Operation-Aware Buffer Management in Flash-based Systems

Yanfei Lv[1] Bin Cui[1] Bingsheng He [2] Xuexuan Chen[1]

[1] Department of Computer Science & Key Lab of High Confidence Software Technologies (Ministry of Education), Peking University
{lvyf,bin.cui,xuexuan}@pku.edu.cn

[2] Division of Computer Science School of Computer Engineering Nanyang Technological University
bshe@ntu.edu.sg

ABSTRACT

The inherent asymmetry of read and write speeds of flash memory poses great challenges for buffer management design. Most of existing flash-based buffer management policies adopt disk-oriented strategies by giving a specific priority to dirty pages, while not fully exploiting the characteristics of the flash memory. In this paper, we propose a novel buffer replacement algorithm named FOR, which stands for ***F***lash-based ***O***peration-aware buffer ***R***eplacement. The core idea of FOR is based on novel operation-aware page weight determination for buffer replacement. The weight metric not only measures the locality of read/write operations on a page, but also takes the cost difference of read/write operations into account. We further develop an efficient implementation FOR^+ with the time complexity of $O(1)$ for each operation. Experiments on synthetic and benchmark traces demonstrate the efficiency of the proposed strategy, which yields better performance compared with some state-of-the-art flash-based buffer management policies.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Query processing*

General Terms

Algorithms, Design, Performance

Keywords

Flash memory, SSD, Buffer management, Operation aware

1. INTRODUCTION

Flash memory has dominated the storage for portable devices due to its advantages of light weight, power saving and shock resistance. With the increasing capacity and dropping price, flash-based solid state drive (SSD, or flash disks) has emerged as a popular storage for enterprise applications [7, 20]. Due to the characteristics of flash based storage, flash-based data structures and search algorithms [4, 14, 16, 18, 29, 30] on database systems have become a fruitful research field. Buffer manager is another core component in database systems, and the buffer replacement strategies need to be revisited on the flash-based storage.

The effectiveness of buffer management policies can heavily influence the performance of database systems. While flash disks have much better I/O performance than hard disks, they are still over two orders of magnitude slower than the main memory. An effective policy can reduce the number of costly disk accesses, and boost the overall performance of the database system. Thus there have been considerable interests in buffer management algorithms for database systems. Traditional disk-based policies [10, 8] mainly aim at reducing the rate of buffer misses that cause disk accesses. When a buffer page needs to be replaced, the victim is chosen based on the recency and/or frequency of the accesses on the page. However, minimizing the cache miss rate may achieve a sub-optimal I/O performance on flash-based systems. This is because, compared with hard disks, flash disks have an inherent feature, namely *read-write asymmetry*. The read-write asymmetry is that, write operations can be an order of magnitude slower than reads on the flash disk. Due to this asymmetry, buffer manager must distinguish read and write operations. For example, the benefit of keeping a write-intensive page tends to be higher than that of keeping a read-intensive one.

A number of buffer management policies [11, 12, 19, 21, 25] have been proposed to address the read-write asymmetry on the flash disk. Most of them adopt the disk-oriented buffer replacement strategies, such as LRU (Least Recently Used) and LIRS [8], but give a higher priority to the dirty pages to be kept in the buffer. The intuition is straightforward: replacement of a dirty page introduces a write on the flash disk, which is much more expensive than evicting a clean page. For example, CFLRU [25] maintains a window at the least used end of the LRU queue. Within the window, clean pages are preferred as victims and dirty pages are considered only when there is no clean page in the window. A series of WSR [11, 12] (Write Sequential Reorder) methods give another chance to a dirty page upon its eviction. While existing flash-based algorithms improve the performance of database systems on the flash disk, they simply improve the traditional disk-based buffer replacement policies with the consideration of current pages' states, and the

SIGMOD'11, June 12–16, 2011, Athens, Greece.

temporal locality of the future *operations* is ignored. When multiple write operations perform on a dirty page, they result in at most one write operation, if the page is kept in the buffer. On the other hand, the dirty page that has no future write operations or has been inactive for long time should be evicted to free the precious buffer resource. Therefore, both the page state and the future operations are important for the buffer management.

In this paper, we argue for an integral combination between the future operations and the page state to define the *hotness* of a page in the buffer. The combination also takes the asymmetry of read and write speeds of the flash disk into account. The output of the combination is to assign a deliberate weight to each page in the buffer. The lower weight indicates that it is more likely to be a victim page. We start with quantifying the benefit of keeping a page in the buffer by distinguishing the page state (clean or dirty) and its next operation (read or write). For example, given a dirty page with a next operation (read), we can save a page read if we keep the page in the buffer. As for future operations, we introduce inter-operation distance (IOD) and the operation recency (OR) for a page operation to quantify its frequency and recency, respectively. Partial weights based on the metrics are finally combined into one.

We further design the algorithms of FOR, a ***F***lash-based ***O***peration-aware buffer ***R***eplacement policy. The core component of FOR is the page weight calculation. A naive policy is to maintain the weight of each page upon each operation, and to choose the page with the lowest weight as the victim. However, this naive policy requires $O(n)$ time to handle a page operation, where n is the number of distinct pages that have been accessed. This is not efficient in real database systems. Therefore, we develop an approximated version of FOR, named FOR^+. The approximation adopts the metrics of the weight with binary representations to approximate the combination of operation type and page state. Additionally, FOR^+ embraces novel data structures to estimate the hotness of page operation and to select the victim page for buffer replacement. The proposed FOR^+ algorithm requires only $O(1)$ for each operation access in the buffer on average.

We evaluate the performance of the proposed buffer replacement approaches by comparing with the state-of-the-art flash-based algorithms. The experiments are conducted on both synthetic and real traces from public benchmarks including TPC-C, TPC-B, and TATP [3], and our experimental study shows that $FOR+$ approach is superior to the other buffer replacement methods.

The remainder of this paper is organized as follows. We review the related work in Section 2. We present the operation aware weight determination strategy in Section 3, followed by the implementation of our buffer algorithms in Section 4. Section 5 delivers the experimental results. Finally, we conclude the paper in Section 6.

2. PRELIMINARIES

In this section, we briefly introduce the background on flash memory, and review the related work on disk- and flash-based buffer management.

2.1 Flash Memory

Based on the characteristics of flash-based storage, we identify two following challenges for existing disk-based I/O systems. We refer readers to some recent studies [5, 6] for more details on flash memory.

- **Asymmetry of I/O:** Random write operations are much slower than reads, due to the costly erase operations. Note that, read latency and write latency of SSDs vary significantly for different brands. System components such as indexing and buffer management should be aware of this asymmetry.

- **Low access latency:** Flash disks are over an order of magnitude faster than hard disks. Such a low access latency prohibits complicated data structures and algorithms with high computational overhead. For example, selecting a victim in the buffer management should be simple to compensate the I/O benefit.

Flash-based databases have become a hot topic for several years. Detailed studies [5, 6] have been conducted to reveal the internal I/O feature of flash disks. Lee et al. [16] proposed an in-page logging strategy to reduce the number of writes. They [17] further performed extensive experiments on a commercial DBMS, which show that flash can improve the overall performance of DBMS especially in logging. A disk-flash mixed storage system [14] is proposed to adopt the merits of both hard disks and flashes. Flash-optimized data structures are developed to improve indexing performance in databases [30, 4, 18].

2.2 Buffer Management

Buffer management is a critical issue in database systems. There is a long stream of research on designing effective buffer replacement strategies, and many approaches have been proposed. LRU (Least Recently Used) and LFU (Least Frequently Used) are two classic buffer replacement policies for recency and frequency, respectively. Many approaches were proposed to combine the advantages of recency and frequency [22, 15, 28, 26]. Maximizing the buffer hit rate is the main goal of these disk-oriented buffer management strategies. This goal is no longer valid on the flash disk, due to the asymmetry of read and write speeds. This asymmetry calls for new buffer management policies on the flash disk.

