FD-Buffer: A Buffer Manager for Databases on Flash Disks

Sai Tung On
Hong Kong Baptist University

Yinan Li
University of Wisconsin-Madison

Bingsheng He
Nanyang Technological University

Ming Wu
Microsoft Research Asia

Qiong Luo
Hong Kong Univ. of Science and Technology

Jianliang Xu
Hong Kong Baptist University

ABSTRACT

We design and implement FD-Buffer, a buffer manager for database systems running on flash-based disks. Unlike magnetic disks, flash media has an inherent read-write asymmetry: writes involve expensive erase operations and as a result are usually much slower than reads. Therefore, we address this asymmetry in FD-Buffer.

Specifically, we use the average I/O cost per page access as opposed to the traditional miss rate as the performance metric for a buffer. We develop a new replacement policy in which we separate clean and dirty pages into two pools. The size ratio of the two pools is automatically adapted according to the read-write asymmetry and the runtime workload. We evaluate FD-Buffer with trace-driven simulations as well as experiments on real flash disks. Our evaluation results show that our algorithm achieves up to 40% less I/O cost in simulation and up to 33% improvement on the overall performance on commodity flash disks, in comparison with the state-of-the-art flash-aware replacement policy.

1. INTRODUCTION

Flash disks have been widely used for mobile devices, embedded systems, and server platforms (in the form of Solid State Drives, SSDs) [10]. Compared with hard disks, flash disks have a myriad of advantages: high random read performance, high reliability, low power consumption, and so on. Moreover, they are expected to have a sharp increase in the market. IDC [11] predicted that the total flash disk volume will increase by 54.8% per year from 2008 to 2012. Gray also visioned that flash will be in place of hard disks, and pointed out locality in the main memory will play the key role in the overall performance [9]. Since buffer manager is the primary component for capturing memory locality in database systems, this paper studies the design of a buffer manager for database systems running on flash disks.

Not all page accesses in a database system go to the disk. The buffer pool keeps a set of recently accessed pages, and thus filters some of the page access requests before they go to the disk. The buffer management policy influences the sequence of requests that access the disk. Traditionally, it is assumed that the costs for a page read and a page write are uniform (which is mostly true for hard disks). Thus, the buffer miss rate, i.e., the ratio of page accesses that need to go to the disk, is the main metric for evaluating the effectiveness of a buffer manager, since optimizing this metric is equivalent to optimizing the I/O performance. As a result, the goal of traditional buffer management has been to minimize the buffer miss rate. For example, the (off-line) miss-rate-optimal buffer replacement policy (called Belady's algorithm [4]) selects the victim page that will not be needed for the longest time in the future.

CIKM'10, October 25–29, 2010, Toronto, Ontario, Canada.
Copyright 2010 ACM 978-1-4503-0099-5/10/10 ...$10.00.

Flash disks	Random reads	Random writes
Intel X25-M SSD	4260	730
Mtron MSD-SATA3035 SSD	5673	143
Kingston SD4/8GB	538	4.5

Table 1: Performance comparison of random accesses on current flash disks (page size: 8KB, unit: IOPS)

However, the uniform read-write cost assumption does not hold any longer on flash disks. Flash disks have an inherent feature of *read-write asymmetry*: their random write performance is much lower than their random read performance due to the erase-before-write limitation. As shown in Table 1, random writes can be over an order of magnitude slower than random reads on current flash disks. Even worse, a recent study [6] showed that this gap would further increase 3.5-10X after the flash storage is fragmented.

This read-write asymmetry implies that evicting a dirty page has a much higher cost than evicting a clean page, which affects the design of buffer management policies in two fundamental aspects. First, this results in inconsistency between minimizing the buffer miss rate and optimizing the I/O performance: a lower miss rate does not necessarily bring a higher I/O performance. As such, it breaks the premise of minimizing the buffer miss rate. Second, the buffer design needs to be aware of the interplay between the read-write asymmetry and workload characteristics such as the ratio of writes and their access locality. In some cases such as reads being dominant, the buffer may evict dirty pages to make room for reads. In other cases, it is desirable to reduce the number of random writes, without significantly increasing the number of misses to other pages. For example, if the buffer replacement policy can selectively keep some dirty pages with high update frequencies, the overhead of writing these pages is significantly reduced.

Therefore, as opposed to the traditional use of miss rate as the performance metric, we use the average I/O cost per buffer page access to measure the effectiveness of the buffer management. We note that under asymmetric read-write performance, Belady's algorithm is no longer cost-optimal (see Section 3 for details). In fact, the average I/O cost is influenced by many factors such as the flash disk characteristics and read/write patterns in the workload.

To reduce the I/O cost, we propose *FD-Buffer*, a buffer management algorithm for flash disks. FD-Buffer divides the buffer pool into two parts: one for clean pages and the other for dirty pages.

Unlike the existing policies such as CFLRU [28] and CFDC [27] that highly depend on a specific existing replacement policy, each pool of FD-Buffer is managed by an independent policy. This flexibility enables FD-Buffer to integrate various traditional buffer management policies with little modification. The size ratio of the two pools is dynamically adjusted based on the flash disk characteristics and the runtime workload. A cost model is developed to determine the optimal ratio, based on a stack-based model of predicting the buffer miss rate of each pool.

