ABR: An Optimized Buffer Replacement Algorithm for Flash Storage Devices

Xian Tang[1(✉)], Na Li[2], and Qiang Ma[2]

[1] School of Economics and Management,
Yanshan University, Qinhuangdao 066004, China
txianz@163.com
[2] School of Information Science and Engineering,
Yanshan University, Qinhuangdao 066004, China

Abstract. Flash disks are being widely used as an important alternative to conventional magnetic disks, although accessed through the same interface by applications, their distinguished feature, i.e., different read and write cost makes it necessary to reconsider the design of existing replacement algorithms to leverage their performance potential.

We propose an adaptive cost-aware replacement policy based on average hit distance (*AHD*) to control the movement of buffer pages when hits occur, thus pages that are re-visited within AHD will stay still. Such a mechanism makes our method adaptive to workloads of different access patterns. The experimental results show that our method not only adaptively tunes itself to workloads of different access patterns, but also works well for different kind of flash disks compared with existing methods.

1 Introduction

Social computing related applications bring great impacts to our daily lives, while produce large volume of data that need to be processed more efficiently. To this end, researchers developed flash-based storage devices to accelerate the computation. Flash-based storage devices have been steadily expanded into personal computer and enterprise server markets with ever increasing capacity of their storage and dropping of their price. A flash disk usually demonstrates extremely fast random read speeds, but slow random write speeds, and the best attainable performance can hardly be obtained from database servers without elaborate flash-aware data structures and algorithms [1], which makes it necessary to reconsider the design of IO-intensive and performance-critical software to achieve maximized performance.

During the past years, researchers proposed many buffer replacement algorithms [2–11] addressing the asymmetric read and write operation of flash disks, these methods, however, cannot work well for workloads suffering from "temporal locality". Here "temporal locality" means that in practice, a workload of databases, Web servers and operating systems, usually contains some pages that are requested with high frequency within a short time, but will not be requested in the future. We call such pages *once-frequently-requested* pages and the remaining pages except *once-requested* pages *frequently-requested* pages.

W. Che et al. (Eds.): ICYCSEE 2016, Part I, CCIS 623, pp. 136–150, 2016.
DOI: 10.1007/978-981-10-2053-7_14

As *once-frequently-requested* pages will not be requested again in the future, keeping them in the buffer for a long time does not facilitate the improvement of system performance. Although existing methods [2–11] can tell the difference between pages that are requested only once and pages that are requested multiple times, they cannot tell the difference between *once-frequently-requested* pages and *frequently-requested* pages.

We propose an optimized buffer replacement strategy, namely ABR, which uses average hit distance (*AHD*) to control the movement of buffer pages when hits occur, thus pages that are re-visited within a distance less than AHD will stay still. As a result, *once-frequently-requested* pages can also be flushed out quickly. This mechanism makes ABR adaptive to workloads of different access patterns and can really improve the hit ratio of *frequently-requested* pages. Moreover, ABR maintains a buffer directory, namely, ghost buffer, to remember recently evicted buffer pages. The reference count of each entry in the ghost buffer is used to adaptively determine the length of the buffer list and the insertion position when it is re-visited, such that ABR can adaptively decide how many pages each list should maintain in response to an evolving workload.

2 Background and Related Work

2.1 Flash Memory

Flash disks usually consist of NAND flash chips. The three basic operations are read, write, and erase. Read and write operations are performed in unit of a page. Erase operations are performed in unit of a block, which is much larger than a page, usually contains 64 pages. NAND flash memory does not support in-place update, the write to the same page cannot be done before the page is erased. To overcome the physical limitation of flash memory, flash disks employ an intermediate software layer called Flash Translation Layer (FTL) to emulate the functionality of block device and hide the latency of erase operation as much as possible.

2.2 Buffer Replacement Policies

Assuming that the secondary storage consists of magnetic disks, the goal of existing buffer replacement policies is to minimize the buffer miss ratio for a given buffer size. Existing studies on magnetic disks, such as 2Q [12], ARC [13], LIRS [14], CLOCK [15], LRU-K [16], FBR [17] and LRFU [18] aim at improving the traditional LRU heuristic, which are not efficient when applied to flash disks due to the asymmetric access times.

The flash aware buffer policy (FAB) [3] maintains a block-level LRU list, of which pages of the same erasable block are grouped together. FAB is mainly used in portable media player applications where most write requests are sequential.

BPLRU [4] also maintains an block-level LRU list. Different from FAB, BPLRU uses an internal RAM of SSD as a buffer to change random write

to sequential write to improve the write efficiency and reduce the number of erase operation. However, this method cannot really reduce the number of write requests from main memory buffer.

Clean first LRU (CFLRU) [2] is a flash aware buffer replacement algorithm for operating systems. It was designed to exploit the asymmetric performance of flash IO by first paging out clean pages arbitrarily based on the assumption that writing cost is much more expensive. The LRU list is divided into two regions: the working region and the clean-first region. Each time a miss occurs, if there are clean pages in the clean-first region, CFLRU will select the least recent referenced clean page in the clean-first region as a victim. Compared with LRU, CFLRU reduces the write operations significantly.

Based on the same idea, [5] makes improvements over CFLRU by organizing clean pages and dirty pages into different LRU lists to achieve constant complexity per request. In CFDC [6], dirty pages that are close to each other in the dirty queue are grouped into different clusters. Compared with CFLRU, CFDC improves the write efficiency.

