Eleventh International Conference on Mobile Data Management

ACR: an Adaptive Cost-Aware Buffer Replacement Algorithm for Flash Storage Devices

Xian Tang, Xiaofeng Meng

School of Information, Renmin University of China
{txianz,xfmeng2006}@gmail.com

***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 in the aspects of time, makes it necessary to reconsider the design of existing replacement algorithms to leverage their performance potential.**

Different from existing flash-aware buffer replacement policies that focus on the *asymmetry* of read and write operations, we address the "*discrepancy*" of the *asymmetry* for different flash disks, which is the fact that exists for a long time, while has drawn little attention by researchers since most existing flash-aware buffer replacement polices are somewhat based on the assumption that the cost of read operation is neglectable compared with that of write operation. In fact, this is not true for current flash disks on the market.

We propose an adaptive cost-aware replacement policy (ACR) that uses three *cost*-based heuristics to select the victim page, thus can fairly make trade off between clean pages (their content remain unchanged) and dirty pages (their content is modified), and hence, can work well for different type of flash disks of large *discrepancy*. Further, in ACR, buffer pages are divided into clean list and dirty list, the newly entered pages will not be inserted at the MRU position of either list, but at some position in the middle, thus the once-requested pages can be flushed out from the buffer quickly and the frequently-requested pages can stay in buffer for a longer time. Such mechanism makes ACR adaptive to workloads of different access patterns. The experimental results on different traces and flash disks show that ACR 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.

I. Introduction

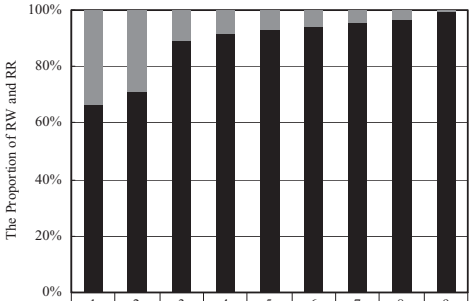
Though primarily designed for mobile devices due to its superiority such as low access latency, low energy consumption, light weight and shock resistance, flash-based storage devices have been steadily expanded into personal computer and enterprise server markets with ever increasing capacity of their storage and dropping of their price. In the past several years, the density of NAND flash memory increased twofold and this trend will continue until year 2012 [1]. Existing operating systems are already providing facilities to take advantage of flash disks (e.g., Solid State Drive) [2].

Typically, a flash disk managed by an operating system is a block device which provides the same interface type as a magnetic disk, however, their I/O characteristics are widely disparate. 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 [3], which makes it necessary to reconsider the design of IO-intensive and performance-critical software to achieve maximized performance.

Buffer is one of the most fundamental component in modern computing. It is widely used in storage systems, databases, web servers, file systems, operating systems, etc. Any substantial progress in buffer replacement algorithms will affect the entire modern computational stack. Assuming that the secondary storage consists of magnetic disks and there is no difference for the time delay between read and write operations, the goal of existing buffer replacement policies [4]–[10] is to minimize the buffer miss ratio for a given buffer size. When the buffer is full and the current requested page is not in the buffer, the replacement policy has to select an in-buffer page as the victim, if the victim is a dirty page, it will be written back to disk before paging in the requested page so as to guarantee data consistency, which may be a performance bottleneck since the process or thread requesting for the requested page must wait until write completion. Early in two decades ago, [11] has realized the fact that whether a page is read only or modified is an important factor which will affect the performance of a replacement policy and should be considered in the replacement decision. As flash disks are becoming an important alternative to magnetic disks, this phenomena should be paid more attention than ever.

Considering the asymmetric read and write operation of flash disks, researchers have proposed flash-aware replacement algorithms [12]–[16] in the past yeas. Based on the assumption that the cost of random read operation is *neglectable* compared with that of random write operation, the fundamental idea behind these policies is reducing random write operations by firstly paging out clean pages arbitrarily no matter how frequently they are requested, which means that the cost of random write operations dominates the overall cost of a replacement policy. However, from Fig. 1, we can get an important observation that is not consistent with the above assumption: the cost of random read operation should *not* be neglected for all cases, since the time consumed by random write and read operation for different type of flash devices various largely. Though all flash devices demonstrate fast random read speeds and slow random write speeds, it is not difficult to see that paging out clean pages before dirty pages without considering their reference frequency is not reasonable for all cases.

978-0-7695-4048-1/10 $26.00 © 2010 IEEE
DOI 10.1109/MDM.2010.34



	1	2	3	4	5	6	7	8	9
RR	1	1	1	1	1	1	1	1	1
RW	2	2.5	8	11	13	17	20	30	118

Fig. 1: The normalized proportion of time consumed by random write (RW) and random read (RR) operations for NAND flash disks. The numbers on the X axis represent 9 flash disks, "1" is Samsung MCAQE32G8APP-0XA, "2" is Samsung K9WAG08U1A, "3" is Samsung K9XXG08UXM, "4" is Samsung K9F1208R0B, "5" is Samsung K9GAG08B0M, "6" is Hynix HY27SA1G1M, "7" is Samsung K9K1208U0A, "8" is Samsung K9F2808Q0B, "9" is Samsung MCAQE32G5APP [17].

Moreover, since the cost of write operations is more expensive than that of read operations, reducing write operations in many cases will improve the overall performance. However, for a sequence of write requests, the write operations cannot be further reduced by keeping *once-requested* dirty pages in the buffer for a long time, but by keeping *frequently-requested* dirty pages from being paged out too early. Existing flash-aware replacement algorithms do not differentiate between frequently-requested dirty pages and once-requested dirty pages, which makes them, though delay the time of evicting a dirty page by paging out clean pages firstly, fail to make further improvement on the hit ratio of *frequently-requested* dirty pages when *once-requested* dirty pages occupying too much space, such that previously *frequently-requested* dirty pages may be paged out before some *once-requested* dirty pages because of their different recency.