A number of flash-based buffer management policies have recently been developed to improve the efficiency of traditional disk-based policies on the flash disk. FAB [9] and BPLRU [13] are two erase block-level buffer management strategies. They are both designed for embedded flash devices. Most existing buffer management policies on flash disk adopt existing disk-based policies by delaying dirty page eviction. CFLRU [25] defines a window in the least recently used end of the LRU queue. All the clean pages in the window are evicted first; dirty pages are evicted only when there is no clean page in the window. LIRS-WSR [12] and LRU-WSR [11] are adopted from LIRS and LRU, respectively. Both of them give higher priority to dirty pages, evicting a dirty page when it is selected as victim at the second time. CCF-LRU [19] improves LRU-WSR by dividing clean pages into hot and cold ones according to the access frequency. The eviction of a clean page is performed on the cold clean pages first and then on the hot clean pages. CASA [23] organizes the buffer as clean page list and dirty page list. Depending on the read/write cost ratio, it dynamically adjusts the size of the two lists. The decision of these flash-based

algorithms (except CCF-LRU) is solely based on the current page state. However, the current page state itself is not sufficient to determine whether the page is a victim or not. Our approach remedies this by taking both the page state and the operation temporal locality into account.

By design, CCFLRU is inferior to *FOR*. CCFLRU fails to detect the hot clean page. When dirty pages fully occupy the entire buffer, it is very difficult for clean ones to get buffer back. This weakness is due to the design of CCFLRU. CCFLRU contains two lists: Cold Clean LRU (CCL) List and Mixed LRU (ML) List. Pages in the CCL list are evicted first. As a result, the CCL list decreases as the system goes, and the CCL list is likely to become empty. In this case, after a newly read page is inserted into the CCL list, this page is evicted immediately when a new page is required. Therefore, unless a page is read twice continuously, a clean page never has a chance to become hot. In comparison, *FOR* does not have this problem, because the read operation information is maintained even after the page is evicted. Therefore, a clean page has the opportunity to become hot when it is read again in *FOR*.

Write clustering has been considered as an effective technique in reducing the total cost [24, 27]. CFDC [24] enhances CFLRU by clustering dirty pages and evicting them as a batch. Similar techniques are used in recently-evicted-first algorithm [27]. Evicting dirty pages as a cluster is an orthogonal solution for buffer management, and it can be integrated into our policy easily to further improve the performance.

3. OPERATION AWARE WEIGHT DETERMINATION

In this section, we present our operation aware strategy to determine the weight of each page for buffer replacement. The weight determination takes the asymmetry of read and write speeds as well as operation-wise statistics such as frequency and recency into account. Table 1 facilitates fast checking on the notations used throughout this paper.

Table 1: Parameters Used in This Paper

Parameters	Description
W_p	Weight of page p
H_e	Hotness of a page operation e
IOD_r, IOD_w	Inter-operation distances of read and write respectively
OR_r, OR_w	Operation recency of read and write respectively
S_{buffer}	Buffer size (pages)
C_r, C_w	Time cost of reading and writing a page respectively
C_{tIO}	Total I/O cost of accessing flash device
C_{tb}	Total buffer cost
N_{df}	Total number of dirty pages flushed (when the system terminates or performs checkpoint)
L_{Upper}, L_{Lower}	Length of Upper queue and Lower queue in FOR^+ respectively
R_{cold}	Ratio of cold pages in the buffer

3.1 Effect of Page State

Read and write operations have similar costs in traditional disk-based database system. Thus, the primary target of buffer management for disk-based databases is to maximize the hit rate of buffer, ignoring what kind of operation is hit. Since the read/write characteristics of flash memory are different from those of hard disk, existing flash-based algorithms further consider cost difference of read/writes, and try to keep more dirty pages in the buffer to reduce the expensive cost of writes to the disk.

Most existing studies adopt total I/O cost to evaluate the performance of buffer replacement strategies. I/O cost may come in two ways. 1) At the run time, a buffer miss will incur a read from the disk device and may incur a write if the evicted page is dirty. 2) At the end of running or checkpointing, system will flush all the dirty pages in the buffer, where the number of writes performed is according to the number of dirty pages. Thus, by accumulating these two parts, we get the following formula for total I/O cost, where C_r and C_w stand for the time cost of performing a read and a write, respectively, and N_{df} stands for number of dirty pages which we need to flush at the end of running or checkpointing.

$$C_{tIO} = \sum_{all\ buffer\ misses} (C_r + evicted\ dirty?C_w : 0) + N_{df}C_w \quad (1)$$

The above formula is obtained by counting the I/O operations to the flash device. However, this is not convenient for buffer performance analysis, as it needs to know whether the evicted page is dirty and the number of dirty pages in the cache. Hence, we transfer the formula from the perspective of buffer operations. In the following analysis, we consider the operations to buffer instead of operations to flash device. We define *buffer cost* to be the cost of each operation to the buffer. Clearly, the buffer cost is subject to whether this operation is a hit or a miss as well as the operation type. We obtain buffer cost for each case in Table 2. The clean or dirty in the table refer to the page state before eviction if the operation is a miss.

Table 2: Buffer Cost for Different Operation Types

Operation	Clean		Dirty	
	miss	hit	miss	hit
read	C_r	0	C_r	0
write	$C_r + C_w$	C_w	$C_r + C_w$	0

It can be observed that a write operation causes a cost of C_r+C_w for a buffer miss while a read causes C_r. A write hit on clean page also results in a cost of C_w. This is because the clean page turns dirty and needs to be written back in the future. For a trace, we can get cost from this table for each operation. Summing up these costs, we get the total buffer cost, recorded as C_{tb}. By comparing the number of read and write operation to flash device respectively, we have $C_{tb} = C_{tIO}$, i.e., the total buffer cost is the same as the total I/O cost.

We further derive the I/O time reduced by keeping a page in the buffer (defined as *buffer benefit*. By comparing miss and hit costs in Table 2, we obtain the buffer benefit of reads and writes in Table 3. Suppose a page is currently held by

Table 3: Buffer Benefit on One Page Operation

Future Operation	Clean	Dirty
Read	C_r	C_r
Write	C_r	C_r+C_w

the buffer waiting for some future operations. There are four cases in total according to the combination of future operations and page states, and each case is analyzed as follows.

1. **Read on clean page:** This means the current state of the page is clean and the next operation on this page is a read operation. If we keep the page in buffer, the page can be used directly without accessing the flash disk, i.e., no extra I/O time is needed. Otherwise, the page has to be loaded into buffer first, a flash read takes place. Thus, I/O time reduced is C_r in this case.

2. **Read on dirty page:** Similar analysis can be done as case 1. If the dirty page is kept in buffer, the operation cost is zero as we can access the buffered page directly. Otherwise, the cost is C_r. The I/O time of dirty page write back is not calculated because this write back will eventually happen afterwards.

3. **Write on clean page:** If the page is kept in the buffer, it will turn dirty. Otherwise, the page should be loaded into the buffer first which incurs read cost, then the page also turns dirty. Hence, the benefit of using buffer is C_r.

4. **Write on dirty page:** As the page is dirty, the write operation can be merged with the previous one if the page is reserved in the buffer, and hence the write cost is saved. On the contrary, a page write to the flash memory will be introduced if the page is evicted, and a page read is used to load it into the buffer. Consequently, the buffer benefit is C_r+C_w.