Different from the above methods, ACR [7] addresses the problem that the ratio of write to read of different flash disks may vary significantly, and is adaptive to different types of flash disks.

3 The ABR Policy

3.1 Data Structures

As shown in Fig. 1, ABR splits the LRU list into two LRU lists, i.e., L_C and L_D. L_C keeps *clean* pages and L_D *dirty* pages. Assume that the buffer contains s pages when it is full, then $|L_C \cup L_D| = s \wedge L_C \cap L_D = \emptyset$. Further, $L_C(L_D)$ is divided into $L_{CT}(L_{DT})$ and $L_{CB}(L_{DB})$, and $L_{CT} \wedge L_{CB} = \emptyset (L_{DT} \wedge L_{DB} = \emptyset)$, $L_{CT}(L_{DT})$ contains *frequently-requested* clean (dirty) pages while $L_{CB}(L_{DB})$ contains *once-requested* and *once-frequently-requested* clean (dirty) pages, and *frequently-requested* clean (dirty) pages that are *not* referenced for a long time.

The sizes of L_{CB} and L_{DB} will be dynamically adjusted with the change of access patterns, which are controlled by δ_C and δ_D, respectively.

The difference of data structure between ACR [7] and ABR lies in that we use a ghost buffer L_H in ABR to trace the past references by recording the page id of those pages that are paged out from L_C or L_D. Fixing this parameter is potentially a tuning question, in our experiment, $|L_H| = s/2$. The notions used in this paper are shown in Table 1.

3.2 Cost-Based Eviction

If the buffer is full and the currently requested page p is in the buffer, it is served without accessing the auxiliary storage, otherwise, we will select from L_C or L_D a page x for replacement according to the metrics of "*cost*", not clean or dirty. The *cost* associated to L_C (L_D), say C_{L_C} (C_{L_D}), is a weighted value denoting the overall replacing cost.

Table 1. Notations used in this paper

Notation	Description
L_C	the clean list
L_{CT}	the top portion of L_C
L_{CB}	the bottom portion of L_C
δ_C	the number of clean pages contained in L_{CB}
L_D	the dirty list
L_{DT}	the top portion of L_D
L_{DB}	the bottom portion of L_D
δ_D	the number of dirty pages contained in L_{DB}
L_H	the ghost buffer containing page id of evicted pages
C_r	the cost of reading a page from a flash disk
C_w	the cost of writing a dirty page to a flash disk
s	the size of the buffer in pages
ρ_{rd}	the number of physical read operations of L_D
ρ_{wd}	the number of physical write operations of L_D
ρ_{rc}	the number of physical read operations of L_C
ι_d	the number of logical operations of L_D
ι_c	the number of logical operations of L_C

The *basic* idea is that the length of L_C (L_D) should be proportional to the ratio of the replacement cost of L_C (L_D) to that of all buffer pages according to recent m requests, in our experiment, $m = s/2$. This ratio can be formally represented as Formula 1:

$$\beta = C_{L_C}/(C_{L_C} + C_{L_D}) \tag{1}$$

The policy of selecting a victim page can be stated as: If $|L_C| < \beta \cdot s$, it means L_D is too long, and the LRU page in L_D should be paged out, otherwise the LRU page of L_C should be paged out.

Hereafter, we call the read and write operations that are served in buffer are logical, and ones that reach the disk are referred to as physical. Logical and physical operations are two different operations, and they all affect the overall performance. Assume that n is the number of pages in a file and s the number of pages allocated to the file in the buffer. The probability that a logical operation will be served in the buffer is s/n, and the probability that a logical operation will be translated to a physical one is $(1 - s/n)$. The probability is used to compute the values of C_{L_C} and C_{L_D}, as shown by Formulas 2 and 3, for which the meaning of each notion is shown in Table 1.

$$C_{L_C} = (\iota_c \cdot (1 - s/n) + \rho_{rc}) \cdot C_r \tag{2}$$

$$C_{L_D} = \iota_d \cdot (1 - s/n) \cdot (C_w + C_r) + \rho_{rd} \cdot C_r + \rho_{wd} \cdot C_w \tag{3}$$

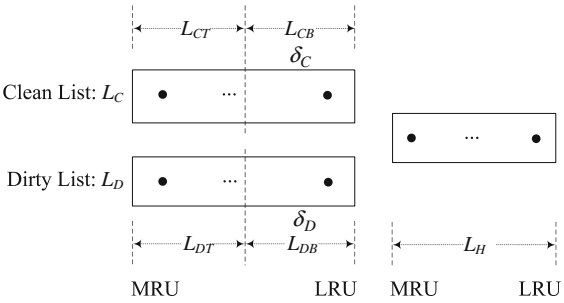


Fig. 1. Data organization of ABR

3.3 The ABR Algorithm

Simply using the above eviction strategy cannot solve the "temporal locality" problem. We introduce to use average hit distance to tackle this problem.

Average Hit Distance. For a sequence of requests $R =$ "$r_1, r_2, \cdots, r_n$", where each r_i refers to a data page, assume that r_i and $r_j (i < j)$ refer to the same page p and no other request $r_k (i < k < j)$ refer to p, we define the *Hit Distance* d of p between r_i and r_j is $p.d = j - i$. $p.d = 0$, if the number of requests ≤ 1 on p.