Further, [9] pointed out that "real-life workloads do not admit a one-size-fits-all characterization. They may contain long sequential request or moving hot spots. The frequency and scale of temporal locality may also change with time. They may fluctuate between stable, repeating access patterns and access patterns with transient clustered references. No static, a priori fixed replacement policy will work well over such access patterns".

Different from the previous buffer replacement policies that focus on either the various access patterns with uniform access cost, or the asymmetry of access cost of flash, in this paper, we further address the impact imposed by the *discrepancy* of the ratio of write cost to read cost on different flash disks. This motivates us to design an adaptive cost-based buffer replacement policy that possesses three features: (1) low overall I/O cost of serving all requests based on flash disks of different ratios of write cost to read cost, (2) constant-time complexity per request, (3) adaptive to dynamically evolving workloads.

We propose a new buffer replacement policy, namely, Adaptive Cost-aware buffer Replacement (ACR). First, ACR uses three cost-based heuristics to select the victim page, thus can fairly make trade off between clean pages and dirty pages, and hence, can work well for different kind of flash disks with large discrepancy of the ratio between read and write operations. Second, ACR organizes buffer pages into clean list and dirty list, the newly entered pages are not inserted at the MRU position of either list, but at some position in the middle. As a result, the once-requested pages can be flushed out from the buffer quickly and the frequently-requested pages can stay in buffer for a longer time. This mechanism makes ACR adaptive to workloads of different access patterns and can really improve the hit ratio of frequently-requested pages so as to improve the overall performance.

Moreover, ACR maintain a buffer directory, namely, ghost buffer, to remember recently evicted "*once-requested*" buffer pages. The hits on the ghost LRU lists are used to adaptively determine the length of the buffer list and identify more frequently-requested pages, such that ACR can adaptively decide how many pages each list should maintain in response to an evolving workload.

The remainder of this paper is organized as follows. Section II introduces background knowledge about flash disks and existing buffer replacement polices. Section III introduces our ACR algorithm and the experimental results are presented in Section IV. We conclude our work in Section V.

II. Background and Related Work

In this section, we firstly review the most important hardware characteristics of flash disks, then give a detailed discussion of existing replacement policies, which motivates us to devise the new replacement policy.

A. Flash Memory

Generally speaking, there are two different types of flash memories: NOR and NAND flash memories[1]. Compared with NAND flash memory, NOR flash memory has separate address and data buses like EPROM and static random access memory (SRAM) while NAND flash memory has an I/O interface which control inputs and outputs. NOR flash memory was developed to replace programmable read-only memory (PROM) and erasable PROM (EPROM) for efficient random access while NAND flash memory was developed for data storage because of its higher density. Flash disks usually consist of NAND flash chips.

There are three basic operations on NAND flash memories: read, write, and erase. Read and write operations are performed in units of a page. Erase operations are performed

[1]http://www.dataio.com/pdf/NAND/MSystems/MSystems_NOR_vs_NAND.pdf

in units 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. Moreover, Each block of flash memory may worn out after the specified number of write/erase operations. To avoid the premature worn out of blocks caused by highly localized writes, it is necessary to distribute erase operations evenly over all blocks.

To overcome the physical limitation of flash memory, flash disks employ an intermediate software layer called Flash Translation Layer (FTL), which is typically stored in a ROM chip, to emulate the functionality of block device and hide the latency of erase operation as much as possible. One of the key roles of FTL is to redirect a write request on a page to an empty area erased previously. Therefore, FTL needs to maintain an internal mapping table to record the mapping information from the logical address number to the physical location. This internal mapping table is maintained in volatile memory. The reconstruction of the mapping table is at startup or in case of a failure. The details of the implementation of FTL are device-related and supplied by the manufacturer, which are transparent to users.

Compared with magnetic disks, although NAND flash memories have various advantages such as small and lightweight form factor, solid-state reliability, no mechanical latency, low power consumption, and shock resistance [18], they also possess inherent limitations, say asymmetric operation latencies, and the degree of the asymmetry various largely from one to another. Specifically, a flash memory has asymmetric read and write operation characteristics in terms of performance and energy consumption. It usually demonstrates extremely fast random read speeds, but slow random write speeds. Moreover, as shown in Fig. 1, the ratio of the cost of write and read operation for different flash disks various largely. Therefore when designing flash-aware buffer replacement policy, not only the asymmetry of read and write should be considered, but also the discrepancy of the asymmetries of different flash disks should be paid more attention.

B. Buffer Replacement Policies

Consider the typical scenario where a system consists of two memory levels: main (or buffer) and auxiliary. Buffer is significantly faster than the auxiliary memory and both memories are managed in units of equal sized pages.

Assuming that the secondary storage consists of magnetic disks and the costs of all eviction operations are equal to each other, the goal of existing buffer replacement policies is to minimize the buffer miss ratio for a given buffer size. The miss ratio reflects the fraction of pages that must be paged into the buffer from the auxiliary memory. For example, recent studies on replacement algorithms such as 2Q [7], ARC [9], LIRS [8], CLOCK [4], LRU-K [6], FBR [5] and LRFU [10] mainly aim to improve the traditional LRU heuristic, which consider page recency or balance both recency and frequency to reduce miss rate. However, the above assumption is not hold anymore when applied to flash disks because of the asymmetric access

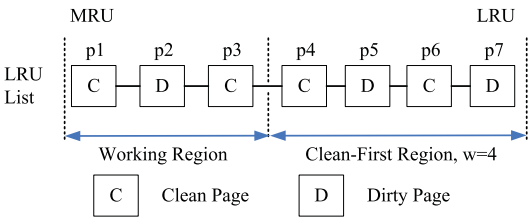


Fig. 2: The CFLRU Replacement Policy

times. This adds another dimension to the management of flash disk based buffer.