Only the immediately succeeding operation is enough for victim decision, because no matter whether a page is kept in the buffer, after the following operation, the page will certainly reside in buffer. Thus, the current victim decision can only affect the cost of the immediately following read and write operation. It can be observed from the table that the costs reduced by buffering a dirty page are totally different, which are affected by succeeding read/write operation on the dirty page. Therefore, not only the page state but also the future operation should be considered to determine an appropriate victim for buffer replacement.

3.2 Effect of Operation Locality

How to make full use of the information provided by operation sequence to estimate the hotness of a page is a key issue to solve in our operation-aware buffer management strategy. We denote a page operation as a pair e=$\langle p, m\rangle$, where p denotes the page to be accessed, and m is the access mode ($m \in$ {read, write}). Thus e_i=e_j means that $e_i.p$=$e_j.p$ and $e_i.m$=$e_j.m$. A reference sequence is a sequence of n page operations, $S = e_1, e_2, ..., e_n$, where e_i is a page operation. The latest operation is recorded as e_{latest}. We say $e_i \prec e_j$, if $i < j$ in the sequence, and $e_i \preceq e_j$, if $i \leq j$. And we define the previous same operation of e as $ps(e) = e_i$ if $(e_i = e) \wedge \neg\exists e_j((e_j = e) \wedge (e_i \prec e_j \prec e))$. Note that $ps(e)$ may be null.

Buffering the hot pages can generally increase the hit ratio of buffer replacement methods, and there are various metrics to evaluate the hotness of pages, among which frequency and recency are the commonly used metrics to reflect the hotness. In this study, we define two concepts Inter-Operation Distance (IOD) and Operation Recency (OR) to reflect the frequency and recency of a page operation, respectively.

DEFINITION 1. *Suppose the inter-operation set of a page operation e is*

$$IOS_e = \{e_j | ps(e) \preceq e_j \preceq e\} \tag{2}$$

Inter Operation Distance is defined as:

$$IOD_e = |IOS_e|, if\ ps(e) \neq null \tag{3}$$

IOD_e is the number of distinct operations performed between two consecutive "identical operations" on one page including the operation itself. IOD_e does not exist if $ps(e) = null$. Taking the operation sequence in Figure 1 as an example, the IOD of the last read on page B is 3, and the IOD of the last write on page A is 5. The smaller the IOD_e is, the hotter operation e is. Note that, the average frequency of an operation can be estimated by dividing the length of operation sequence by the number of the occurrences of the operation. However, such calculation is computationally expensive and infeasible in real applications. Therefore, we use IOD to represent the frequency in our study.

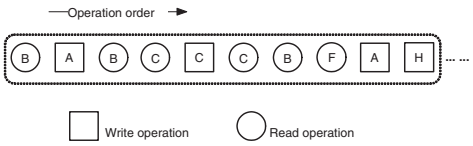


Figure 1: Illustration of inter operation distance

DEFINITION 2. *Operation Recency of a page operation e is defined as:*

$$OR_e = |\{e_j | e \preceq e_j \preceq e_{latest}\}| \tag{4}$$

The definition of OR_e denotes the number of distinct operations performed after e appeared. Note that recency is the potential IOD in the future. If only one e is in the operation sequence, the value of IOD_e does not exist. When the second e, i.e., the same operation on the same page, occurs, OR_e is equal to IOD_e. That means, the future IOD should be no less than the current operation recency.

The two metrics (IOD_e and OR_e) are complementary in measuring the hotness of the corresponding page accessed by e. The OR only reflects the freshness of the page operation, while ignoring the long-term workload pattern. On the other hand, only using IOD is problematic as the recency information is not included. For example, if an operation with low IOD never appears again in the operation sequence, the value of IOD will never change. Therefore, we design to integrate IOD and OR for page hotness determination.

DEFINITION 3. *Hotness of an operation e is reflected by weighted sum of IOD and OR.*

$$H_e = \alpha * IOD_e + (1 - \alpha) * OR_e \tag{5}$$

Where α is a tuning parameter to determine the importance of the two factors, and the experimental tuning evaluation will be presented in our performance study. If IOD_e does not exist, we use the value of OR_e as that of IOD_e in the above definition. H_e indicates the popularity of a page according to the operation statistics.

3.3 Operation-aware Page weight

In Sec 3.1 & 3.2, we have discussed the operation-wise page information which affects the buffer replacement strategy. According to our observation and analysis, we need to consider page state and page operation statistical information to determine the importance of a page for buffer management. We give the page weight in the Definition 4. In the formula, W_p distinguishes the I/O time reduced by buffering the page p as shown in Table 3, as well as the hotness of read/write operations on the page.

DEFINITION 4. *Suppose r and w are the latest read and write operations on page p, respectively. The weight of p (W_p) is defined as:*

$$W_p = \begin{cases} C_r/H_r + C_r/H_w, & p \text{ is clean} \\ C_r/H_r + (C_r + C_w)/H_w, & p \text{ is dirty} \end{cases} \quad (6)$$

Note that, if there is no previous read or write operation on a certain page, the corresponding item in above Formula 6 should be zero. Given the definition of W_p, we can calculate its value and choose the victim for buffer replacement using W_p. The page with the minimum weight is chosen as the victim. The algorithm will be presented in the next section.

4. THE IMPLEMENTATION OF FOR

In this section, we will introduce our algorithm for buffer management based on the concept of operation-aware page weight determination, named FOR, which stands for ***F***lash-based ***O***peration-aware buffer ***R***eplacement. Due to the computational complexity of FOR, we also design an approximated version of FOR, named FOR^+, which achieves $O(1)$ time complexity on handling a page operation.

4.1 The FOR algorithm

Based on the concept of page weight W_p, we develop a buffer replacement algorithm FOR, in which the page with the minimum weight is chosen as the victim. We use an LRU queue named *operation list* to facilitate computation of IOD and OR. As shown in Figure 2, the operation list stores the latest read and write operations for all the pages that have been accessed. Note that, we only use page ID in the list, and the pages may have already been evicted from the buffer. These operations are adjusted in the LRU manner. Thus the number of operations between the head and operation e is OR_e. When operation e comes again, OR_e at that time is IOD_e. On every operation performed, we can calculate its IOD. Thus, OR and IOD can be calculated for every operation in the operation list. Besides the data structure shown in Figure 2, all the control information including pointers to the node in operation list, pointer to the real data page, IOD and the state information of the page are integrated as a structure named *frame*. To accelerate page and information accesses, all the frames are indexed by an in-memory hash table according to page ID.

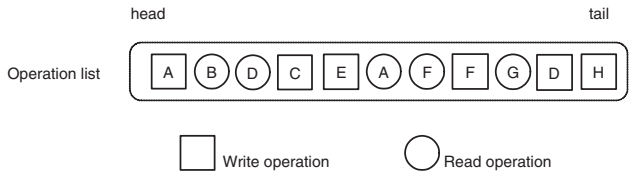


Figure 2: Data structure of FOR

The detailed algorithm of FOR is listed as Algorithm 1. If the buffer is full and thus an eviction is needed, we first calculate ORs and weights for all pages (Lines 2–5), and find a page with the minimum weight as victim (Lines 7-11). After the operation, we calculate IOD if the operation has already existed in the operation list (Line 16).