For example, for "$r_1(p_1), r_2(p_2), r_3(p_3), r_4(p_1), r_5(p_4), r_6(p_1)$", if the current request is r_4, we have $p_1.d = 3$ and $p_2.d = p_3.d = 0$. If the current request in r_6, then $p_1.d = 2$. To differentiate the two Hit Distances of p_1, we denote them as $r_4.d = 3$ and $r_6.d = 2$, similarly, we have $r_1.d = r_2.d = r_3.d = r_5.d = 0$.

Definition 1 ***(Average Hit Distance(AHD, ξ)).*** *For the recent n requests $R =$"$r_1, r_2, \cdots, r_n$", $R^+ =$"$r_1^{'}, r_2^{'}, \cdots, r_m^{'}$" is the set of m requests of R ($m \leq n$), where each request r of R^+ satisfies $r.d > 0$. The* Average Hit Distance ξ *of R is the average number of the Hit Distance of requests in R^+, as shown in Formula 4.*

$$\xi = \frac{\sum_{i=1}^{m} r_i.d}{m} \tag{4}$$

For example, assume the recent 8 requests are $R =$"$r_1(p_1)$, $r_2(p_2)$, $r_3(p_2)$, $r_4(p_3)$, $r_5(p_2)$, $r_6(p_4)$, $r_7(p_1)$, $r_8(p_2)$", we have $r_1.d = 0$, $r_2.d = 0$, $r_3.d = 1$, $r_4.d = 0$, $r_5.d = 2$, $r_6.d = 0$, $r_7.d = 6$, $r_8.d = 3$. According to Definition 1, the AHD for R is $\xi = (r_3.d + r_5.d + r_7.d + r_8.d)/4 = 3$.

Intuitively, if the Hit Distance of a request r on p is greater than or equal to AHD, i.e. $r.d \geq \xi$, it means that the interval between two continuously access to p is relatively large, thus p should be considered as a *frequently-requested* page, otherwise, if p will not be re-visited in the future, even if p is visited frequently within a short time, as its Hit Distance is less than AHD, p should be considered as a *once-frequently-requested* page instead of a *frequently-requested* page.

We use AHD to determine whether a hit on page p should be taken into account. Let r be the request corresponding to the hit on p, the idea of changing $p.hit$ can thus be stated as Formula 5.

$$p.hit = \begin{cases} p.hit, & r.d < \xi \\ p.hit + 1, & otherwise \end{cases} \tag{5}$$

However, there are still two problems that are needed to be solved before using the above method, otherwise, some frequently-requested pages with Hit Distance *less* than AHD will be wrongly moved out from the buffer, as shown in below.

$P1$: *where is the start position to compute Hit Distance*?

$P2$: *what is the upper bound of AHD*?

Example and Solution to $P1$. As shown in Fig. 2 (A), the dashed line represents a sequence of requests, where each large red dot denotes a request on page p, while each small black dot denotes a request on other page. d_1 to d_4 represent the Hit Distance of r_2 to r_5, respectively. If the current request is r_2, we have $r_2.d = d_1$. Since $d_1 < AHD$, $p.hit$ will not increase. Similarly, $p.hit$ will not increase when processing r_3, r_4 and r_5. Such case usually occurs if p is a page containing meta data or the root node of a B-tree index.

In our method, $p.hit$ will not increase when processing r_2, but will increase by 1 when processing r_3. The Hit Distance of r_3 is the distance from r_1 to r_3, instead of that between r_2 and r_3. That is, we use a flag to denote whether a certain request is a valid one. When processing r_2, since $d_1 < AHD$, r_2 is marked as an invalid request. When computing the Hit Distance of r_3, we will check the previous valid request of p, then we have $r_3.d{=}d_1^{'}{=}d_1{+}d_2{>}AHD$. Similarly, $p.hit$ will not increase when processing r_4, but will increase by 1 when processing r_5, because $r_4.d{=}d_3{<}\ AHD$, while $r_5.d{=}d_2^{'}{=}d_3{+}d_4{>}AHD$.

Example and Solution to $P2$. As shown in Fig. 2 (B), the thick line represents pages in the buffer. The circled blue numbers ① and ② represent two cases of AHD, the red dots denote the requests on page p. δ_C and δ_D are the size of L_{CB} and L_{DB}, respectively. Notice that the newly entered pages that have no page id in the ghost buffer are always inserted at the MRU position of either L_{CB} or L_{DB}.

Case ① ($AHD > \frac{\delta_C+\delta_D}{2}$): In this case, there may exist some request (e.g., r_2 on p) such that $\frac{\delta_C+\delta_D}{2} < r_2.d < AHD \wedge 2 \cdot r_2.d > (\delta_C + \delta_D)$. Thus r_2 is an invalid request and $p.hit$ doesn't increase by 1, thus p will not be moved to L_{CT} or L_{DT} when processing r_2. As a result, even if p is frequently revisited with a Hit Distance relatively large, p may be flushed out when processing r_3 in the future.

Case ② ($AHD \leq \frac{\delta_C+\delta_D}{2}$): In this case, for any request r_2 on p satisfying $r_2.d \leq AHD$, we have $2 \cdot r_2.d \leq (\delta_C + \delta_D)$. Thus in the worst case, p still can be moved to L_{CT} or L_{DT} when processing r_3.