The replacement problem for buffers with non-uniform access time can be modeled by the weighted buffering problem. The goal is to minimize the total cost to serve the request sequence. [19] proposed an optimal *off-line* algorithm for this problem in $O(sn^2)$ time by reducing it to the minimal cost maximum flow problem [20], where s is the buffer size and n is the number of total requests. Unfortunately, this optimal algorithm is resource intensive in terms of both space and time, even though it knows all prior knowledge of the complete request sequence.

For an *online* algorithm, any knowledge about the future requests is unknown in advance. Recently, researchers have proposed many online flash-aware buffer replacement policies.

The flash aware buffer policy (FAB) [13] maintains a block-level LRU list, of which pages of the same erasable block are grouped together. When a hit occurs on a page, the group containing the page is moved to the beginning of the LRU list. When a miss occurs, the group that has the largest number of pages will be selected as victim and all dirty pages in this group will be paged out. FAB is mainly used in portable media player applications where most write requests are sequential.

BPLRU [14] also maintains an block-level LRU list. Different from FAB, BPLRU [14] 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) [12] 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. Fig. 2 illustrates the idea of CFLRU. 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. Only when there is no clean page in the clean-first region, the dirty page at the LRU position of the clean-first region is selected as a victim. The size of the clean-first region is controlled by a parameter w called the window size. Compared with LRU, CFLRU reduces the write operations significantly.

TABLE I: Summary of notations

Notation	Description
L_C	the LRU list containing clean pages
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 LRU list containing dirty pages
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_{CH}	the LRU list containing page id of once-requested clean pages
L_{DH}	the LRU list containing page id of once-requested dirty 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
M_{L_D}	the number of physical operations on pages in L_D
M_{L_C}	the number of physical operations on pages in L_C
R_{L_D}	the number of logical operations on pages in L_D
R_{L_C}	the number of logical operations on pages in L_C

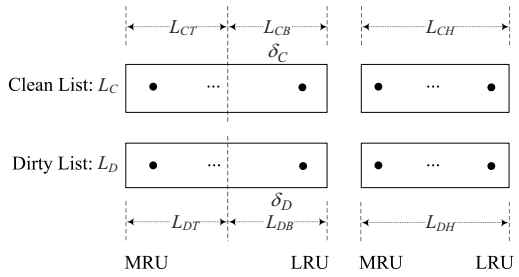


Fig. 3: The ACR Replacement Policy

Based on the same idea, [15] makes improvements over CFLRU by organizing clean pages and dirty pages into different LRU lists to achieve constant complexity per request. Further, CFDC [16] improves the system performance by clustering together dirty pages whose page numbers are close to each other, thus can improve the efficiency of write operations. In CFDC, a cluster has variable size determined by the set of pages currently kept, which is different from block-level LRU list.

III. The ACR Policy

A. Data Structures

As shown in Fig. 3, ACR splits the LRU list into two LRU lists, say L_C and L_D. L_C is used to keep *clean* pages and L_D is used to keep *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 is divided into L_{CT} and L_{CB}, and $L_{CT} \wedge L_{CB} = \emptyset$, L_{CT} contains *frequently-requested clean* pages while L_{CB} contains *once-requested clean* pages and frequently-requested clean pages that are *not* referenced for a long time. Similarly, L_D is also divided into L_{DT} and L_{DB}, and $L_{DT} \wedge L_{DB} = \emptyset$. L_{DT} contains *frequently-requested dirty* pages while L_{CB} contains *once-requested dirty* pages and frequently-requested dirty pages that are *not* referenced for a long time. The sizes of L_{CB} and L_{DB} will dynamically change with the change of access patterns, which are controlled by δ_C and δ_D, respectively. Besides the pages that are in the buffer, we use a ghost buffer to trace the past references by recording the page id of those pages that are paged out from L_C or L_D. The ghost buffer is also divided into two LRU lists, say, L_{CH} and L_{DH}, which are used to keep the past references of clean and dirty pages, respectively. All pages in L_{CH} and L_{DH} are those that are *never* being requested again since they were paged into the buffer last time, that is, they are the *once-requested* pages. Fixing this parameter is potentially a tuning question, in our experiment, $|L_{CH} \cup L_{DH}| = s/2$. The notions used in this paper are shown in Table I.

B. Cost-based Eviction

If the buffer is full and the currently requested page p is in the buffer, then it is served without access the auxiliary storage, otherwise, ACR 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 replacement cost caused by the pages in L_C (L_D). The basic idea behind our replacement policy is that the length of L_C (L_D) should be proportional to the ration of the replacement cost of the pages in L_C (L_D) to that of all buffer pages according to recent m requests, in our experiment, m equals to half the buffer size, i.e., $m = s/2$. 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 $|L_C| < \beta \cdot s$, which means that L_D is too long, then the LRU page in L_D should be paged out, otherwise L_C is too long and the LRU page of L_C should be paged out, the "s" in this inequation is the buffer size in pages.

In the following discussion, we call the read and write operations that are served in buffer are logical hereafter, while ones that reach the disk are referred to as physical. The cost of reading a page from the flash disk is C_r, while the cost of writing a page to a random position in a flash disk is C_w. We present a family of methods to compute the values of C_{L_C} and C_{L_D} to decide the optimal scheme.

1) Conservative Scheme: Let M_{L_C} be the number of physical operations on pages in L_C and M_{L_D} the number of physical operations on pages in L_D. The first scheme used for computing C_{L_C} and C_{L_D}, which we refer to as conservative, is given by Formula 2 and Formula 3. Upon eviction, C_{L_C} and C_{L_D} are examined to compute the value of β.

$$C_{L_C} = \begin{cases} C_r, & M_{L_C} = 0 \\ M_{L_C} \cdot C_r, & M_{L_C} \neq 0 \end{cases} \tag{2}$$

$$C_{L_D} = \begin{cases} C_w, & M_{L_D} = 0 \\ M_{L_D} \cdot (C_w + C_r), & M_{L_D} \neq 0 \end{cases} \tag{3}$$

Note that before L_C or L_D seeing the first physical operation, C_{L_C} and C_{L_D} are assigned with C_r and C_w, respectively.