We evaluate our algorithms with two complementary approaches: trace-driven simulations and experiments on real flash disks. The workloads contain publicly available benchmarks on transactional processing such as TPC-C [31], as well as our synthetic workloads. The results validate the accuracy of the cost model and the effectiveness of FD-Buffer. It is shown that FD-Buffer achieves lower I/O costs than LRU (Least Recently Used) with an improvement of up to 21% and 50% on benchmarks and synthetic workloads, respectively. It outperforms CFLRU [28] and CFDC [27], a recent flash-aware algorithm, with an improvement of 17% and 40% on benchmarks and synthetic workloads, respectively. Moreover, FD-Buffer adapts well to the dynamics from workloads and flash disk fragmentation. Experiments based on real flash disks show that FD-Buffer is 70% and 33% faster than LRU and CFDC, respectively.

The rest of this paper proceeds as follows. The next section briefly describes the background and related work, followed by the problem definition in Section 3. Section 4 presents the FD-Buffer algorithm. We present our evaluation results in Section 5, and conclude this paper in Section 6.

2. PRELIMINARIES AND RELATED WORK

In this section, we first give a brief introduction to flash disks and flash-based databases. Next, we review the related work on buffer management policies, followed by the buffer miss rate estimation.

2.1 Flash disks

Flash disks are non-volatile storage with unique characteristics. Both reads and writes of flash disks are at the granularity of flash pages. A typical size of flash page is between 512B to 2KB. Due to the physical characteristics of flash memory, writes are only able to change bits from 1 to 0. Thus, an erase operation that sets all bits to 1 must be performed before rewriting. However, the unit of erase operations is *block*, which typically contains 16-64 pages. Moreover, the latency of an erase operation is far higher than a read or a write. As a result, this *erase-before-write* limitation leads to inferior write, especially random write performance and hence the read-write asymmetry for flash disks. In addition, each flash block can only be erased by a finite number of times before wearing out.

To emulate a traditional hard disk interface that has no erase operations, flash disks employ a firmware layer, called the *flash translation layer* (FTL), to implement page mapping for out-place updates, *garbage collection* and *wear leveling*.

The internal structure of flash disks has been well studied [2]. More recently, a performance study was conducted to analyze the system issues about flash disks [6]; and the uFLIP benchmark [5] was proposed to understand the performance characteristics of flash disks by evaluating the spatial and temporal correlations of flash I/O patterns.

2.2 Flash-based Databases

Performance studies on flash-based databases have been conducted in recent years. Lee et al. [17] show that several components and operations of databases, such as logging, MVCC, and merge joins, are naturally suitable for the I/O characteristics of flash disks. Additionally, basic database constructs such as indexing and joins have been studied on flash memories [24, 20]. Both studies conclude that using SSDs provides better performance than using magnetic hard disks for database applications.

On the other hand, the poor performance of random writes on flash disks has received much attention from the database research community. Specialized data structures and algorithms have been designed to address this issue. Lee et al. [16] proposed the In-Page Logging (IPL) to improve the update performance. Different from the log file system, IPL appends the update logs into a special page that is placed in the same erase block as the updated data pages in order to minimize erase operations. Flash-aware tree indexes [18, 19, 1] have been proposed to address the read-write asymmetry with lazy and batched updates. Tsirogiannis et al. [32] demonstrated that the column-based layout within a page can leverage fast random reads of flash disks to speed up different query operators. Chen [7] proposed a synchronous logging solution by exploiting the fast sequential writes of flash devices. Different from the above studies that developed flash-optimized data structures and algorithms for query processing, this paper investigates flash-optimized buffer management for expediting query processing in a database system.

2.3 Buffer Management Policies

Buffer management is an active research area in databases and operating systems. As mentioned earlier, most algorithms have been focused on minimizing the buffer miss rate. The theoretically miss-rate-optimal replacement policy, known as Belady's algorithm [4], is to evict the page whose next use will occur farthest in the future. Belady's algorithm is an off-line algorithm that provides a lower bound for the miss rate that any online policy could achieve. The policy most widely used by commercial systems is LRU and its variants [13, 26]. LRU always evicts the least recently used page. 2Q [13] is a clock-based approximation of LRU, supporting higher concurrency. LRU-K [26] keeps track of the times of the last K references for each page. It achieves a lower miss rate in database systems by distinguishing frequent pages from infrequent ones.

Several previous studies have explored to partition the buffer pool into two or multiple separate regions for different purposes. DBMIN [8] was proposed to allocate a separate buffer pool for each query. In order to capture both recency and frequency, ARC [23] and its clock-based approximation CAR [3] divide the buffer pool into two parts: one region contains frequent pages, the other contains recent pages. Different from these previous studies, we divide the buffer pool into clean and dirty regions in order to optimize performance for the asymmetric read-write speeds of flash disks.