Algorithm 1: The FOR algorithm

```
   Input: an operation request e on page p
 1 if p is non-resident and buffer is full then
 2 |   foreach operation op in the operation list do
 3 |   |   calculate OR_op;
 4 |   |   calculate the weight of the corresponding
   |   |   page;
 5 |   end
 6 |   W_victim=inf;
 7 |   foreach page p_i in the buffer do
 8 |   |   if W_pi < W_victim then
 9 |   |   |   update victim to p_i;
10 |   |   end
11 |   end
12 |   evict victim;
13 end
14 perform the operation;
15 if e exists on operation list then
16 |   adjust the operation list and calculate IOD_e;
17 else
18 |   add the operation to the head of operation list;
19 end
```

Upon each eviction, FOR calculates the weights for all accessed pages, and chooses the page with the lowest weight as the victim. Suppose the working set of the database system is n pages, and the buffer pool size is S_{buffer} pages. The loops in Lines 2–5 and 7–11 have the time complexity of $O(n)$ and $O(S_{buffer})$, respectively. Since in practice, $S_{buffer} \ll n$, the time complexity of handling a page operation in FOR is $O(n)$. The efficiency is unacceptable in a real database system. This motivates us to reduce the computational complexity while taking the weight concept into account.

4.2 An approximated version FOR^+

The dominant computational cost of FOR algorithm is the calculation of page weights in the buffer. To reduce the time complexity, we have to design an efficient metric to approximate the weight. In contrast to the numerical representation of weight, we adopt categorical weight states to approximate the weight of a page. That leads to an approximated algorithm FOR^+ with $O(1)$ time complexity.

In Equation 6, there are two adding factors for each weight state, including the page state and the operation hotness. So

there are 4 high weight states in total. We can use 4 marks RCH (Read Clean page High), WCH (Write Clean page High), RDH (Read Dirty page High) and WDH (Write Dirty page High) to represent the high weights for C_r/H_r, C_r/H_w, C_r/H_r and $(C_r + C_w)/H_w$, respectively. Among these four marks, RCH and RDH are with same cost and similar semantic, so they can be combined to RH (Read page High). On the other hand, once a page is written, the state of page changes to dirty. So, WCH is volatile and we only reserve the WDH mark for a dirty page. Consequently, we utilize two weight status marks RH and WDH in our proposed FOR^+ approach.

For simplicity, FOR^+ sets the marks according to the value of IOD as the IOD reflects the access frequency, and cleans the marks according to the value of OR to discard the operation without reference for a long time. For example, for a read operation e, if C_r/IOD_e is relatively high among all the operations, we set e's RH.

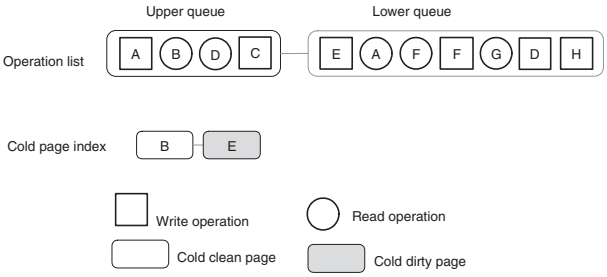


Figure 3: Data structure of FOR^+

FOR^+ extends the data structure shown in Figure 2 to facilitate the weight status mark determination. The operation list is divided into two virtual queues, i.e., Upper and Lower, as illustrated in Figure 3. Overall, these two queues work in the LRU manner as a whole, that is, the operation evicted from Upper queue is inserted into Lower queue as the head of Lower queue, while the operation hit on Lower queue or Upper queue is moved to the head of Upper queue. The most recent operation is inserted as the head of Upper queue. Thus, it can be observed easily that the number of operations from the head of Upper queue to an operation e is OR_e, and if an identical operation of e arrives, OR_e at that moment is IOD_e. For example, in Figure 3 the OR of operation write E is 5, and if a write E happens at this moment, IOD of write E is 5.

In FOR^+, we try to ensure $L_{Upper}/C_r = L_{Lower}/C_w$ during the buffer process, where L_{Upper} and L_{Lower} stand for length of Upper and Lower queues respectively. Thus, we can derive: $C_r/L_{Upper} = (C_r+C_w)/(L_{Lower}+L_{Upper})$, which is denoted as M_{bound}. We use the M_{bound} as the threshold to allocate the marks. Take the read operation e on B, in the Upper queue as an example, when another e comes, we have $IOD_e \leq L_{Upper}$, that is, $C_r/IOD_e \geq M_{bound}$. Thus, B's RH mark is set. Similarly, if a write hit on either Upper or Lower queue means $(C_r + C_w)/IOD_e \geq M_{bound}$, we set the WDH mark. Initially, we assume both marks of newly arrived pages are cleared.

The entire buffer pool in FOR^+ is virtually divided into two sub-pools, one for hot pages and the other for cold pages. A page is cold if it is a clean page without setting the RH mark, or it is a dirty page without setting RH or WDH. Otherwise, the page is hot. In FOR^+, only the cold pages can be selected as the victim for buffer replacement. FOR^+ ensures a certain ratio of cold pages (denoted as R_{cold}) in the buffer, which is a tuning parameter in our algorithm. If the corresponding marks of a page are set, the status of the page will be upgraded to hot. Meanwhile, if the ratio of cold pages in the buffer is lower than R_{cold}, we have to downgrade a page to the cold page pool, and this process is named as "Compensation". The byproduct of the compensation process is to maintain the ratio between L_{Upper} and L_{Lower}, and to shrink Upper queue or Lower queue to clear the marks of less important pages.

The process of compensation is realized by shrinking Upper queue and Lower queue in turn. If $L_{Upper}/C_r > L_{Lower}/C_w$, the Upper queue shrinks; otherwise, the Lower queue does. To shrink an operation queue, the tail operation of the queue is removed and the mark of that page is cleared. According to the operation types in different queues, we have following cases corresponding to the mark status: 1) If a read operation is removed from Upper queue, the RH mark of the corresponding page is cleared; 2) If a write operation is removed from Lower queue, the WDH mark is cleared. After that, a cold test is performed on the mark-cleared page, i.e., whether clean page with RH cleared, or a dirty page with RH and WDH cleared. If the page is changed to cold, the compensation process ends. Otherwise, the process above repeats until one hot page is changed to cold. The tail operation removed from lower queue will be deleted to save the space used, and this operation will be treated as new if the page is accessed again in the future.

Algorithm 2: The FOR$^+$ Algorithm

```
   Input: an operation request, e
 1 if the frame does not exist in hash table then
 2 |  create a new frame;
 3 end
 4 get the frame from hash table;
 5 if the frame is nonresident in buffer then
 6 |  Allocate free slot(Algorithm 3);
 7 end
 8 perform the operation;
 9 if e is read and hit on Upper queue then
10 |  set e's RH mark;
11 end
12 if e is write and (hit on Upper queue or on Lower
   queue) then
13 |  set e's WDH mark;
14 end
15 adjust the Upper queue and Lower queue in LRU
   manner as a whole;
16 if the page operated is in cold page index then
17 |  if the page operated is cold then
18 |  |  adjust the cold page index in the LRU
   |  |  manner;
19 |  else
20 |  |  remove the page from cold page index;
21 |  end
22 end
23 L_cold= the number of cold pages in the buffer;
24 if L_cold < R_cold * S_buffer then
25 |  compensation (Algorithm 4);
26 end
```