The conservativity of Formula 2 and Formula 3 lies in that they take into account only physical operations on pages, not logical ones. Therefore the conservative scheme does not try to induce the access pattern from the logical operation. Rather, it waits until the logical operation has been translated into physical accesses.

2) Optimistic Scheme: Though physical operations capture the actual cost paid by L_C and L_D, their sequences are dictated by logical operations. Moreover, while the pages remain in the buffer, many logical operations may occur between two consecutive physical operations. Formula 2 and Formula 3 will only record physical operations on these pages, and thus, if the workload changes, L_C and L_D will take many physical operations before conservative adapts. This motives us to design an "*optimistic*" version of the eviction scheme that works only on logical operations and thus, can adapt to new workloads as quickly as possible; the optimistic scheme is given by Formula 4 and Formula 5, where R_{L_C} refers to the number of logical operations on the pages of L_C while R_{L_D} the number of logical operations on the pages of L_D. R_{L_C} (R_{L_D}) is incremented by 1 when a logical read (write) occurs on a page in L_C (L_D). The two counters hold the total logical read and write operations on L_C and L_D, respectively. Upon eviction, our method will compute the total cost L_C and L_D would pay if these operations were physical.

$$C_{L_C} = R_{L_C} \cdot C_r \tag{4}$$

$$C_{L_D} = R_{L_D} \cdot (C_w + C_r) \tag{5}$$

The optimistic scheme is not conservative in the number of evicted pages. It assumes that when the workload changes from read-intensive to write-intensive (or vice-versa), the selection of a victim page should be changed from L_D to L_C (or vice-versa). Thus, optimistic adapts quickly to changing workloads. However, if the changes of the access pattern do not last long enough for the eviction cost to be expensed, the overall cost paid by the system grows.

Notice that the optimistic scheme tries to minimize the cost of future physical operations on L_C and L_D based on its history of logical operations. Consider the case that before a new eviction, L_C having been logically read a large number of times, then L_C upon eviction is found to be strongly read-intensive and the selection of victim page is very probably from L_D, that is, the LRU page p of L_D will be paged out. After that, if there is a write request on p, an expensive write cost is already paid by L_D. In such a case, not only the benefit from the cost-based eviction never realized, but also the system performance degenerates.

3) Hybrid Scheme: Logical and physical operations are two different operations, however, they all have impact on the overall performance. Both the conservative and optimistic scheme introduced above choose to consider only one of them, therefore may not really work in some cases. To minimize the total cost of physical operations, we introduce a hybrid scheme that takes both physical and logical operations into account by combining the strong points of the conservative and optimistic scheme, while avoid their weak points.

Assume that n is the number of pages in a file and s the number of pages allocated to the file in the buffer. Therefore 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 $(1 - s/n)$. In our hybrid scheme, the probability is used to compute the overall impact of logical operations on L_C and L_D, as shown by Formula 6 and 7. Upon eviction, our method will compute the total cost of L_C and L_D by considering the impacts of both logical and physical operations.

$$C_{L_C} = (R_{L_C} \cdot (1 - s/n) + M_{L_C}) \cdot C_r \tag{6}$$

$$C_{L_D} = (R_{L_D} \cdot (1 - s/n) + M_{L_D}) \cdot (C_w + C_r) \tag{7}$$

When selecting a victim page, the logical operations allow our hybrid scheme to recognize changes in the access pattern very quickly like the optimistic scheme. Moreover, it is also not so eager as the optimistic scheme to page out the expensive dirty page by considering the actually happened physical operations. By taking into account the cost of actually happened physical operations, the hybrid scheme has a realistic view of the impact the logical operations imposed on the buffer.

C. The ACR Replacement Policy

We now introduce the whole ACR replacement policy that adapts and tunes the length of L_C and L_D in response to an observed workload. Before running, $\delta_C = \delta_D = 0$. For easier discussion, we call a request on a page that is not in the buffer a *miss-request*, otherwise a *hit-request*.

As show in Algorithm 1, in the beginning stage before the buffer is full, i.e., $|L_C \cup L_D| < s \wedge |L_{CH} \cup L_{DH}| = 0$, if the request on p is a *miss-request* and $p's$ page id is not in $L_{CH} \cup L_{DH}$, Algorithm 1 will execute the code in Case III. Since $|L_C \cup L_D| < s$, ACR increases the logical and physical counters according to the operation type, then fetch p into the buffer and insert it to the MRU position of L_{CB} or L_{DB} according to the value of T. At last, δ_C or δ_D will increase by 1 by calling the procedure AdjustBottomProtionList(). If the current request on p is a *hit-request*, that is, $p \in L_C \cup L_D$, ACR 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 and frequently-requested clean (dirty) pages that are not requested yet for a long time. Then ACR will 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 (line 1-8 of Algorithm 1), the process is already discussed in the above paragraph. If the current request is a *miss-request*, then ACR will check whether $p's$ id is contained in $L_{CH} \cup L_{DH}$ or not. If $p's$ id is contained in $L_{CH} \cup L_{DH}$, which corresponds to Case II, it means that p has not been request after it entered into $L_{CB} \cup L_{DB}$. In this case, ACR will firstly

Algorithm 1: ACR(page p, type T) /* ACR is triggered on each request on a page p, T denotes the type of operation on p, T can be either "$read$" or "$write$"*/