Recently, there have been several proposals of buffer management policies to address the read-write asymmetry of flash disks [28, 27]. FAB [12] and BPLRU [14] are two erase block-level buffer management strategies. They are both designed for the small buffer on embedded flash devices. The techniques used in these proposals can be categorized into two: giving priority to choosing clean pages as victims over dirty pages [28, 27, 21], and improving the locality of writes [27, 14]. The first category of techniques is more relevant to our study. CCF-LRU [21] considers the access frequency for clean pages in their clean-page first policy.CFLRU (Clean-First LRU) [28] maintains the LRU list into two regions, namely *working region* and *clean-first region*. The working region consists of the most recently used pages that are placed at the header of the LRU list, and the clean-first region is at the tail of the LRU list. Victims are identified in the following preference: firstly clean pages in the clean-first region, then dirty pages in the clean-first region and finally the working region. The working region size is a parameter in CFLRU. Based on CFLRU, CFDC [27] further splits the clean-first region into a clean queue and a dirty queue, and avoids scanning extra dirty pages in the clean-first region of CFLRU. In both studies, compared with dirty pages, clean pages are always given a higher priority for replacement. In contrast, the cost-based approach proposed in this paper adaptively determines their priority based on the running-time workload. Moreover, both of [28, 27] are designed for specific algorithms such as LRU.

Write clustering has been considered an effective technique in reducing the total cost [27, 29]. It combines multiple writes with locality into a single write request to the flash disk. CFDC [27] further enhances CFLRU by clustering dirty pages and evicting them as a batch. Similar techniques are used in recently-evicted-first algorithm [29].

2.4 Buffer Miss Rate Estimation

Mattson et al. [22] proposed a stack-based algorithm to estimate the buffer miss rate for LRU and its variants on any buffer size. The algorithm is done with a single-pass scan on the page references. Kim et al. [15] reduces the computational overhead of the stack-based algorithm. In the following, we briefly describe the stack-based algorithm with LRU as an example. This algorithm will later be extended to estimate the cost of two-pool management in our proposed buffer replacement algorithm (Section 4).

Mattson's algorithm is based on the *inclusion property* of buffers: for any sequence of memory accesses, the contents of a buffer of size k (pages) should be a subset of the contents of a buffer of size $k+1$ or larger. To estimate the buffer miss rate of LRU, Mattson's algorithm uses an LRU stack to store the accessed pages so that the most recently accessed page is on the top of the stack. The algorithm maintains the hit counters, $Hit[1, ..., \infty]$, for calculating the number of buffer misses.

Upon each page access, the algorithm searches the pages in the stack. If the accessed page is in the stack as the i^{th} element from the top, the *stack distance* of the current page access is i. Otherwise, its distance is ∞. The algorithm then moves the accessed page to the top of the stack, and updates the hit counter corresponding to the stack distance. That is, for a page access with stack distance i, $Hit[i]$ is increased by one. This reflects the fact that if the buffer sizes are between 1 and $i-1$ pages, this access will incur a page miss. For buffer sizes larger than or equal to i, the access will result in a page hit. Finally, after processing the entire access sequence which accesses N distinct pages, the miss rate for the buffer with M pages, $P(M)$, is given by Eq. (1):

$$P(M) = 1 - \frac{\sum_{i=1}^{M} Hit[i]}{\sum_{i=1}^{N} Hit[i] + Hit[\infty]}. \quad (1)$$

3. PROBLEM DEFINITION

Since the buffer miss rate is not consistent with the I/O performance due to the read-write asymmetry, we use the average I/O cost as the primary metric. The cost model by definition quantifies the costs of evicting a clean page and a dirty page.

We consider the average I/O cost per page access (*the average I/O cost* in short) in the long run after the warm-up (i.e., after the buffer is filled up). This excludes the cost of buffer misses during the warm-up. Thus, the average I/O cost in our model includes two parts: the cost of fetching a page from the flash disk upon a buffer miss, and the cost of writing a page back to the flash disk upon evicting a dirty page. Eq. (2) gives the average I/O cost:

$$Cost_{io} = P_{total} \cdot C_{read} + P_{total} \cdot E_{dirty} \cdot C_{write}, \quad (2)$$

where P_{total} is the buffer miss rate, E_{dirty} is the ratio of evicting a dirty page in all the evictions, and C_{read} and C_{write} are the costs for reading and writing a page from the flash disk, respectively. For simplicity, we assume each read operation on the flash disk has a cost of C_{read}, and each write is a random write with a cost of C_{write}. While this assumption does not take the access patterns of reads and writes into account, our experiments on real SSDs justify the effectiveness of this simple cost model (see Section 5 for more details).

	Belady [4]			Alternative (FD-Buffer)		
Request	Buffer	Hit?	I/O operation	Buffer	Hit?	I/O operation
-	$[X, Y]$	-	-	$[X, Y]$	-	-
W_A	$[A, X]$	Miss	Read A	$[A, X]$	Miss	Read A
W_B	$[A, B]$	Miss	Read B	$[B, X]$	Miss	Read B, Write A
R_C	$[B, C]$	Miss	Read C, Write A	$[B, C]$	Miss	Read C
R_D	$[C, D]$	Miss	Read D, Write B	$[B, D]$	Miss	Read D
R_C	$[C, D]$	Hit	–	$[B, C]$	Miss	Read C
R_D	$[C, D]$	Hit	–	$[B, D]$	Miss	Read D
R_C	$[C, D]$	Hit	–	$[B, C]$	Miss	Read C
W_B	$[B, C]$	Miss	Read B	$[B, D]$	Hit	–
R_A	$[A, B]$	Miss	Read A	$[B, A]$	Miss	Read A
Summary	–	6 misses, 3 hits	6 reads, 2 writes	–	8 misses, 1 hit	8 reads, 1 write

Table 2: Examples of Belady's algorithm and FD-Buffer

To quantify the read-write asymmetry of the flash disk, we further define the asymmetry factor $\mathbb{R} = \frac{C_{write}}{C_{read}}$. Normalizing Eq. (2) by C_{read}, we obtain the *normalized average I/O cost* in Eq. (3):

$$Cost'_{io} = P_{total} \cdot (1 + E_{dirty} \cdot \mathbb{R}) \quad (3)$$

This cost model is general on capturing both read-write symmetric devices such as hard disks (where $\mathbb{R} = 1$) and read-write asymmetric devices such as flash disks (where $\mathbb{R} > 1$). In the rest of the paper, we assume $\mathbb{R} \geq 1$ to model both hard disks and flash disks.