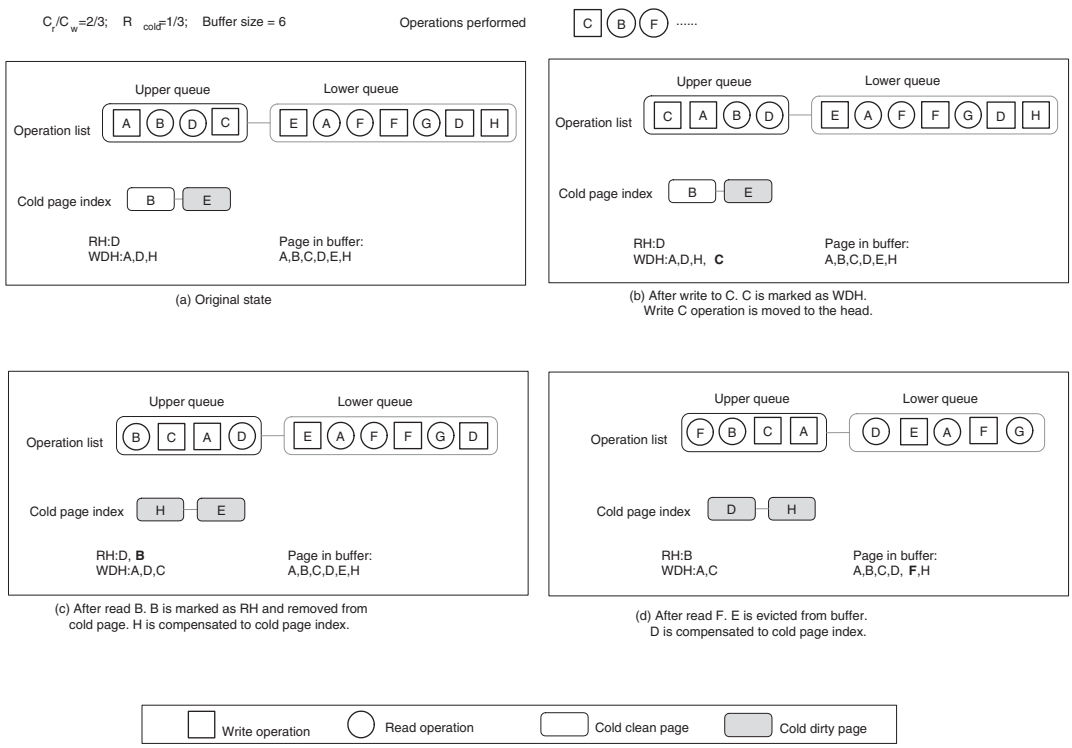


Figure 4: An example of FOR^+

A cold page index is used to accelerate the eviction, as illustrated in Figure 3. The cold page pool works in the LRU manner. When a free slot is needed from the buffer, the cold page which is not referenced for the longest time is chosen as the victim. Page id instead of operation is stored in cold page index. Cold page index is a queue acting in the LRU manner. A new page is inserted at the head of cold page index. Every access hit on the index will cause the referenced page to be moved to the head, if the page is not upgraded to hot. Hence, the tail page is selected as the victim when a free slot is needed. With the help of cold page index, the victim can be determined quickly when an eviction takes place.

Algorithm 3: Allocate a free slot in FOR^+

```
1 if buffer is not full then
2 |  return any free slot;
3 else
4 |  get the tail from the cold page index;
5 |  if the page is dirty then
6 |  |  write back;
7 |  end
8 |  free and return the slot;
9 end
```

The overall process of FOR^+ algorithm is given in Algorithm 2. We extend a frame in FOR with the two weight status marks. The algorithm first gets the frame information and performs the operation (Lines 1-8). When a free page is requested, Algorithm 3 is called. After the operation is performed, the marks (Lines 9-14) and data structures (Lines 15-22) are updated. The queue adjustment process (Line 15) includes 1) inserting a new item into the head and 2) adjusting the queue in the LRU manner if one item is hit on the queue. The operation hit on Lower queue is also moved to the head of Upper queue. If the buffer pool is full, one page will be evicted from the buffer. Once a cold page is changed to hot, compensation may be called to change a hot page to

Algorithm 4: Compensation in FOR^+

```
1  repeat
2  |  if L_Upper /C_r > L_Lower/C_w then    /* Shrink
   |  Upper queue */
3  |  |  remove the tail operation from Upper queue;
4  |  |  insert the operation into Lower queue;
5  |  |  if the operation is read then
6  |  |  |  clear the RH mark;
7  |  |  end
8  |  else                    /* Shrink Lower queue */
9  |  |  remove the tail operation from Lower queue;
10 |  |  if the operation is write then
11 |  |  |  clear the WDH mark;
12 |  |  end
13 |  end
14 |  change = whether the clear marked page is
   |  changed from hot to cold;
15 |  if change then
16 |  |  add the page to the head of the cold page
   |  |  index;
17 |  end
18 until a cold page is added ;
```

a cold one (Lines 23-26) so that the number of cold page is constant in the buffer. Algorithm 4 illustrates the process of compensation. Compensation is done by shrinking Upper queue and Lower queue in turn, during which some marks are cleared, until a hot page is changed to cold.

An example is given in Figure 4 illustrating the process of FOR^+. Figure 4 (a) shows the initial state of the operation list and the cold page index. Suppose a write on the page C. According to FOR^+, the buffer state is shown in Figure 4 (b). Operation write C is moved to the head of Upper queue. Since this operation is a hit on the operation list, C's WDH is set. Similarly, we can handle a read on page B, and the buffer state is shown in Figure 4 (c). As reading B is a hit on Upper queue, B's RH is set, and B is removed from cold page index. The compensation process is invoked, because $L_{Upper}/C_r > L_{Lower}/C_w$. The operation at the end of Lower queue, operation H, is removed and inserted to the head of the cold page index. After that, reading F is performed, and a buffer miss occurs (Figure 4 (d)). To free a page slot, the end of the cold page index, E, is evicted from the buffer. As $L_{cold}<R_{cold} * S_{buffer}$, one page will be compensated. First, the write operation on D is removed from Lower queue, and its WDH mark is cleared. However, D is still not cold, so we continue the compensation process. As $L_{Upper}/C_r < L_{Lower}/C_w$, the read operation on D is moved from the tail of Upper queue to the head of Lower queue and RH operation of D is cleared. Thus, page D has no mark set, and it is inserted into the cold page index.

Since we simply record the page operation and page ID in the queue, our approach only introduces negligible space overhead compared with the buffered data. Each operation insertion and removal have the time complexity of $O(1)$. Note that, one operation is only removed once during the compensation process.

5. PERFORMANCE STUDY

In this section, we conduct a trace-driven simulation including benchmark and synthetic workloads to evaluate the effectiveness of our approach, and the experimental results are illustrated in comparison with some state-of-the-art flash-based buffer replacement algorithms, including CFLRU [25], LRUWSR [11], CCFLRU [19] and CASA [23]. The simulation is developed in Visual Studio 2010 using C#. All experiments are run on a Windows 7 PC with a 2.4 GHz Intel Quad CPU and 2 GB of physical memory.

5.1 Experimental setup

We use both real and synthetic traces for performance evaluation. Three benchmark traces on transactional processing are deployed, namely TATP [3], TPC-B, and TPC-C. To get the traces on buffer page accesses, we run the three benchmarks on PostgreSQL 8.3.5 with default settings, e.g., the page size is 8KB. For each benchmark, we run a sufficient period of time around 3 hours, including a 30-minute warm-up period. For all benchmarks, the number of clients is 20. Specification on the traces was given in Table 4. For the synthetic trace, we generate a operation sequence conforming to 80/20 distribution as those in the previous study [10], and an operation is randomly assigned as a read or a write such that the ratio of write operations ranges from 10% to 50%.

We consider two performance metrics: $Buffer\ Hit\ Ratio$ which records the percentage of page hits in the buffer, and

Table 4: Specification on the Traces

	Page Number (thousands)	Reference Number (millions)	Write Ratio
TATP	50.6	2.5	4.6%
TPC-B	259	12.7	3.3%
TPC-C	95.7	16.8	16.3%
Synthetic	300	10	10%-50%

I/O cost per operation which is the result of *total I/O time* divided by *number of operations*. In our experiment, I/O cost per operation is used to evaluate the performance besides the traditional hit rate metric. The reason is that the read and write latency in flash-based systems are different, which actually motivates our work. We employ two types of settings for read and write asymmetry. One represents the NAND flash, i.e., C_r and C_w are $100\mu s$ and $800\mu s$ respectively for the 8K page [1]. We select one popular Samsung SSD [2] as the SSD representative, also used in [17, 18]. The parameters used in our experiments are listed in Table 5. Unless stated explicitly, the default parameter values, given in bold, are used.