Based on the above discussion, we use the following Formula to get AHD, not Formula 4.

$$\xi = \begin{cases} \frac{\sum_{i=1}^{m} r_i.d}{m}, & \xi \leq \delta_C + \delta_D \\ (\delta_C + \delta_D)/2, & otherwise \end{cases} \tag{6}$$

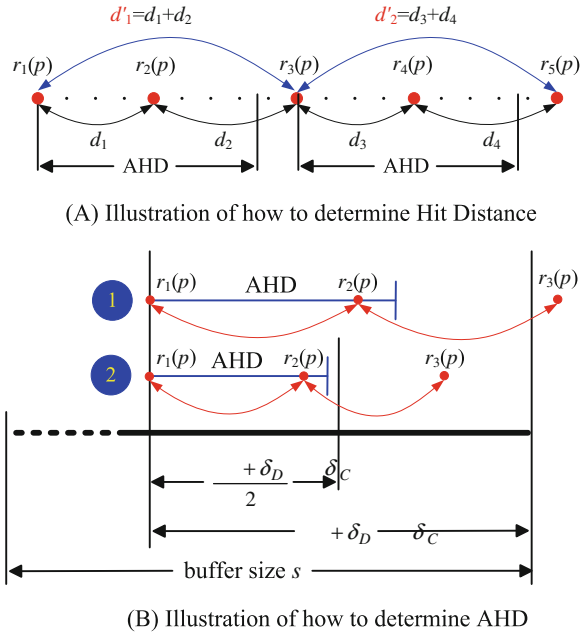


Fig. 2. Problems about Hit Distance and AHD. (Color figure online)

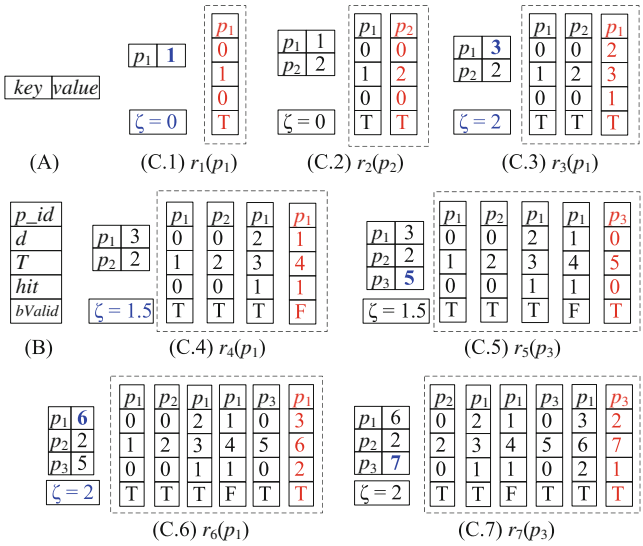


Fig. 3. Illustration of computing AHD. (A) is the structure of an item of a hash table H, (B) is the structure of an item of a priority queue Q, (C.1) to (C.7) show the process of a reference sequence.

AHD Computing. To compute the AHD of the past n requests, we use two data structures, a hash table H and a priority queue Q. For a reference sequence R, H is used to maintain the position of the most recent valid request on each page. Q is used to maintain the recent n requests. Figure 3 (A) shows the structure of an item of H, the *key* is a page id, the *value* is the position of an entry in Q, which denotes the start position to compute Hit Distance. Figure 3 (B) shows the structure of an item of Q, which consists of five member variables, p_{id} is the page id of p, d is the Hit Distance of r, T is the order of r in a reference sequence, *hit* is the reference count of p after processing r, $bValid$ is a flag denoting whether the current request r on p is a valid request. If $bValid$ = T, it means the position of p in H should be changed according to r.

Let R="$r_1(p_1)$,$r_2(p_2)$,$r_3(p_1)$,$r_4(p_1)$,$r_5(p_3)$,$r_6(p_1)$,$r_7(p_3)$" be a reference sequence. Assume $\delta_C + \delta_D = 4$ and $n = 6$ ($Q's$ length) in this example, i.e. we compute the AHD of the recent 6 requests. Initially, H and Q are both empty. The process of R is as follows.

(1) process $r_1(p_1)$. The status is shown in Fig. 3 (C.1).

(2) process $r_2(p_2)$. The status is as Fig. 3 (C.2).

(3) process $r_3(p_1)$. As shown in Fig. 3 (C.3), p_1 is hit and $r_3.T = 3$. From H in Fig. 3 (C.2) we know p_1's valid position is 1 and $\xi = 0$, then we have $r_3.d = 2$ and $r_3.hit = 1$ as $r_3.d \geq \xi = 0$. Thus $r_3.bValid$ =T and we change p_1's valid position in H to 3. Finally, we set the value of ξ as $\xi = r_3.d/1 = 2$.

(4) process $r_4(p_1)$. As shown in Fig. 3 (C.4), p_1 is hit and $r_4.T = 4$. From H in Fig. 3 (C.3) we know that p_1's valid position is 3 and $\xi = 2$, then we have $r_4.d = 1$ and $r_4.hit = 1$ as $r_4.d \leq \xi = 2$. Thus $r_4.bValid$ =F and p_1's valid position is still equal to 3. Finally, $\xi = \frac{r_3.d+r_4.d}{2} = 1.5$.

(5) process $r_5(p_3)$. It's the first time p_3 enters into Q, the value of its member variables is shown in Fig. 3 (C.5), p_3's valid position in H is set to 5.