Given an I/O request sequence S, a buffer management algorithm A, a buffer with M pages, the asymmetry factor $\mathbb{R}$, we denote the normalized average I/O cost of A by $Cost'_{io}(A(S, M, \mathbb{R}))$. Our definition of the cost-optimal buffer management algorithm is as follows:

Definition A buffer management algorithm A is cost-optimal *iff* for any other algorithm A', and for any S, M and $\mathbb{R}$ values, $Cost'_{io}(A(S, M, \mathbb{R})) \leq Cost'_{io}(A'(S, M, \mathbb{R}))$.

This definition has two implications on the design of buffer management. First, the traditional miss-rate-optimized replacement policies are no longer cost-optimal. Table 2 gives an example to show that Beledy's algorithm [4] is not cost-optimal. In this example, the buffer can accommodate two pages. The reference list consists of nine page access requests. The working set contains four pages, which is larger than the buffer size. Initially, after warm-up, the buffer contains two clean pages X and Y. We compare the normalized average I/O costs of Belady's algorithm and an alternative algorithm (our FD-Buffer algorithm in Section 4). According to Eq. (3), they are $\frac{6}{9} \cdot (1 + \frac{2}{6}\mathbb{R})$ and $\frac{8}{9} \cdot (1 + \frac{1}{8}\mathbb{R})$, respectively. The $\mathbb{R}$ value determines which algorithm is better. If $\mathbb{R} < 2$, Beledy's algorithm wins. They have the same cost when $\mathbb{R} = 2$. If $\mathbb{R} > 2$, the alternative algorithm wins, and the performance gap would enlarge with increasing $\mathbb{R}$.

Second, this definition suggests some guideline in the design of a cost-optimal buffer management algorithm for flash disks. That is, an ideal buffer management algorithm should minimize both P_{total} and E_{dirty}. In a fixed-size buffer, these two sub-goals may conflict on certain workloads. For example, we need to put more buffer space for dirty pages, in order to reduce E_{dirty}. But this will increase the buffer miss rate, when the dirty pages have a lower degree of locality than the clean pages. Nevertheless, since reads of flash disks are much faster than writes, it might be still beneficial to reduce E_{dirty} even at the cost of increased buffer miss rate. The key issue is how to achieve a balance between these two sub-goals such that the overall I/O performance is optimized.

4. FD-BUFFER

Our cost definition clearly indicates two design points for an asymmetry-aware algorithm: (1) distinguish clean and dirty pages, and (2) compare the locality of the two kinds of pages to make the replacement decision. Based on these two points, we develop *FD-Buffer*, a unified buffer management system that attempts to minimize the average I/O cost on flash disks.

Without the knowledge of future references, FD-Buffer follows the two design points closely. First, we divide the buffer pool into two sub-pools, the *clean* pool for clean pages and the *dirty* pool for dirty pages. The two pools are independent from each other, with each being managed by a traditional buffer management policy to adapt to the locality. This design allows us to utilize previous research results on traditional buffer management policies with each sub-pool. Second, the global localities of reads and writes on the entire buffer pool are affected by the relative size of the clean pool and the dirty pool. We *dynamically* adjust the size ratio of the two sub-pools by comparing the localities of the two sub-pools. Since the total size of the buffer pool is fixed, increasing the size of one sub-pool will reduce misses on it but increase misses on the other sub-pool.

4.1 Overview

FD-Buffer has three main components, including *buffer manager*, *cost estimator* and *policy advisor*.

The buffer manager has the following four parameters $< M$, M_c, $Policy_c$, $Policy_d >$: the total buffer pool size is M pages, the size threshold of the clean pool is M_c pages, and the replacement policies are $Policy_c$ and $Policy_d$ on the clean and the dirty pool, respectively. The threshold for the dirty buffer size is $M_d = M - M_c$. In FD-buffer, M_c is the key parameter for adaptation to the flash disk characteristics and the workload.

The cost estimator is responsible for estimating the cost for different M_c values, with runtime statistics and the flash disk profile as input. The main statistics include the counters in stack-based cost estimation. We use a light-weight method to collect these statistics (Section 4.3). The profile for the flash disk includes the average latency for reads and writes, and the asymmetry factor, which are obtained through measurements at runtime. The reason for runtime measurement as opposed to offline calibration is that these characteristics of the flash disk may change dynamically over time [6].

The policy advisor is used to adaptively recommend the optimal setting to the buffer manager. It determines the optimal setting through choosing the clean pool ratio that minimizes the I/O cost for FD-Buffer.