Table 5: Experimental Parameters

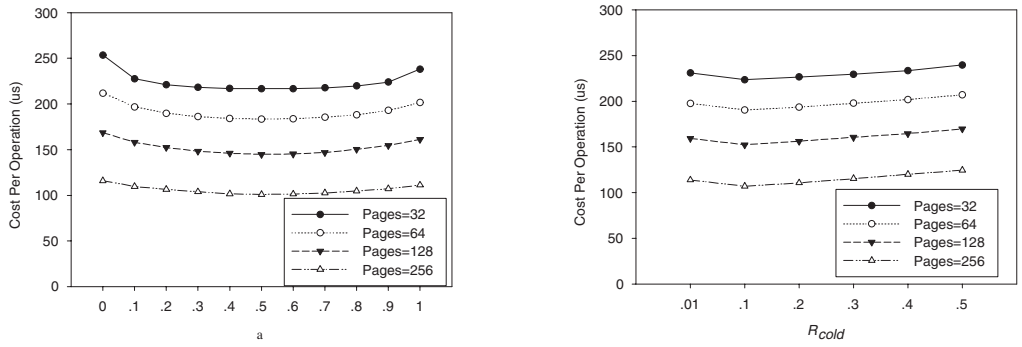
Parameter	Value
$S_{buffer}(pages)$	32, 64, **128**, ..., 16384
C_r, $C_w(\mu s)$	**100, 800** (for flash chip) 245, 9663 (for Samsung SSD)
α	0, 0.1, 0.2, ..., 1
R_{cold}	0.01, **0.1**, 0.2, ..., 0.5

5.2 Parameter tuning of FOR approaches

We first evaluate the effect of the tuning parameter α in FOR and R_{cold} in FOR^+ on the buffer performance. α shown in $Definition$ 3 is a tuning parameter for the importance of IOD and OR in page weight determination, and R_{cold} is the percentage of cold pages which can be selected as a victim for buffer replacement FOR^+. We use the synthetic trace in the default setting. As FOR is time consuming, we generate a trace with 1000 page operations.

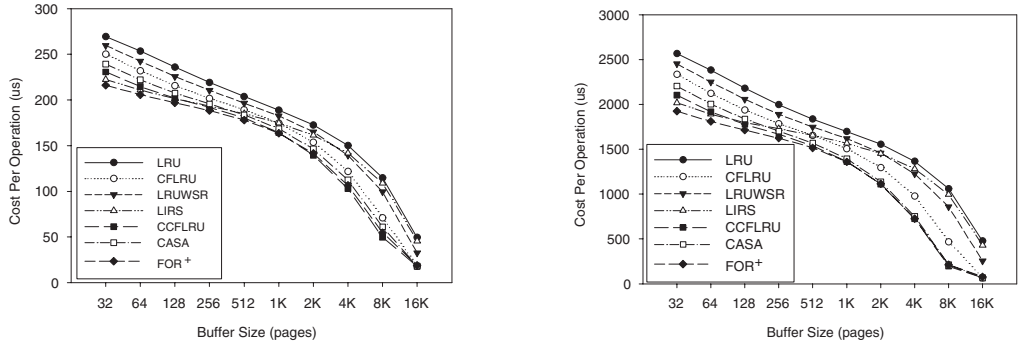
As shown in Figure 5 (a), the performance curve is concave, meaning that there is an optimal α that should be used. When α is zero, only the recency information is considered; when $\alpha = 1$, the frequency is the only used information. The best performance is obtained when α is around 0.4 - 0.6, indicating that we should take both frequency and recency into account for optimal performance. Thus, IOD and OR are integrated to measure the hotness of the corresponding page in FOR. Figure 5 (b) shows the performance of FOR^+ by varying R_{cold}. If R_{cold} is too small, a recently used page may be evicted, because the page becomes hot only when it has been accessed more than twice. On the other hand, a larger R_{cold} decreases the buffer space used to store the frequently used pages. In our experiment, the optimal value for R_{cold} is 0.1.

Comparing the I/O cost per operation of FOR and FOR^+ in Figure 5, we found FOR^+ performs only slightly worse than FOR. This shows that the approximation in FOR^+ can effectively model the page hotness. However, it takes FOR over 120 times longer than FOR^+ to run this trace, due to the high time complexity of operation process in



(a) α (b) R_{cold}

Figure 5: Parameters Tuning



(a) I/O cost of synthetic trace on flash chip (b) I/O cost of synthetic trace on SSD

Figure 6: Buffer replacement performance on synthetic workload

FOR. Therefore, FOR^+ is a more promising choice for buffer replacement in real flash-based systems, and we only use FOR^+ in the following experiments.

5.3 Comparison with other techniques

We first demonstrate the results on synthetic workloads, followed by more experimental results on TATP, TPC-B and TPC-C.

5.3.1 Results on Synthetic Workloads

The experimental results on synthetic traces are shown in Figure 6. The write ratio is fixed to be 20%. The window size of CFLRU is set to 0.5, and the parameter of LIRS is set to 0.01. The buffer size ranges from 32 to 16384 pages. As the buffer size increases, the average operation cost of all the algorithms decreases rapidly. Overall, FOR^+ exceeds CASA by 5%-10%, has similar performance to CCFLRU, and significantly outperforms other algorithms by up to 100% on flash chip. As CCFLRU always allocates a rather high priority on dirty page, it can prevent the write operations as much as possible at the cost of more read operations. We can also see CCFLRU is more sensitive to traces in our later experiments on public benchmarks. LRUWSR, CFLRU and LRU have similar performance trends, as both LRUWSR and CFLRU are based on LRU. LIRS always outperforms LRU, but the difference becomes smaller as the buffer size increases. This is because LIRS keeps some historical information, which is more useful for a small buffer. The difference between FOR^+ and other algorithms is more significant on SSD, since the I/O asymmetry is larger on SSD.

Figure 7 shows the performance by varying the write ratio and with the buffer size fixed to 4096. The I/O costs per operation of all the algorithms increase, as the write ratio increases. The CCFLRU and FOR^+ perform similarly and better than others (e.g., up to 30% lower I/O cost than CFLRU), since both of them gives higher priority to dirty pages than other algorithms.