(6) process $r_6(p_1)$. As shown in Fig. 3 (C.6), p_1 is hit and $r_6.T = 6$. From H in Fig. 3 (C.5) we know that p_1's valid position is 3 and $\xi = 1.5$, then we have $r_6.d = 3$ and $r_6.hit = 2$ since $r_6.d \geq \xi = 1.5$. Thus $r_6.bValid$ =T and we change p_1's valid position in H to 6. Finally, $\xi = \frac{r_3.d+r_4.d+r_6.d}{3} = 2$.

(7) process $r_7(p_3)$. As shown in Fig. 3 (C.7), p_3 is hit and $r_7.T = 7$. As the size of Q is 6, before r_7 is added to the tail of Q, r_1 is firstly removed from the head of Q. From H in Fig. 3 (C.6) we know that p_3's valid position is 5 and $\xi = 2$, then we have $r_7.d = 2$ and $r_7.hit = 1$ since $r_7.d \geq \xi = 2$. Thus $r_7.bValid$ =T and we change p_3's valid position in H to 7. Finally, as r_1 is not a hit request, we just need to add 1 and $r_7.d$ to the denominator and numerator of Formula 4 respectively, i.e. $\xi = \frac{r_3.d+r_4.d+r_6.d+r_7.d}{4} = 2$, otherwise, 1 and $r_1.d$ are firstly subtracted from the denominator and numerator of Formula 4, respectively.

As illustrated in this example, for each request, the computing of AHD can be done with time complexity of $O(1)$. In this example, we assume $\delta_C + \delta_D = 4$, thus ξ is directly computed by $\frac{\sum_{i=1}^{m} r_i.d}{m}$ according to Formula 6. In Algorithm 1, we use updateAHD(ξ) to denote the procedure of computing AHD, the pseudo-code is omitted for limited space.

The Algorithm. As shown in Algorithm 1, in the beginning stage before the buffer is full, i.e., $|L_C \cup L_D| < s \wedge |L_H| = 0$, if the request on p is a *miss-request* and $p's$ page id is not in L_H, Algorithm 1 will execute the code in Case III. Since $|L_C \cup L_D| < s$, we will call Procedure ReadIn() in line 16 to fetch p from the disk. After that updateAHD() is called in line 17. At last, δ_C or δ_D will increase by 1 by calling Procedure AdjustBottomProtionList(). If the current request on p is a *hit-request*, that is, $p \in L_C \cup L_D$, we will execute the code in Case I. Specifically, if $p \in L_{CB}$ (L_{DB}), it means that p should not stay anymore in L_{CB} (L_{DB}), since L_{CB} (L_{DB}) is used to maintain once-requested clean (dirty) pages. Then we move p to the MRU position of L_{CT} or L_{DT} and adjust the size of L_{CB} and L_{DB}, respectively.

If the buffer is full, for a *hit-request* corresponding to Case I and discussed already. If the current request is a *miss-request*, then we will check whether $p's$ id is contained in L_H. If $p's$ id is contained in L_H, which corresponds to Case II. In this case, we will firstly call evictPage() to select a victim page to make room for p. If $p.hit = 0$, it means that the size of L_{CB} or L_{DB} is too small, then δ_C or δ_D will increase by 1 (lines 10-11). After that, we will fetch p from disk by calling Procedure ReadIn(), then call updateAHD() in line 13, and adjust the length of L_{CB} and L_{DB} in line 14. If the current request is a *miss-request* and $p's$ id is not in L_H, which corresponds to Case III.

Based on Algorithm 1, we can effectively identify *once-frequently-requested* pages from pages that are requested multiple times to improve the overall performance. By using a hash table to maintain the pointers to each page in the buffer, the complexity serving each request is $O(1)$.

Analysis Adaptivity. The adaptivity means that our method continually revises the parameter δ_C and δ_D that are used to control the size of L_{CB} and L_{DB}. Moreover, the adaptivity means that if the workload is to change from one access pattern to another one or vice versa, we will track such change and adapt itself to exploit the new opportunity.

Scan-Resistant. When serving a long sequence of one-time-only requests, ABR will only evict pages in $L_{CB} \cup L_{DB}$. This is because, when requesting a new page p, i.e., $p \notin L_C \cup L_D \cup L_H$, p is always put at the MRU position of L_{CB} or L_{DB}. It will not impose any affect on pages in $L_{CT} \cup L_{DT}$ unless it is requested again before it is paged out from L_H. For this reason, we say ABR is scan-resistant. Furthermore, a buffer is usually used by several processes or threads concurrently, when a scan of a process or thread begins, less hits will be encountered in $L_{CB} \cup L_{DB}$ compared to $L_{CT} \cup L_{DT}$, and hence, according to Algorithm 1, the size of L_{CT} and L_{DT} will grow gradually, and the resistance of ABR to scans is strengthened again.