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

1 **if** $(p \in L_C)$ **then** $\{R_{L_C} \leftarrow R_{L_C} + 1$; **if** $(p \in L_{CB})$ **then** $\{\delta_C \leftarrow max\{0, \delta_C - 1\};\}\}$;
2 **else** $\{R_{L_D} \leftarrow R_{L_D} + 1$; **if** $(p \in L_{DB})$ **then** $\{\delta_D \leftarrow max\{0, \delta_D - 1\};\}\}$
3 **if** $(T = read \wedge p \in L_C)$ **then** {move p to the MRU position of L_{CT};}
4 **else if** $(p \in L_D)$ **then** {move p to the MRU position of L_{DT};}
5 **else** {move p to the MRU position of L_{DB};}
6 **if** (p is moved from L_C to L_D) **then** $\{p.hit \leftarrow 0;\}$
7 **else** $\{p.hit \leftarrow p.hit + 1;\}$ /*$p.hit$ is the number of hit occurred on p since it entered into L_C or L_D*/
8 AdjustBottomPortionList();

Case II: $p \in L_{CH} \cup L_{DH}$, a buffer miss has occurred.

9 evictPage(); $p.hit \leftarrow 0$;
10 **if** $(p \in L_{CH})$ **then** $\{\delta_C \leftarrow min\{|L_C|, \delta_C + 1\};\}$
11 **else** $\{\delta_D \leftarrow min\{|L_D|, \delta_D + 1\};\}$
12 **if** $(T = read)$ **then** {fetch p from the disk; insert it to the MRU of L_{CT}; $M_{L_C} \leftarrow M_{L_C} + 1$; $R_{L_C} \leftarrow R_{L_C} + 1$;}
13 **else** {fetch p from the disk; insert it to the MRU of L_{DT}; $R_{L_D} \leftarrow R_{L_D} + 1$;}
14 AdjustBottomPortionList();

Case III: $p \notin L_C \cup L_D \cup L_{CH} \cup L_{DH}$, a buffer miss has occurred.

15 $p.hit \leftarrow 0$;
16 **if** $(|L_C \cup L_D| = s)$ **then** {evictPage();}
17 **if** $(T = read)$ **then** {fetch p from the disk; insert it to the MRU of L_{CB};$R_{L_C} \leftarrow R_{L_C} + 1$; $M_{L_C} \leftarrow M_{L_C} + 1$;}
18 **else** {fetch p from the disk; insert it to the MRU of L_{DB};$R_{L_D} \leftarrow R_{L_D} + 1$;}
19 AdjustBottomPortionList();

Procedure evictPage()

1 $\beta \leftarrow C_{L_C}/(C_{L_C} + C_{L_D})$; /*$\beta$ is computed based on the recent $s/2$ requests*/

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

2 $M_{L_D} \leftarrow M_{L_D} + 1$;
3 let q be the page in the LRU position of L_{DB} and $q.hit$ the number of hit occurred on q since it entered into L_D;
4 **if** $(q.hit > 0)$ **then** {write $q's$ content to disk; delete q; return;}
5 **if** $(|L_{CH} \cup L_{DH}| = s/2)$ **then** {delete the item in the LRU position of L_{DH};}
6 delete q and insert the page id of q as a new item in the MRU position of L_{DH};

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

7 let q be the page in the LRU position of L_{CB} and $q.hit$ the number of hit occurred on q since it entered into L_C;
8 **if** $(q.hit > 0)$ **then** {delete q; return;}
9 **if** $(|L_{CH} \cup L_{DH}| = s/2)$ **then** {delete the item in the LRU position of L_{CH};}
10 delete q and insert the page id of q as a new item in the MRU position of L_{CH};

Procedure AdjustBottomPortionList()

1 **if** $(|L_C \cup L_D| = s)$ **then**
2 Move the MRU (or LRU) page of L_{CB} and L_{DB} (or L_{CT} and L_{DT}) to LRU (MRU) position of L_{CT} and L_{DT} (or L_{CB} and L_{DB}) to make $|L_{CB}| = \delta_C \wedge |L_{DB}| = \delta_D$;
3 **else** $\{\delta_C \leftarrow |L_{CB}|; \delta_D \leftarrow |L_{DB}|;\}$

call evictPage() to select a victim page and page it out to make room for p, then δ_C or δ_D will increase by 1 since the size of L_{CB} or L_{DB} is too small. After that, ACR will fetch p from disk and insert it to the MRU position of L_{CT} or L_{DT}. At last, ACR will adjust the length of L_{CB} and L_{DB}. If the current request is a *miss-request* and $p's$ id is not in $L_{CH} \cup L_{DH}$, this case corresponds to Case III. Compared with the case that buffer is not full, ACR will firstly evict a page in this case.

Note that in ACR, pages that are served only once in the whole processing will be inserted at the MRU position of L_{CB} or L_{DB}, thus will be paged out earlier than those served more than once. Moreover, the pages in L_{CT} and L_{DT} can further utilize the space of L_{CB} and L_{DB} to make them staying longer in the buffer, such that the hit ratio of frequently-requested pages can be actually improved, especially for dirty pages. By using a hash table to maintain the pointers to each page in the buffer, the complexity of ACR for each page request is $O(1)$ and is only greater than the complexity of LRU by some constant c.

1) Adaptivity: The adaptivity of ACR lies in two aspects: (1) ACR continually revises the parameter δ_C and δ_D that are used to control the size of L_{CB} and L_{DB}. The fundamental intuition behind is: if there is a hit on page p of $L_{CB}(L_{DB})$ that mainly contains *once-requested* pages, then p becomes a frequently-requested page from now on and should be placed in $L_{CT}(L_{DT})$, and we should increase the size of $L_{CT}(L_{DT})$. Similarly, if $p's$ page id is in $L_{CH}(L_{DH})$ that records the historical reference information of once-requested pages, then we should increase the size of $L_{CB}(L_{DB})$. Hence, on a hit in $L_{CB}(L_{DB})$, we decrease $\delta_C(\delta_D)$, and on a hit in $L_{CH}(L_{DH})$, we increase $\delta_C(\delta_D)$. If the workload is to change from one access pattern to another one or vice versa, ACR will track such change and adapt itself to exploit the new opportunity. (2) ACR will choose a page for replacement according to the accumulative replacement costs of L_C and L_D, which gives a fair chance to clean and dirty pages for competition. Together, the two aspects of adaptivity makes ACR very wise in exploiting the asymmetry of flash IO and the new opportunity of various access pattern.