4.2 Replacement Algorithms in FD-Buffer

Each pool has an independent replacement policy. In principle, we can use any replacement policy. In this study, we use LRU and its variants for two reasons. First, they are widely used and evaluated in the traditional buffer management. Second, their miss rate predictions have been studied for long, which we can leverage for cost estimation.

Our FD-buffer algorithm uses APIs including *Lookup*, *Update*, *Add*, *Remove*, and *GetVictim*, as commonly used in other buffer algorithms. *Lookup* is to locate a page in the pool and return the frame that contains the page. *Update* is to update the book-keeping data structure to record that the page is referenced. *Add* is to add a page frame to the pool, *Remove* is to remove a page frame from the pool, and *GetVictim* is to get a victim frame for replacement. Take LRU as an example. LRU maintains the metadata of all the page frames in a queue according to their access recency, where the head is the least recently used page frame. Initially, the queue consists of metadata of all unused page frames. *Add* is to add the metadata of a new page frame to the tail of the queue. *Remove* is to remove the metadata of the page frame from the queue. *Lookup* is to locate a page frame in the queue. *Update* is to move the metadata of the page to the tail of the queue. *GetVictim* is to select the page frame at the head of the queue as the victim.

Algorithm 1 Reads and Writes on FD-Buffer.

```
Procedure: Read (Page $p$)
 1: Frame $v = C.Lookup(p)$;
 2: if $v \neq NULL$ then
 3:     $C.Update(v)$;
 4: else
 5:     $v = D.Lookup(p)$;
 6:     if $v \neq NULL$ then
 7:         $D.Update(v)$;
 8:     else
 9:         $v$=FindVictimForClean ($C$, $D$).  // Algorithm 2.
10:         Fetch page $p$ from the disk to frame $v$;
11: Return frame $v$;
Procedure: Write (Page $p$)
 1: Frame $v = C.Lookup(p)$;
 2: if $v \neq NULL$ then
 3:     $C.Remove(v)$;
 4:     $D.Add(v)$;
 5: else
 6:     $v = D.Lookup(p)$;
 7:     if $v \neq NULL$ then
 8:         $D.Update(v)$;
 9:     else
10:         $v$=FindVictimForDirty ($C$, $D$).  // Algorithm 2.
11:         Fetch page $p$ from the disk to frame $v$;
12: Return frame $v$;
```

Algorithm 1 illustrates our FD-Buffer algorithm. The entire buffer pool is divided into the clean pool, C, and the dirty pool, D. Maintaining the locality of each individual pool is the responsibility of each replacement policy, whereas FD-buffer is in charge of adjusting the sizes of both pools. We denote the number of frames in the clean and the dirty pool to be $|C|$ and $|D|$, respectively.

The sizes of the two pools, $|C|$ and $|D|$, are dynamically adjusted along with the reads and writes in FD-buffer. Specifically, *FindVictimForClean* and *FindVictimForDirty* select the victim page from the clean or dirty pool by comparing the number of clean pages and its threshold (Algorithm 2). When the sizes of the two pools are adjusted, frames move between the two. A frame moves from the clean pool to the dirty pool in two scenarios: (1) writing a page in the clean pool, or (2) writing a page to a frame previously occupied by a clean page. In contrast, a frame moves from the dirty pool to the clean pool when reading a page to a frame originally occupied by a dirty page. With this policy, $|C|$ and $|D|$ are dynamically approaching M_c and M_d respectively.

An example of running FD-Buffer is illustrated in Table 2. The threshold size for the clean and dirty pools is one page each. Both pools are managed by LRU. We represent the buffer with a tuple $[D, C]$, with the first frame belonging to the dirty pool, and the second frame to the clean pool. Initially, D is empty and C contains two clean pages. As W_A comes, D grows to one page. The dirty page B stays in the dirty pool for its next write, since no other writes occur during the period. In this example, FD-Buffer can achieve a lower cost than Belady's algorithm.

Algorithm 2 Page evictions in FD-Buffer.

```
Procedure: FindVictimForClean($C$, $D$)
1: if $|D| > M_d$ then
2:     $v = D.GetVictim()$;
3:     $D.Remove(v)$;
4:     $C.Add(v)$;
5:     Write the page in frame $v$ to the disk;
6: else
7:     $v = C.GetVictim()$;
8: Return frame $v$;
Procedure: FindVictimForDirty($C$, $D$)
1: if $|C| > M_c$ then
2:     $v = C.GetVictim()$;
3:     $C.Remove(v)$;
4:     $D.Add(v)$;
5: else
6:     $v = D.GetVictim()$;
7:     Write the page in frame $v$ to the disk;
8: Return frame $v$;
```

We analyze the average I/O cost of FD-Buffer based on the miss rate. A buffer miss occurs in FD-Buffer when the miss occurs in both the clean pool and the dirty pool. Given the miss rate of the clean and the dirty pools, P_c and P_d, respectively, we can derive $P_{total} = P_c \times P_d$.

A dirty page is evicted when a write causes a miss in the dirty pool. We denote this miss rate to be P_d^w. Thus, $E_{dirty} = P_d^w$. Substituting P_{total} and E_{dirty}, we have the normalized average I/O cost for FD-Buffer as expressed in Eq. (4).

$$Cost'_{io} = P_c \times P_d(1 + P_d^w \cdot \mathbb{R}) \quad (4)$$

4.3 Cost Estimation

In order to adapt the clean pool size to the flash disk characteristics and the workload, we need to formulate how P_c, P_d, and P_d^w are influenced by different M_c and M_d values. This leads us for online estimation on P_c, P_d, and P_d^w.