Loop-Resistant. We say that ABR is loop-resistant means that when the size of the loop is larger than the buffer size, ABR will keep partial pages of the loop sequence in the buffer, and hence, achieve higher performance. We explain this point from three aspects. (1) The loop requests only pages in L_C. In the first

Algorithm 1: ABR(page p, type T)

Case I: $p \in L_C \cup L_D$, a buffer hit has occurred.

```
1   if (p ∈ L_C) then {ι_c ← ι_c + 1; if (p ∈ L_CB) then {δ_C ← max{λ · s, δ_C − 1};}};
2   else {ι_d ← ι_d + 1; if (p ∈ L_DB) then {δ_D ← max{λ · s, δ_D − 1};}}
3   if (T = read ∧ p ∈ L_C) then {if (p.d ≥ ξ) then {move p to the MRU of L_CT;} }
4   else if (p.d ≥ ξ) then {move p to the MRU position of L_DT;}
5   else if (T = write ∧ p ∈ L_C) then {move p to the MRU position of L_DB;}
6   if (p.d ≥ ξ) then {p.hit ← p.hit + 1; p.bHasHit = TRUE;}
7   updateAHD(ξ);
8   AdjustBottomPortionList();
```

Case II: $p \in L_H$, a buffer miss has occurred.

```
9   evictPage();
10  if (T = read ∧ p.bHasHit = FALSE) then {δ_C ← min{|L_C|, δ_C + 1};}
11  if (T = write ∧ p.bHasHit = FALSE) then {δ_D ← min{|L_D|, δ_D + 1};}
12  ReadIn(p, T, true);
13  updateAHD(ξ);
14  AdjustBottomPortionList();
```

Case III: $p \notin L_C \cup L_D \cup L_H$, a buffer miss has occurred.

```
15  if (|L_C ∪ L_D| = s) then {evictPage();}
16  ReadIn(p, T, false);
17  updateAHD(ξ);
18  AdjustBottomPortionList();
```

Procedure evictPage()

```
1   β ← C_{L_C}/(C_{L_C} + C_{L_D});   /*β is computed based on the recent s/2 requests*/
```

Case I: $|L_C| < \beta \cdot s$ /* L_D is longer than expected*/.

```
2   ρ_wd ← ρ_wd + 1;
3   q ← FindEvictedPage(L_DB); write the content of q to disk;
4   if (|L_H| = s/2) then {delete the item in the LRU position of L_H;}
5   delete q from L_DB and insert its page id as a new item in the MRU position of L_H;
```

Case II: $|L_C| \geq \beta \cdot s$ /* L_C is longer than expected*/.

```
6   if (|L_H| = s/2) then {delete the item in the LRU position of L_H;}
7   q ← FindEvictedPage(L_CB); delete q and insert its page id to MRU of L_H;
```

Procedure ReadIn(page p, type T, bool $bTop$)

```
1   if (T = read) then increase ρ_rc and ι_c by 1
else increase ρ_rd and ι_d by 1
2   fetch p from the disk; p.hit ← 0; p.bHasHit = FALSE;
3   if (T = read ∧ bTop = TRUE) then insert it to the MRU of L_CT;
4   if (T = read ∧ bTop = FALSE) then insert it to the MRU of L_CB;
5   if (T = write ∧ bTop = TRUE) then insert it to the MRU of L_DT;
6   if (T = write ∧ bTop = FALSE) then insert it to the MRU of L_DB;
```

Function FindEvictedPage(list L)

```
1   q ← L.Tail;
2   while (q.hit > 0) do {q.hit ← q.hit/2; move q to MRU of L; q ← L.Tail;}
3   return q;
```

Procedure AdjustBottomPortionList()

```
1   if (|L_C ∪ L_D| = s) then
2       Move the MRU (or LRU) page of L_CB and L_DB (or L_CT and L_DT) to LRU
        (MRU) of L_CT and L_DT (or L_CB and L_DB) to make |L_CB| = δ_C ∧ |L_DB| = δ_D;
3   else {δ_C ← |L_CB|; δ_D ← |L_DB|;}
```

cycle of the loop request, all pages are fetched into the buffer and inserted at the MRU position of L_{CB} sequentially. Before each insertion, ABR will select a victim page q. If q is the LRU page of L_{DB}, then after the insertion of p in the MRU position of L_{CB}, ABR will adjust the size of L_{CB} and p will be adjusted to the LRU position of L_{CT}; otherwise p is still at the MRU position of L_{CB}. With the processing of the loop requests, more pages of the loop sequence will be moved to L_{CT} and these pages are thus kept in buffer, therefore the hit ratio will not be zero anymore. (2) The loop requests only pages in L_D. This is same to (1). (3) The loop contains pages in both L_C and L_D. In this case, obviously, dirty pages will stay in buffer longer than clean pages and the order of the pages eviction is not same as they entered into the buffer, and hence, ABR can process them elegantly to achieve higher hit ratio.

4 Experiments

4.1 Experimental Setup

Due to the implementation of FTL is device-related and supplied by the disk manufacturer, and there is no interface supplied for users to trace the number of write and read, we choose to use the simulator of [19] to count the numbers of read and write operations.

Methods for Comparison. We implemented LRU, CFLRU [2] and CFDC [6]. Further, we implemented ACR [7] and ABR. All these methods were implemented on the simulator using Visual C++ 6.0. For CFLRU, we set the "window size" of "clean-first region" to 75 % of the buffer size, for CFDC, the "window size" of "clean-first region" is 50 % of the buffer size, and the "cluster size" of CFDC is 64, the value of these parameters are suggested by the papers.