2) Scan-Resistant: When serving a long sequence of one-time-only requests, ACR will only evict pages in $L_{CB} \cup L_{DB}$ and it never evicts pages in $L_{CT} \cup L_{DT}$. This is because, when requesting on a totally new page p, i.e., $p \notin L_C \cup L_D \cup L_{CH} \cup L_{DH}$, 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_{CB} or L_{DB}. For this reason, we say ACR 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 ACR to scans is strengthened again.

3) Loop-Resistant: A loop requests is a sequence of pages that are served in a special order repeatedly. We say that ACR is loop-resistant means that when the size of the loop is larger than the buffer size, ACR will keep partial pages of the loop sequence in the buffer, and hence, achieve higher performance. We explain this point from three aspects in the case that the size of a loop is larger than the buffer size.

(1) the loop requests only pages in L_C. In the first 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, ACR 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}, ACR 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 in the buffer, and hence, ACR can process them elegantly to achieve higher hit ratio.

IV. Experiments

A. Experimental Setup

The goal of our experiment is to verify the effectiveness of ACR for flash disks of different characteristics on read and write operations. For a flash disk, the performance of a buffer replacement algorithm is affected by the number of physical read and write. However, 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. Therefore, we choose to use a simulator [21] to count the numbers of read and write operations. We implemented three existing state-of-the-art replacement policies for comparison, i.e., LRU, CFLRU [12] and CFDC [16]. We implemented three versions of ACR based on the three heuristics (Conservative, Optimistic and Hybrid), which are denoted as ACR-C, ACR-O and ACR-H, respectively. All 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.

We simulated a database file of 64MB, which corresponds to 32K physical pages and each page is 2KB, the buffer size ranges from 2K pages to 8K pages.

We have generated 4 types of synthetical traces which will access all pages randomly. The statistics of the four traces are shown in Table II, 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.

We select two flash disks for our experiment, the first is Samsung MCAQE32G5APP, the second is Samsung MCAQE32G8APP-0XA [17]. The ratio of the cost of random

TABLE II: The statistics of the traces used in our experiment

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%

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. In our experiment, the simulator assume that the page size is 2KB, and each block contains 64 pages.

We choose the following metrics to evaluate the six buffer replacement policies: (1) number of physical read operations, (2) number of physical write operations, and (3) running time. The running time is computed by adding up the cost of read and write operations, 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. We do not use hit ratio as a metric since it cannot really reflect the overall performance. The results of the second metrics in our experiment include the write operations caused by the erase operations of flash disks.

Note that if running on a real flash disk, CFDC may achieve better performance, this is because CFDC can make many random write to sequential write. On the contrary, CFLRU suffers from high CPU cost, which is also not reflected in our results.

B. Experimental Results and Analysis

1) Impact of large discrepancy on read and write operation: Fig. 4 shows the results of random read, random write and normalized running time on trace T1 to T4 for Samsung MCAQE32G5APP flash disk. Fig. 4 (a), (d), (g) and (j) are the results of the number of random read operations on trace T1 to T4, 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 LRU, CFDC, ACR-C, ACR-O and ACR-H. We can see from Fig. 4 (b), (e), (h) and (k) that LRU consumes more write cost than all other methods, and among the five flash-based methods, i.e., CFLRU, CFDC, ACR-C, ACR-O and ACR-H, CFDC and ACR-O suffer from more write operations, this is because, CFDC will page out all pages in a cluster before paging out pages in other clusters, and ACR-O often makes wrong predictions for the four traces when the cost ratio is 1:118. Although CFLRU suffers from less write operations than LRU, CFDC and ACR-O, we can see that ACR-C and ACR-H consume less write operations than CFLRU, this is because for the cost ratio of 1:118, (1) ACR-C and ACR-H often make correct predictions, (2) ACR-C and ACR-H maintain more dirty pages in the buffer than CFLRU. Fig. 4 (c), (f), (i) and (l) present the results of normalized running time, from which we know that ACR-C and ACR-H achieve higher performance than LRU, CFLRU, CFDC and ACR-O. The reason lies in that the cost of write operation is much more expansive than that of read operation for Samsung MCAQE32G5APP flash disk.

Thus for flash disks with large discrepancy on read and write operations, by firstly paging out clean pages, CFLRU, CFDC, ACR-C, ACR-O and ACR-H are better than LRU since they improve the overall performance by reducing the costly write operations significantly. Moreover, ACR-C and ACR-H are better than CFLRU and CFDC, because they make correct prediction and frequently-requested dirty pages stay in buffer longer than the once-requested dirty pages, thus can further reduce the cost of write operations.

2) Impact of small discrepancy on read and write operation: Fig. 5 just shows the results for trace T1 and T2 running on Samsung MCAQE32G8APP-0XA flash disk for limited space. Fig. 5 (a) and (d) are the results of the number of random read operations on trace T1 and T2, from which we know that LRU, ACR-C, ACR-O and ACR-H have 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. Although our ACR policy keeps more dirty pages in buffer than clean pages since the cost of read operation is still cheaper than that of write operation, ACR achieves competing performance to LRU for read operation by improving the hit ratio of frequently requested clean pages. CFLRU and CFDC firstly page out clean pages, thus they need to read in many more pages than LRU and ACR. As a result, they suffer from large read cost. We can see from Fig. 4 (b) and (e) that the number of write operations of ACR-C, ACR-O and ACR-H becomes larger than that in Fig. 4, 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, ACR-C, ACR-O and ACR-H for flash disks of small discrepancy on read and write operation, as shown in Fig. 5 (c) and (f).

