRef](#)]
12. Zheng, K.; Wang, J. Page Weight-Based Buffer Replacement Algorithm for Flash-Based Databases. In Proceedings of the 2018 International Computers, Signals and Systems Conference (ICOMSSC), Dalian, China, 28–30 September 2018; pp. 107–111.
13. Thakare, A.O.; Deshpande, P.S. Probabilistic Page Replacement Policy in Buffer Cache Management for Flash-Based Cloud Databases. *Comput. Inform.* **2019**, *38*, 1237–1271. [[CrossRef](#)]
14. Zhang, X.; Duan, X.N.; Yang, J.C.; Wang, J.Y. ARW: Efficient Replacement Policies for Phase Change Memory and NAND Flash. *IEICE T. Inf. Syst.* **2017**, *100*, 79–90. [[CrossRef](#)]
15. Kim, J.J. Hot/Cold Based Replacement Algorithm for Flash Memory Buffer Management. *J. Appl. Sci. Eng.* **2019**, *14*, 5072–5077.
16. Yuan, Y.; Shen, Y.; Li, W.; Yu, D.; Yan, L.; Wang, Y. PR-LRU: A Novel Buffer Placement Algorithm Based on the Probability of Reference for Flash Memory. *IEEE Access* **2017**, *5*, 12626–12634. [[CrossRef](#)]
17. Liu, M.; Yao, Z.; Huang, T. F-LRU: An Efficient Buffer Replacement Algorithm for NAND Flash-Based Databases. *Optik-Int. J. Light Electron. Opt.* **2016**, *127*, 663–667.
18. Li, C.; Feng, D.; Hua, Y.; Xia, W.; Wang, F. GASA: A New Page Replacement Algorithm for NAND Flash Memory. In Proceedings of the 2016 IEEE International Conference on Networking, Architecture and Storage (NAS), Long Beach, CA, USA, 8–10 August 2016; pp. 1–9.
19. Park, S.Y.; Jung, D.; Kang, J.U.; Kim, J. CFLRU: A Replacement Algorithm for Flash Memory. In Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), Seoul, Korea, 22–25 October 2006; pp. 234–241.
20. Jung, H.; Yoon, K.; Shim, H.; Park, S.; Cha, J. LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory. *IEEE Trans. Consum. Electron.* **2008**, *54*, 1215–1223. [[CrossRef](#)]
21. Jin, P.; Ou, Y.; Harder, T.; Li, Z. AD-LRU: An Efficient Buffer Replacement Algorithm for Flash-Based Databases. *Data Knowl. Eng.* **2012**, *72*, 83–102. [[CrossRef](#)]
22. Huang, Q.; Chen, R.; Lin, M.; Yang, C.; Li, X. Clean-First Adaptive Buffer Replacement Algorithm for NAND Flash-based Consumer Electronics. In Proceedings of the 17th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA-2019), Xiamen, China, 16–19 December 2019; pp. 1217–1223.
23. Jiang, S.; Zhang, X. LIRS: An Efficient Low Inter-reference Recency Set Replacement to Improve Buffer Cache Performance. In Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, Marina Del Rey, CA, USA, 15–19 June 2002; pp. 31–42.
24. Chang, H.P.; Chiang, C.P.; Yu, Y.C. An Adaptive Buffer Cache Management Scheme. In Proceedings of the 2016 International Computer Symposium (ICS), Chiayi, Taiwan, 15–17 December 2016; pp. 124–127.
25. Su, X.; Jin, P.; Xiang, X.; Cui, K.; Yue, L. Flash-DBSim: A Simulation Tool for Evaluating Flash-based Database Algorithms. In Proceedings of the 2009 2nd IEEE International Conference on Computer Science & Information Technology (ICCSIT), Beijing, China, 8–11 August 2009; pp. 185–189.
26. UMass Trace Repository. Available online: <http://traces.cs.umass.edu/index.php/Storage/Storage> (accessed on 28 November 2020).
27. Jung, H.; Yoon, K.; Shim, H.; Park, S.; Cha, J. LIRS-WSR: Integration of LIRS and Writes Sequence Reordering for Flash Memory. In Proceedings of the International Conference on Computational Science and Its Applications—ICCSA 2007, Kuala Lumpur, Malaysia, 26–29 August 2007; pp. 224–237.

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).