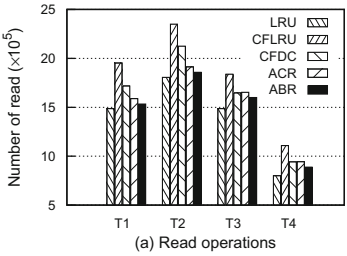
Traces for Experiment. We simulated a database file of 64MB, which corresponds to 32 K physical pages and each page is 2KB, the buffer size is 4 K pages. We generated four synthetical traces which satisfy Zipf distributions. The statistics of the four traces are shown in Table 2, where $x\%/y\%$ in column "Read/Write Ratio" means that for a certain trace, $x\%$ of total requests are about read operations and $y\%$ about write operations; while $x\%/y\%$ in column "Locality" means that for a certain trace, $x\%$ of total operations are performed in a certain $y\%$ of the total pages.

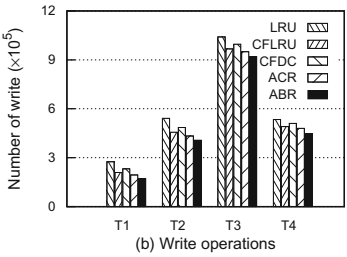
Storage Mediums. We select two flash chips for our experiment, Samsung MCAQE32G5APP and Samsung MCAQE32G8APP-0XA [20]. The ratio of the cost of random read to that of random write is 1:118 and 1:2, respectively. The reason for the huge discrepancy of the two flash disks lies in that the first flash disk is based on MLC NAND chip, while the second flash disk is based on SLC NAND chip. Both type of flash disks are already adopted as auxiliary storage in many applications.

Metrics. We choose the following metrics to evaluate the nine buffer replacement policies: (1) number of physical read operations; (2) number of physical write operations, which includes the write operations caused by the erase

Table 2. The statistics of the traces

Trace	Total requests	Read/Write ratio	Locality
T1	3,000,000	90 % / 10 %	60 % / 40 %
T2	3,000,000	80 % / 20 %	50 % / 50 %
T3	3,000,000	60 % / 40 %	60 % / 40 %
T4	3,000,000	80 % / 20 %	80 % / 20 %





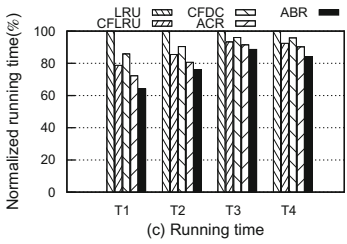
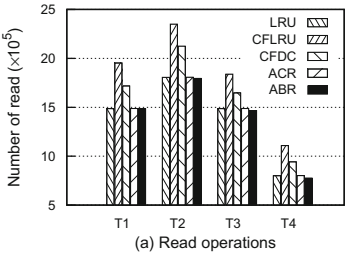
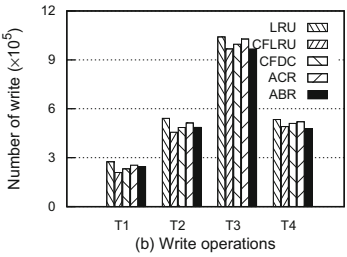


Fig. 4. The comparison of random read, random write and normalized running time on trace T1 to T4 for Samsung MCAQE32G5APP flash disk.





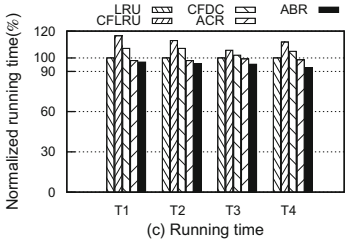


Fig. 5. The comparison of random read, random write and normalized running time on trace T1 to T4 for Samsung MCAQE32G8APP-0XA flash disk.

operations of flash disks; and (3) running time. Though there may exist some differences compared with the results tested on a real platform, they reflect the overall performance of different replacement policies by and large with neglectable tolerance.

4.2 Impacts of Large Asymmetry

Figure 4 (a) shows the comparison of the number of random read operations on trace T1 to T4 w.r.t. Samsung MCAQE32G5APP flash disk, from which we know that LRU has least read operations, the reason lies in that LRU does not differentiate read and write operations, thus it will not delay the paging out of dirty pages in the buffer. On the contrary, CFLRU firstly pages out clean pages, thus it needs to read in more pages than other methods.

From Fig. 4 (b) we know that LRU suffers from more write cost than other methods, and CFDC suffers from more write operations due to that it will page out all pages in a cluster before paging out pages in other clusters. Although CFLRU and ACR suffer from less write operations than LRU and CFDC, we can see that ABR consumes less write operations than them, this is because for the cost ratio of 1:118, (1) ABR often makes correct predictions, and (2) ABR keeps more dirty pages in the buffer than CFLRU and ACR.

Figure 4 (c) presents the results of normalized running time, from which we know that ABR works better than other methods. The reason lies in that the cost of write operation is much more expansive than that of read operation for Samsung MCAQE32G5APP flash disk and our eviction policy is based on cost of clean and dirty lists. Besides, we can see that by exploiting AHD and reference frequency, ABR works better than ACR.

Thus for flash disks with large asymmetry on read and write operations, by firstly paging out clean pages, flash-aware buffer replacement policies work better than LRU since they improve the overall performance by reducing the costly write operations significantly. Moreover, ABR works best, due to that it makes correct prediction and that frequently-requested dirty pages stay in buffer longer than once-requested and once-frequently-requested dirty pages.