Therefore, for flash disks with small discrepancy on read and write operations, ACR-C, ACR-O and ACR-H are better than LRU, CFLRU and CFDC, because ACR-C, ACR-O and ACR-H only consume 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 much more than that wasted by write operations.

3) Impact of different heuristics: By comparing Fig. 4 and Fig. 5, we can see that for trace T1 to T4, ACR-O is not efficient as ACR-C and ACR-H for flash disks of large discrepancy of read and write operation, but is better than ACR-C and ACR-H for flash disks of small discrepancy. For flash disks of large discrepancy, LRU is very inefficient, both CFLUR and CFDC can work better than LRU, but for flash disks of small discrepancy, LRU is more efficient than CFLRU

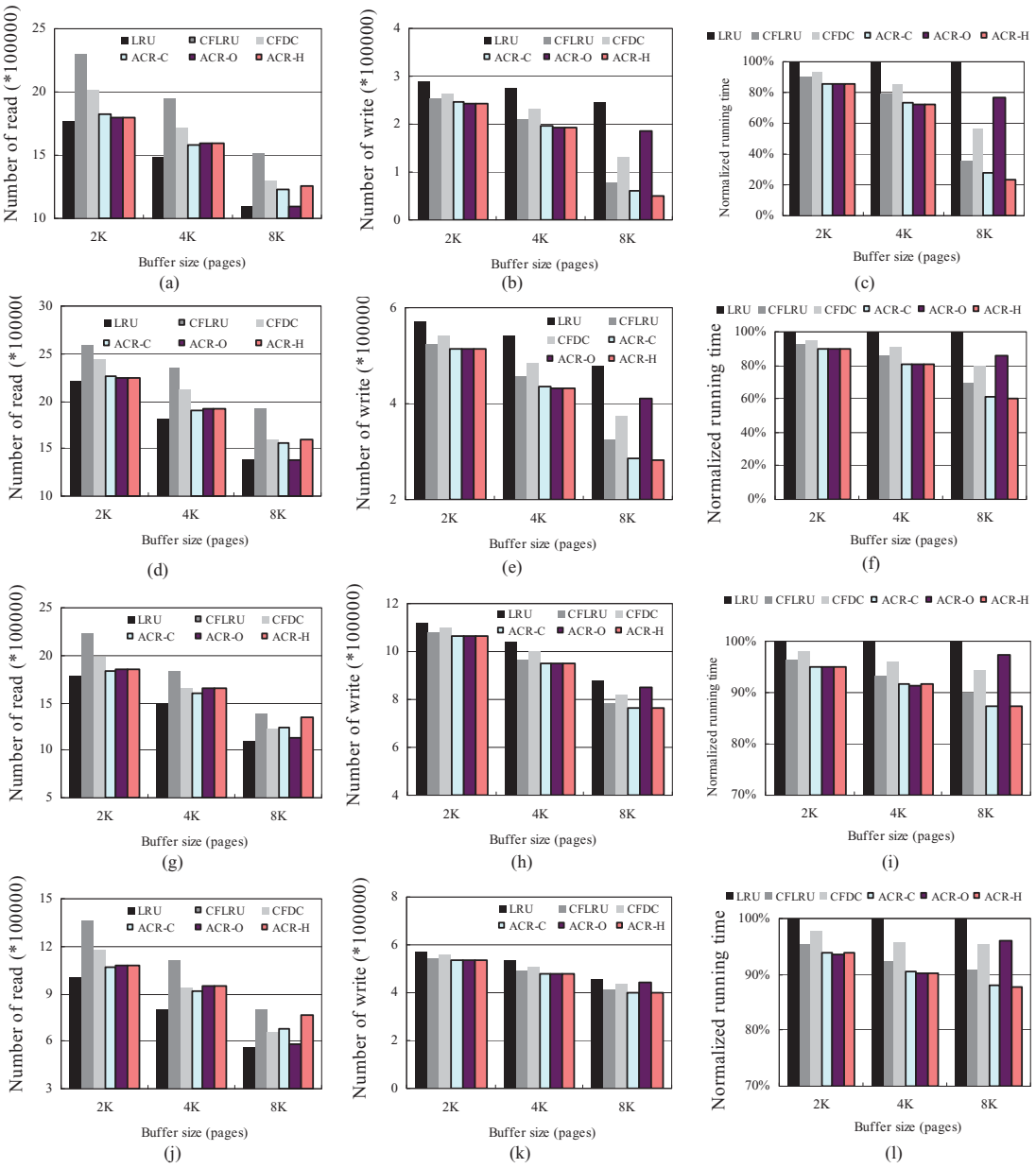


Fig. 4: The comparison of random read, random write and normalized running time on trace T1 to T4 for Samsung MCAQE32G5APP flash disk, (a) to (c) is the result for trace T1, (d) to (f) for trace T2, (g) to (i) for trace T3, and (j) to (l) for trace T4.

and CFDC, the reason lies in that CFLRU and CFDC firstly page out clean pages arbitrarily. If the dirty pages in the buffer are not re-referenced in the near future, then many clean pages will be paged out after they are paged in the buffer for a little time, which will cause many read operations.

In a summary, ACR-C, ACR-O and ACR-H work very efficient when being applied to flash disks with different ratio of read and write costs. Moreover, ACR-C, ACR-O and ACR-H can also work very efficient on workloads of different access patterns.