We estimate these miss rates using Mattson's stack-based algorithm [22]. Since the traditional Mattson's stack algorithm only works on computing the miss rate of the entire buffer, we extend it to working on FD-Buffer with two pools. The extension is a non-trivial task, since these two pools interact with each other. For example, some page may exist in different pools at different periods of time and the page miss rate of one pool is affected by the size of the other pool.

To address these challenges, we have developed the *Two-Stack* algorithm, and made two major extensions to Mattson's stack-based algorithm [22]. For ease of presentation, we use LRU as an example to illustrate our algorithm.

First, Two-Stack uses two LRU stacks, CLRU and DLRU, for the clean and dirty pools, respectively. Similar to the LRU stack in Mattson's algorithm, the CLRU and DLRU stacks store the accessed pages according to their access recency. According to our FD-Buffer algorithm, the page in CLRU can be moved to DLRU and vice versa. For example, a write access to a page in CLRU may move the page to DLRU, since the page becomes dirty. However, the page movement from DLRU to CLRU needs extra care. Consider a case where a read access comes and the accessed page exists in DLRU but not in CLRU. After the read access, the page may still stay in DLRU if the read access is a hit in the dirty pool. Otherwise, the read access results in a miss in the dirty pool, and the page should be put into CLRU, since the read access will load the page into the clean pool. Whether the access is a hit or a miss depends on the pool size.

Algorithm 3 Handling a read in Two-Stack.

```
Procedure: ReadHandler (Page $p$)
1: if ($p \notin$ CLRU) $\wedge$ ($p \notin$ DLRU) then
2:     $Hit_c^r[\infty]$++, $Hit_d^r[\infty]$++;
3:     Put $p$ on top of CLRU as the entry $e_c$;
4:     $PCond_c[e_c] = true$;
5: if ($p$ in CLRU at entry $e_c$) $\wedge$ ($p \notin$ DLRU) then
6:     Let $d_c$ be the stack distance in CLRU;
7:     Compute $i$ so that if $M_c \geq i$ then $(M_c \geq d_c) \wedge PCond_c[e_c]$;
8:     $Hit_c^r[i]$++, $Hit_d^r[\infty]$++;
9:     Move entry $e_c$ to the top of CLRU as entry $e'_c$;
10:    $PCond_c[e'_c] = true$;
11: if ($p \notin$ CLRU) $\wedge$ ($p$ in DLRU at entry $e_d$) then
12:    Let $d_d$ be the stack distance in DLRU;
13:    $Hit_d^r[d_d]$++, $Hit_c^r[\infty]$++;
14:    Move entry $e_d$ to the top of DLRU as entry $e'_d$;
15:    $PCond_d[e'_d] = (M_d \geq d_d)$;
16:    Copy entry $e'_d$ to the top of CLRU as entry $e_c$;
17:    $PCond_c[e_c] = (M_c > M - d_d)$;
18: if ($p$ in CLRU at entry $e_c$) $\wedge$ ($p$ in DLRU at entry $e_d$) then
19:    Let $d_c$ and $d_d$ be the stack distances in CLRU and DLRU, respectively.
20:    Compute $i$ so that if $M_c \geq i$ then $(M_c \geq d_c) \wedge PCond_c[e_c]$;
21:    Compute $j$ so that if $M_d \geq j$ then $(M_d \geq d_d) \wedge PCond_d[e_d]$;
22:    $Hit_c^r[i]$++, $Hit_d^r[j]$++;
23:    Move entry $e_c$ to the top of CLRU as entry $e'_c$;
24:    $PCond_c[e'_c] = (PCond_c[e_c] \vee M_c > M - d_d)$;
25:    Move entry $e_d$ to the top of DLRU as entry $e'_d$;
26:    $PCond_d[e'_d] = (PCond_d[e_d] \wedge M_d \geq d_d)$;
```

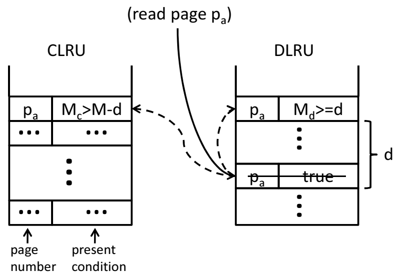


Figure 1: An example of Two-Stack simulation for page read access

The second extension is to handle tricky cases on page movement between two stacks. We store an auxiliary *present condition* for each page in CLRU and DLRU stacks. The present condition states that if this condition is satisfied, the last access to the page makes it not only exist in the stack (either CLRU or DLRU) but also exist in the pool that the stack simulates. It is typically represented by a comparison between the stack distance and the pool size. We use $PCond_c$ and $PCond_d$ to represent the present conditions for pages in clean and dirty pools respectively.