4.3 Impacts of Small Asymmetry

Figure 5 (a) shows the comparison of the number of read operations w.r.t. Samsung MCAQE32G8APP-0XA flash disk, from which we know that CFLRU and CFDC consume much more read operations than other methods, the reason lies in that they firstly page out clean pages without considering the real cost of fetching clean pages from a flash disk into buffer. As a result, they suffer from large read cost. Although our methods keeps more dirty pages in buffer than clean pages since the cost of read operation is still cheaper than write operation, ABR achieves competing performance to LRU for read operation by improving the hit ratio of *frequently-requested* clean pages.

From Fig. 4 (b) we know that the number of write operations of ABR becomes larger than that in Fig. 4 (b), this is because the ratio of read and write becomes smaller than before, and our policy will pay more attention to clean pages. Though CFLRU and CFDC have less write operations than LRU, they waste many more read operations, which makes them achieving worse performance than LRU, as shown in Fig. 5 (c). Again, we can see that ABR works best by exploiting AHD and reference frequency.

Therefore, for flash disks with small asymmetry on read and write operations, ABR is better than LRU, CFLRU, CFDC and ACR, because ABR only consumes the same or less read operations than LRU, which is much less than that consumed by CFLRU and CFDC; though still need to consume more write operations than CFLRU and CFDC, the saved cost of read operation is far more than that wasted by write operations.

5 Conclusions

Considering that existing buffer replacement methods cannot process workload with temporal locality, we propose an adaptive cost-based replacement policy, namely ABR. ABR organizes buffer pages into clean list and dirty list, and the newly entered pages will not be inserted at the MRU position of either list, but at some position in middle. By exploiting average hit distance to control the movement of buffer pages, ABR is wise in identifying once-frequently-requested pages and the frequently-requested pages can stay in the buffer for a longer time. Besides, ABR considers reference count and is more adaptive than existing works to workloads of different access patterns, and thus, achieves better performance. The experimental results on different traces and flash disks show that ABR not only adaptively tunes itself to workloads of different access patterns, but also works well for different kind of flash disks compared with existing methods.

We plan to make further improvement on ABR by considering changing the write operations from random write to sequential write and implement ABR in a real platform to evaluate it with various real workloads for flash-based applications.

References

1. Lee, S.-W., Moon, B.: Design of flash-based DBMS: an in-page logging approach. In: SIGMOD, pp. 55–66 (2007)
2. Park, S.-Y., Jung, D., Kang, J.-U., Kim, J., Lee, J.: CFLRU: a replacement algorithm for flash memory. In: CASES, pp. 234–241 (2006)
3. Jo, H., Kang, J.-U., Park, S.-Y., Kim, J.-S., Lee, J.: FAB: flash-aware buffer management policy for portable media players. IEEE Trans. Consum. Electron. **52**(2), 485–493 (2006)
4. Kim, H., Ahn, S.: BPLRU: a buffer management scheme for improving random writes in flash storage. In: FAST, pp. 239–252 (2008)
5. Koltsidas, I., Viglas, S.: Flashing up the storage layer. PVLDB **1**(1), 514–525 (2008)
6. Yi, O., Harder, T., Jin, P.: CFDC: a flash-aware replacement policy for database buffer management. In: DaMoN, pp. 15–20 (2009)
7. Tang, X., Meng, X.: ACR: an adaptive cost-aware buffer replacement algorithm for flash storage devices. In: MDM, pp. 33–42 (2010)
8. Kim, B.-K., Lee, D.-H.: LSF: a new buffer replacement scheme for flash memory-based portable media players. IEEE Trans. Consum. Electron. **59**(1), 130–135 (2013)
9. Jin, R., Cho, H.-J., Chung, T.-S.: LS-LRU: a lazy-split LRU buffer replacement policy for flash-based B+-tree index. J. Inf. Sci. Eng. **31**(3), 1113–1132 (2015)

10. Jin, P., Yi, O., Härder, T., Li, Z.: AD-LRU: an efficient buffer replacement algorithm for flash-based databases. Data Knowl. Eng. **72**, 83–102 (2012)
11. On, S.T., Gao, S., He, B., Wu, M., Luo, Q., Xu, J.: FD-Buffer: a cost-based adaptive buffer replacement algorithm for flashmemory devices. IEEE Trans. Comput. **63**(9), 2288–2301 (2014)
12. Johnson, T., Shasha, D.: 2Q: a low overhead high performance buffer management replacement algorithm. In: VLDB, pp. 439–450 (1994)
13. Megiddo, N., Modha, D.S.: ARC: a self-tuning. low overhead replacement cache. In: FAST (2003)
14. Jiang, S., Zhang, X.: Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance. IEEE Trans. Comput. (TC) **54**(8), 939–952 (2005)
15. Babaoglu, O., Joy, W.N.: Converting a swap-based system to do paging in an architecture lacking page-reference bits. In: SOSP, pp. 78–86 (1981)
16. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The LRU-K page replacement algorithm for database disk buffering. In: SIGMOD, pp. 297–306 (1993)
17. John, T., Robinson, M.V.: Data cache management using frequency-based replacement. In: SIGMETRICS, Devarakonda, pp. 134–142 (1990)
18. Lee, D., Choi, J., Kim, J.-H., Noh, S.H., Min, S.L., Cho, Y., Kim, C.-S.: LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE Trans. Comput. (TC) **50**(12), 1352–1361 (2001)
19. Jin, P., Su, X., Li, Z.: A flexible simulation environment for flash-aware algorithms. In: CIKM, Lihua Yue, pp. 2093–2094 (2009)
20. http://www.datasheetcatalog.net