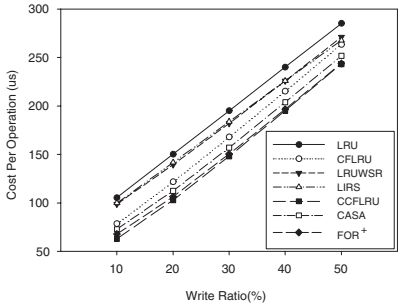
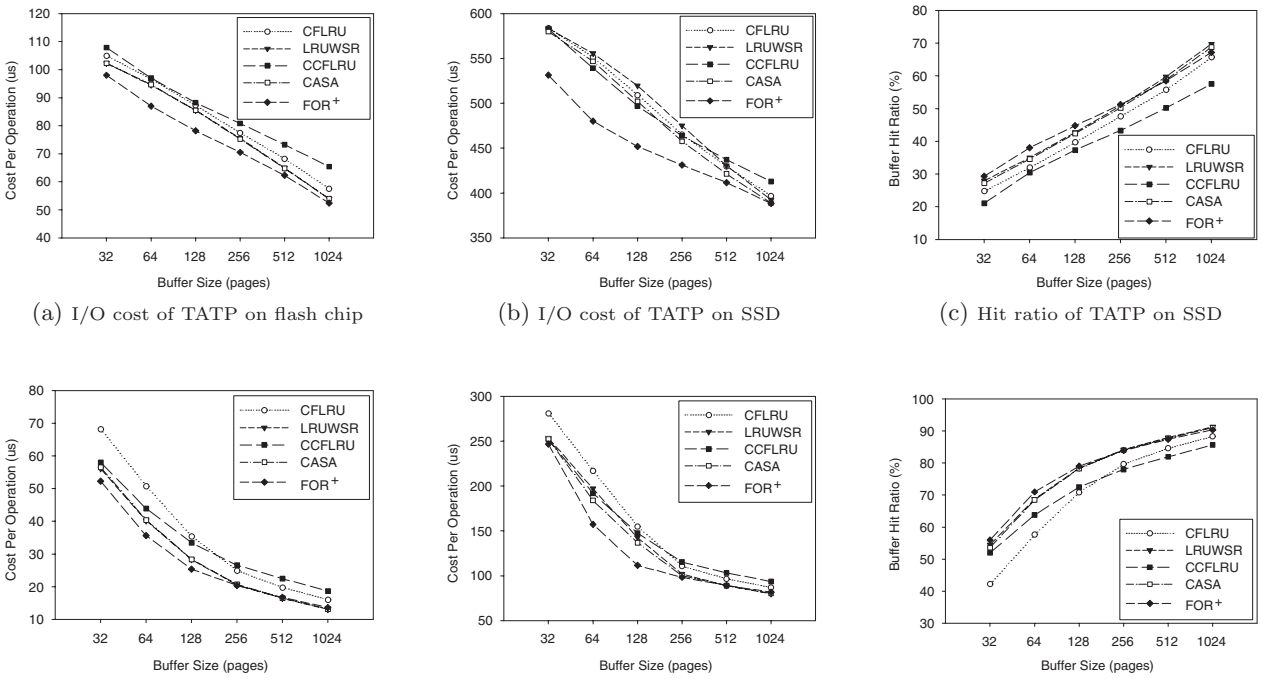


Figure 7: Performance effected by write ratio

5.3.2 Results on Benchmark Workloads

We first focus on comparing FOR^+ with other flash-based algorithms. Figures 8 (a), (b), (d) and (e) show I/O costs per



(a) I/O cost of TATP on flash chip (b) I/O cost of TATP on SSD (c) Hit ratio of TATP on SSD

(d) I/O cost of TPC-B on flash chip (e) I/O cost of TPC-B on SSD (f) Hit ratio of TPC-B on SSD

Figure 8: Comparison on buffer replacement performance

Table 6: Read and write counts to SSD on TATP for different algorithms

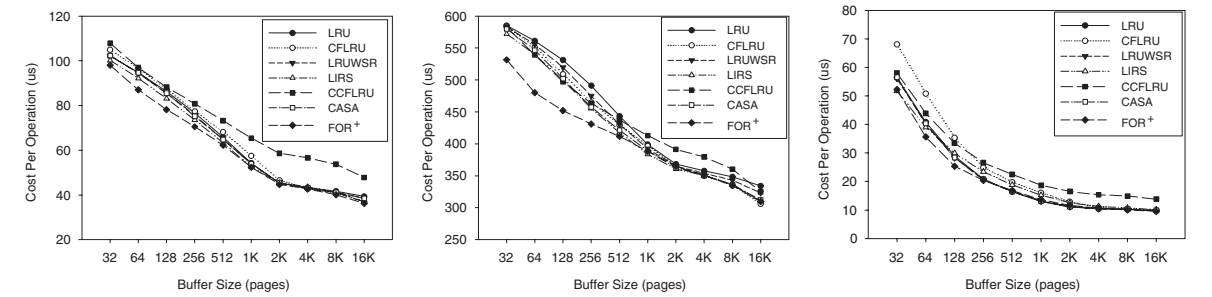
	CFLRU		LRUWSR		CCFLRU		CASA		FOR$^+$	
Buffer Size(pages)	read	write	read	write	read	write	read	write	read	write
32	1766205	105547	1685335	107528	1861041	102910	1705125	106094	1664170	94693
64	1591511	101583	1512794	104724	1633644	97465	1526095	102114	1455070	86751
128	1406144	95445	1325756	100226	1469433	90757	1338712	95374	1291214	83650
256	1214153	88971	1130193	93707	1325149	85819	1149824	88693	1131086	82368
512	1017177	84863	914505	87668	1156174	83295	940055	84687	951406	81901
1024	770001	82629	669042	84077	975519	81596	693858	82548	736001	81410

operation of different approaches by varying the buffer size. FOR^+ is about 5–20% better than other algorithms averagely. The four competitors are based on LRU, which only takes recency and state of the page into consideration. However, The proposed FOR^+ algorithm exploits the read/write asymmetry and operation statistics for buffer replacement, thus it is more adaptive to system characteristics. The clean pages are much easier to be evicted in CFLRU than LRUWSR, because the cold clean page is evicted first, only if there is no cold clean page, victim will be selected on other pages as LRUWSR. Even though the buffer size is large, too many clean pages are evicted by CCFLRU which increases the number of read operation. This is why CCFLRU is worse than CFLRU and LRUWSR for large buffer size. CASA has very similar performance with LRUWSR. CASA only employs the access information of pages in the buffer which losses most of the historical information compared with FOR^+.

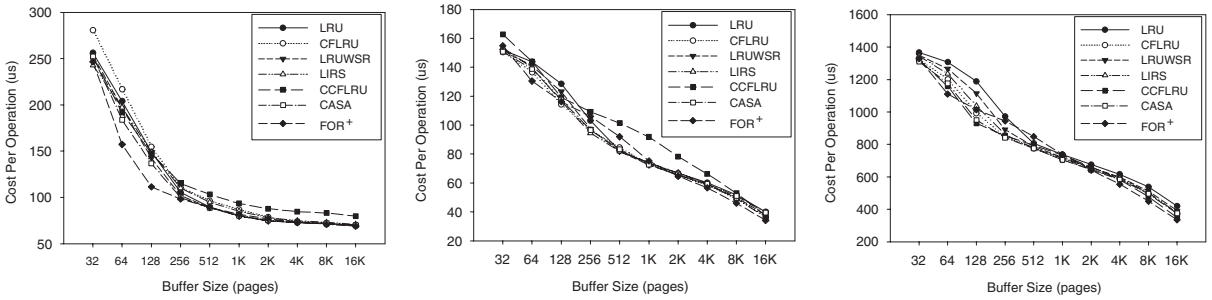
The I/O cost of each algorithm on SSD is illustrated in Figures 8 (b) & (e). FOR^+ yields more performance advantage over the competitors. Since the C_w/C_r of SSD is much larger than flash chip, the performance difference is more significant on SSD. CCFLRU performs relatively better on SSD compared with the case on flash chip, because the increment of C_w/C_r makes it more worthwhile to hold dirty pages in the buffer.

Figures 8 (c) and (f) show buffer hit ratios for SSD settings. The number of I/O read/write operations for TATP are also listed in Table 6. From the figures, we observe that the hit ratio of every algorithm increases with the enlargement of buffer size. All the flash-based algorithms aim to reduce some write operations at the expense of more read operations. The hit rate gain of our approach FOR^+ is not significant as that on I/O cost. From Table 6, we can see that the number of I/O write operations of FOR^+ is smaller than thoseof other algorithms, which means FOR^+ provides a better hit ratio of write operations. CFLRU always holds dirty pages in its window, while LRUWSR gives dirty pages only one more chance to be kept in the buffer. Consequently, the number of I/O write operations of CFLRU is smaller than that of LRUWSR. CFLRU decreases the effective buffer space for clean pages. The hit ratio of CFLRU is clearly lower than LRUWSR.