Figure 1 shows an example on page movement from DLRU to CLRU. Assume that the current page read exists only in DLRU and its stack distance is d. If $M_d < d$, the current read access is a miss in the dirty pool, and the page will be moved to the top of CLRU with the present condition $M_d < d$. Since $M_c + M_d = M$, the present condition can also be expressed as $M_c > M - d$. Denoting the top of CLRU to be e, we have $PCond_c[e] = (M_c > M - d)$. If $M_d \geq d$, the current read access is a hit in the dirty pool, and the page is moved to the top of the DLRU as the present condition $M_d \geq d$. Denoting the top of DLRU to be e', we have $PCond_d[e'] = (M_d \geq d)$.

Algorithms 3 and 4 illustrate read and write handling in the Two-Stack algorithm. Two-Stack considers the present conditions on the two stacks. The algorithm maintains hit counters for CLRU

Algorithm 4 Handling a write in Two-Stack.

```
Procedure: WriteHandler (Page $p$)
1: if $p$ in CLRU at entry $e_c$ then
2:     Let $d_c$ be the stack distance in CLRU;
3:     Compute $i$ so that if $M_c \geq i$ then $(M_c \geq d_c) \wedge PCond_c[e_c]$;
4:     $Hit_c^w[i]$++;
5:     Remove entry $e_c$ from CLRU;
6: if $p \notin$ CLRU then
7:     $Hit_c^w[\infty]$++;
8: if $p$ in DLRU at entry $e_d$ then
9:     Let $d_d$ be stack distance in DLRU;
10:    Compute $i$ so that if $M_d \geq i$ then $(M_d \geq d_d) \wedge PCond_d[e_d]$;
11:    $Hit_d^w[i]$++;
12:    Move entry $e_d$ to the top of DLRU as entry $e'_d$;
13:    $PCond_d[e'_d] = true$;
14: if $p \notin$ DLRU then
15:    $Hit_d^w[\infty]$++;
16:    Put $p$ on the top of DLRU as entry $e''_d$;
17:    $PCond_d[e''_d] = true$;
```

and DLRU. It also distinguishes the hit counters for reads or writes, since the computation of P_d^w only depends on write accesses. Let Hit_c^r and Hit_c^w be the hit counters for reads and writes in the clean pool, respectively. Let Hit_d^r and Hit_d^w be the hit counters for reads and writes in the dirty pool, respectively.

According to the definition of hit counters, after processing a sequence of page references, P_c, P_d, and P_d^w are calculated in Eqs. (5)–(7).

$$P_c(M_c) = 1 - \frac{\sum_{i=1}^{M_c}(Hit_c^r[i] + Hit_c^w[i])}{\sum_{i=1}^{n}(Hit_c^r[i] + Hit_c^w[i]) + Hit_c^r[\infty] + Hit_c^w[\infty]} \quad (5)$$

$$P_d(M_d) = 1 - \frac{\sum_{i=1}^{M_d}(Hit_d^r[i] + Hit_d^w[i])}{\sum_{i=1}^{n}(Hit_d^r[i] + Hit_d^w[i]) + Hit_d^r[\infty] + Hit_d^w[\infty]} \quad (6)$$

$$P_d^w(M_d) = 1 - \frac{\sum_{i=1}^{M_d} Hit_d^w[i]}{\sum_{i=1}^{n} Hit_d^w[i] + Hit_d^w[\infty]} \quad (7)$$

4.4 Policy Advisor

The design goal of the policy advisor is to predict the optimal replacement policy for future references. Since workload prediction is difficult and challenging in general, we use a simple window based prediction method, where a window is defined as a predefined number of consecutive page references. Denote the previous window and the current window as Win and Win', respectively. The current policy advisor is to decide the M_c value based on the statistics collected from Win, and use this M_c value for Win'.

Based on the hit counters in Win, we iterate M_c from one to $(M - 1)$, and use Eqs. (5)–(7) to calculate P_c, P_d, and P_d^w values for each M_c value. With P_c, P_d, and P_d^w values, we use Eq. (4) to calculate the normalized average I/O cost for the M_c value. Thus, we obtain the optimal M_c value of Win to be the minimum average I/O cost among the M estimated costs.

The window size is a tuning parameter for the policy advisor. Ideally, it should balance the gain of adaptation on workload changes and the overhead of running the policy advisor. If the window size is too small, we run the policy advisor too often, and the computation overhead of the policy advisor is high. On the other hand, if the window size is too large, the policy may not well adapt to workload changes. Thus, we determine the window size considering the computation overhead of the policy advisor. The basic idea is to limit the overhead to a threshold ratio of q of the total I/O time performed on the window. Suppose the computation time of the cost estimation in the policy advisor is $Comp$, and we set the window

	database size (GB)	#Reference (millions)	Write ratio	Description
TPCC	2.4	16.8	15.0%	20 warehouses
TPCB	2.2	12.7	3.5%	150 branches
TM1	2.4	10.2	4.6%	1 million subscribers
Synthetic	2.0	14.0	1–99%	Reads and writes with Zipf distributions.

Table 3: Specification on the traces in the experiment

size so that the total I/O time reaches $\frac{Comp}{q}$. We have validated the effectiveness of this simple method in our experiments.

5. EVALUATION

In this section, we evaluate our algorithms with trace-driven simulations as well as experiments on real flash disks.