V. Conclusions

Considering the fact that the discrepancy of the ratio of write cost to read cost for different flash disks various largely and has great affect on designing flash-based buffer replacement policy, in this paper, we address this problem and propose an adaptive cost-based replacement policy, namely ACR. Different from the previous buffer replacement policies that focus on either the various access patterns with uniform access cost, or the asymmetry of access cost for flash, ACR considers

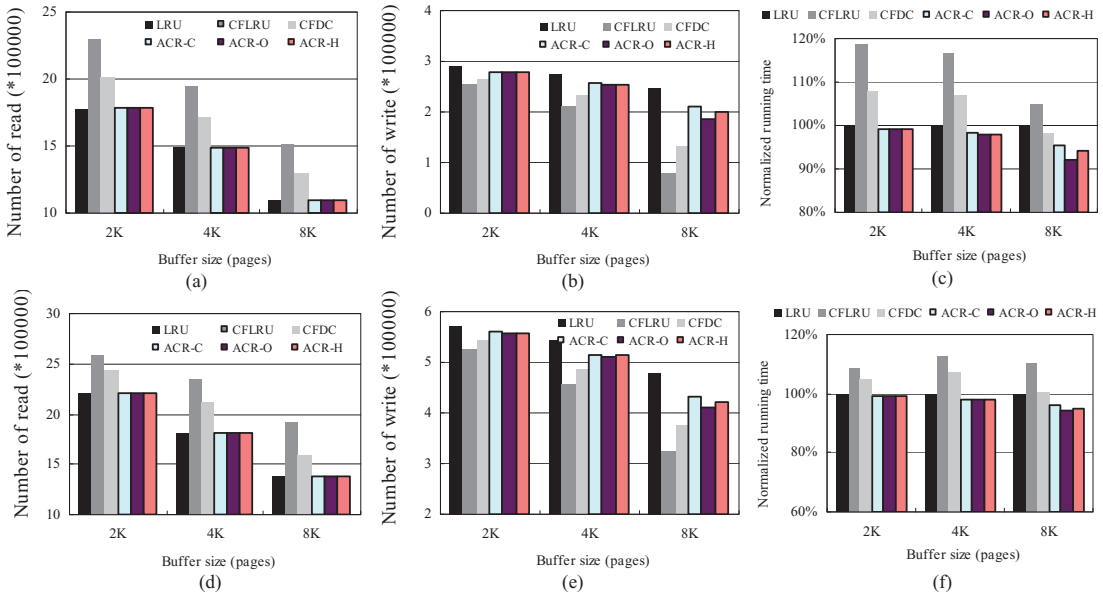


Fig. 5: The comparison of random read, random write and normalized running time on trace T1 and T2 for Samsung MCAQE32G8APP-0XA flash disk, (a) to (c) is the result for trace T1, (d) to (f) for trace T2.

all the above aspects and 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, thus the once-requested pages can be flushed out from the buffer quickly and the frequently-requested pages can stay in the buffer for a longer time. Moreover, ACR uses three cost-based heuristics to select the victim page, thus can fairly make trade off between clean pages and dirty pages. The experimental results on different traces and flash disks show that ACR not only adaptively tunes itself to workloads of different access patterns, but also works well for different kinds of flash disks compared with existing methods.

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

ACKNOWLEDGMENT

This research was partially supported by the grants from the Natural Science Foundation of China (No.60833005); the National High-Tech Research and Development Plan of China (No.2009AA011904); and the Doctoral Fund of Ministry of Education of China (No. 200800020002).

REFERENCES

[1] C. gyu Hwang, "Nanotechnology enables a new memory growth model," in *Proceedings of the IEEE*, 2003, pp. 1765– 1771.
[2] "Windows vista operating system: Readyboost," in *Microsoft Corp.*
[3] S.-W. Lee and B. Moon, "Design of flash-based dbms: an in-page logging approach," in *SIGMOD Conference*, 2007, pp. 55–66.
[4] Ö. Babaoglu and W. N. Joy, "Converting a swap-based system to do paging in an architecture lacking page-reference bits," in *SOSP*, 1981, pp. 78–86.
[5] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *SIGMETRICS*, 1990, pp. 134–142.
[6] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *SIGMOD Conference*, 1993, pp. 297–306.
[7] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *VLDB*, 1994, pp. 439–450.
[8] S. Jiang and X. Zhang, "Making lru friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance," *IEEE Trans. Computers*, vol. 54, no. 8, pp. 939–952, 2005.
[9] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *FAST*, 2003.
[10] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C.-S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.
[11] W. Effelsberg and T. Härder, "Principles of database buffer management," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 560–595, 1984.
[12] S.-Y. Park, D. Jung, J.-U. Kang, J. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *CASES*, 2006, pp. 234–241.
[13] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "Fab: flash-aware buffer management policy for portable media players," in *IEEE Transactions on Consumer Electronics*, 2006, pp. 485–493.
[14] H. Kim and S. Ahn, "Bplru: A buffer management scheme for improving random writes in flash storage," in *FAST*, 2008, pp. 239–252.
[15] I. Koltsidas and S. Viglas, "Flashing up the storage layer," *PVLDB*, vol. 1, no. 1, pp. 514–525, 2008.
[16] Y. Ou, T. Härder, and P. Jin, "Cfdc: a flash-aware replacement policy for database buffer management," in *DaMoN*, 2009, pp. 15–20.
[17] "http://www.datasheetcatalog.net."
[18] F. Douglis, R. Cáceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers," in *OSDI*, 1994, pp. 25–37.
[19] M. Chrobak, H. J. Karloff, T. H. Payne, and S. Vishwanathan, "New results on server problems," *SIAM J. Discrete Math.*, vol. 4, no. 2, pp. 172–181, 1991.
[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001.
[21] P. Jin, X. Su, and Z. Li, "A flexible simulation environment for flash-aware algorithms," in *CIKM*, 2009.