FOR^+ outperforms CCFLRU, because CCFLRU fails to detect hot clean pages. As Table 6 shows, CCFLRU often has a much larger number of read operations than FOR^+.



(a) I/O cost of TATP on flash chip (b) I/O cost of TATP on SSD (c) I/O cost of TPC-B on flash chip



(d) I/O cost of TPC-B on SSD (e) I/O cost of TPC-C on flash chip (f) I/O cost of TPC-C on SSD

Figure 9: More results on buffer replacement performance

Among these competitors, CCFLRU introduces most read operations in all cases.

More experimental comparisons are included in Figure 9. Besides the flash-based methods, we also include two traditional disk-based algorithms, i.e., LRU and LIRS [8]. We show the experimental results with larger range of buffer size from 32 to 16382 pages. The proposed FOR^+ is the best algorithm in most cases. The comparison on the hit ratio is similar to those in Figure 8, and we omit the results.

The performance gain decreases when the buffer size increases. This phenomena is reasonable and consistent with the findings in previous buffer management studies [11, 19]. As the buffer size increases, all the algorithms can hold more pages in the buffer. The hit ratio increases and the room for performance improvement becomes small.

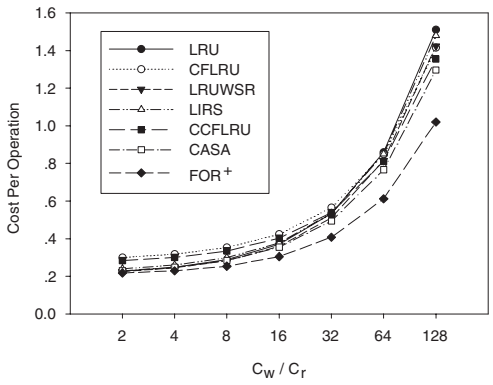


Figure 10: Performance effected by C_w/C_r

FOR^+ is able to adjust its policy according to flash characteristics in order to achieve better performance. The performance with varying C_w/C_r is given in Figure 10. We normalize C_r to be one, and range C_w from 2 to 128, so that the C_w/C_r cover the situations of most SSDs. The experiments are performed on TPC-B with buffer size 128 pages. As the ratio increases, the performance advantage of FOR^+ over other algorithm becomes larger. Our proposed approach well exploits the locality of read/write operation as well as the read/write asymmetry of the flash memory.

We further investigate the impact of different database sizes. Table 7 shows the I/O cost per operation for the TPC-C traces on 0.5GB to 4GB databases. We fix the buffer to 10% of total database size. FOR^+'s consistently outperforms other algorithms for different database sizes.

Table 7: Performance on TPC-C(μs) with 10% buffer/data ratio

Database Size(GB)	0.5	1	2	4
LRU	50.8	46.1	41.0	51.8
CFLRU	47.6	42.8	37.6	45.8
LRUWSR	48.7	43.9	38.7	48.0
LIRS	47.9	42.7	37.4	46.3
CCFLRU	49.5	44.4	39.1	44.8
CASA	49.4	45.4	40.4	48.7
FOR^+	45.0	39.8	35.0	42.7

Finally, we examine the average processing time for each operation as the computational overhead of buffer management. Table 8 shows the average processing time per operation when the buffer size is 128 pages. The results are similar for other buffer sizes. Since FOR^+ records more information and adopts additional lists compared with other

Table 8: Execution time(μs) for per operation

	TATP	TPC-B	TPC-C
LRU	3.9	3.9	4.3
CFLRU	4.1	4.0	4.4
LRUWSR	4.0	3.9	4.5
LIRS	4.5	4.1	4.7
CCFLRU	4.2	4.0	4.4
CASA	4.1	4.2	4.3
FOR^+	4.8	4.4	5.2

strategies, it has the largest average processing time per operation. Nevertheless, the time difference is less than 1 μs, which has negligible impact on the overall performance (compared with the I/O costs, shown in Figure 8).

6. CONCLUSION AND FUTURE WORK

The inherent features of flash memory pose great challenges for buffer management in flash-based systems. In this paper, we have presented a novel buffer replacement method named FOR to deal with the read/write asymmetry of flash memory. The proposed FOR algorithm exploits the read/write characteristics of flash memory and the operation-wise statistical information to determine the weight of buffered pages. Due to the computational complexity of FOR algorithm, we have developed an approximated version FOR^+ with novel data structure to reduce the computational complexity while preserving the key concepts in the weight determination of FOR. The experimental study on benchmark and synthetic traces demonstrates up to 20% improvements over some state-of-the-art flash-based buffer replacement strategies. As for future work, we plan to study write clustering on FOR^+, and to integrate the FOR^+ into a DBMS.

7. ACKNOWLEDGMENTS

This research was supported by the grants of Natural Science Foundation of China (No. 60873063) and MIIT grant 2010ZX01042-001-001-04.

8. REFERENCES

[1] http://samsung.com/global/business/semiconductor.
[2] http://uflip.inria.fr/ uFLIP.
[3] *Telecom Application Transaction Processing Benchmark*. http://tatpbenchmark.sourceforge.net/index.html.
[4] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An optimized index structure for flash devices. *PVLDB*, 2(1):361–372, 2009.
[5] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding flash io patterns. In *CIDR*, 2009.
[6] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS/Performance*, pages 181–192, 2009.
[7] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4), 2008.
[8] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, pages 31–42, 2002.
[9] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. FAB: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 7 2006.
[10] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.
[11] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha. LRU-WSR: Integration of lru and writes sequence reordering for flash memory. *IEEE Transactions on Consumer Electronics*, 54(3):1215–1223, 10 2008.
[12] H. Jung, K. Yoon, H. Shim, S. Park, S. Kang, and J. Cha. LIRS-WSR: Integration of lirs and writes sequence reordering for flash memory. In *ICCSA (1)*, pages 224–237, 2007.
[13] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In M. Baker and E. Riedel, editors, *FAST*, pages 239–252. USENIX, 2008.
[14] I. Koltsidas and S. Viglas. Flashing up the storage layer. *PVLDB*, 1(1):514–525, 2008.
[15] 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*, 50(12):1352–1361, 2001.
[16] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD Conference*, pages 55–66, 2007.
[17] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD Conference*, pages 1075–1086, 2008.
[18] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *ICDE*, pages 1303–1306, 2009.
[19] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue. CCF-LRU: A new buffer replacement algorithm for flash memory. *IEEE Transactions on Consumer Electronics*, 55(3):1351–1359.
[20] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *ACM European conference on Computer systems*, 2009.
[21] S. T. On, Y. Li, B. He, M. Wu, Q. Luo, and J. Xu. FD-Buffer: A buffer manager for databases on flash disks. In *CIKM (short paper)*, 2010.
[22] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD Conference*, pages 297–306, 1993.
[23] Y. Ou and T. Härder. Clean first or dirty first?: a cost-aware self-adaptive buffer replacement policy. In *IDEAS*, pages 7–14, 2010.
[24] Y. Ou, T. Härder, and P. Jin. CFDC: a flash-aware replacement policy for database buffer management. In *DaMoN*, pages 15–20, 2009.
[25] S.-Y. Park, D. Jung, J.-U. Kang, J. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *CASES*, pages 234–241, 2006.
[26] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS*, pages 134–142, 1990.
[27] D. Seo and D. Shin. Recently-evicted-first buffer replacement policy for flash storage devices. *IEEE Transactions on Consumer Electronics*, 54(3):1228–1235, 10 2008.
[28] L. B. Sokolinsky. LFU-K: An effective buffer management replacement algorithm. In *DASFAA*, pages 670–681, 2004.
[29] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD conference*, 2009.
[30] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. MicroHash: An efficient index structure for flash-based sensor devices. In *FAST*, 2005.