5.1 Experimental Setup

We ran our experiments on a Windows workstation with an Intel 2.4GHz quad-core CPU, 4GB main memory, a 160GB 7200rpm SATA magnetic hard disk, an SD card and two flash SSDs. The hard disk is with 109 and 100 random reads and writes on 8KB pages per second, respectively. The SD card is an 8GB Kingston SD4, which represents low-end flash disks. The two SSDs are an Mtron MSD-SATA3035 64GB and an Intel X25-M 80GB, which represent high-end flash disks. They have different characteristics: the Mtron uses block-level mapping in the FTL and contains SLC flash memory, whereas the Intel one uses page-level mapping and consists of MLC flash memory. The asymmetry factor values for the hard disk, Kingston SD card, Intel and Mtron SSDs are 1.1, 121.3, 5.8 and 39.6 respectively.

Workloads. The workloads include publicly available benchmarks, and our synthetic ones (Table 3). We use three benchmarks on transactional processing, namely TM1 [25], TPC-B [30], and TPC-C [31]. TM1 is a telecom workload benchmark, TPC-B simulating transactions on a hypothetical bank, and TPC-C for online transaction processing. To get the traces on buffer page accesses, we run the three benchmarks on PostgreSQL 8.4 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 minutes warm-up period. The number of clients is 20 for all benchmarks. We explicitly configure the buffer size to be smaller than the database size in order to exercise disk I/O operations. Moreover, the trace includes all the accesses to the pages in the buffer, and the logging I/O is not included in the trace.

While public benchmarks approximate workloads from the real world, we do not have the full control of the workload characteristics such as distributions, and the read/write ratio. Therefore, we have generated synthetic traces with page accesses of reads and writes conforming to Zipf distributions. The trace generation follows the setting ($< \alpha_r$, α_w, $w >$), where w is the write ratio in the trace, and α_r and α_w represent the α values in the Zipf distribution for reads and writes, respectively. This allows us to examine the impact of the skewness in reads and writes, which affects the working set size of reads and writes. The larger the α value, the more skewed the Zipf distribution is and the smaller the working set is. If the α value is zero, the distribution is uniform, and the working set size is at maximum.

The synthetic traces have the same number of page accesses as the one generated from TPC-C, using the first $\frac{1}{6}$ of page accesses to warm up the buffer, and the rest for performance evaluation.

In order to evaluate the performance impact of workload dynamics, we simulate the dynamics in the read/write ratio. In particular, we divide the TPC-C trace into epochs, and each epoch consists of around 5 thousand page references. We dynamically changed the write ratio w in these epochs by changing reads to writes in the traces. We use two models for dynamics: (WM_1) $w_i = w_0(1 + i\%)$, and (WM_2) $w_i = 0.95w_0 \cdot (i \bmod 2) + w_0 \cdot ((i+1) \bmod 2)$, where w_i is the write ratio in the $i+1$th epoch of trace, and w_0 is obtained in the first epoch of the original TPC-C trace. The first model simulates the case when write requests are becoming dominant in the workload, and the second model simulates the case where the workload periodically changes.

We have implemented a simulator and a buffer manager running on SSDs and SD cards. Both implementations are driven by traces.

Implementation for simulation. The simulator models the behavior of a buffer manager on flash disks. It takes a trace as input and accumulates the number of hits and misses according to the replacement policy. The output is the average I/O cost on the trace.

In FD-Buffer, each of the two pools is managed by LRU. we evaluate the effectiveness of FD-Buffer in comparison with LRU and CFDC [27], as the representatives for traditional disk-based and flash-based replacement policies, respectively. We use LRU as the base for comparison, since LRU and its variants are the most widely used replacement policies in traditional buffer managers. Since the I/O cost of write clustering depends on complicated hardware configurations such as disk caches and FTL implementations, we do not implement write clustering in simulations, and evaluate the impact of write clustering on the real flash disk only.

CFDC without write clustering is similar to CFLRU [28], with smaller computation overhead achieved by splitting the clean-first region into a clean queue and a dirty queue. Thus, we denote this algorithm with "CFLRU(WRR)", where WRR is the ratio of the working region in the total buffer size. In principle, this ratio could be adapted to the workload and the flash disk. CFLRU [28] designed for virtual memory relies on the operating system specific information to make the adaptation. However, to the best of our knowledge, there is no adaptive tuning on WRR for different workloads and flash disks in database systems. For example, CFDC statically sets this ratio to 0.5. Therefore, we also use the static WRR in our experiments.

Simulations allow us to examine arbitrary values for the asymmetry factor. A large asymmetry factor value may come from two sources – either the inherent and static property of the device or the dynamic fragmentation effect of the device. For instance, a hard disk has an asymmetry factor value of approximately one, and a low-end SSD may have a high asymmetry factor value. Furthermore, due to fragmentation, the asymmetry factor value may significantly increase on the same flash disk over time. In our simulations, the value range for the asymmetry factor is 1–128 with 32 being the default value. When we vary the value of the asymmetry factor, we fix the read cost to be the read latency of the Mtron SSD (0.176 ms in our measurement), and increase the write cost linearly to the asymmetry factor.

To evaluate the performance impact of fragmentation in long running scenarios, we simulate the dynamics in the asymmetry factor in a similar way to workload dynamics. In particular, we use two models for dynamics: (RM_1) $\mathbb{R}_i = \mathbb{R}_0 \cdot (1 + i \times 0.1)$, and (RM_2) $\mathbb{R}_i = 0.95 \cdot \mathbb{R}_0 \cdot (i \bmod 2) + \mathbb{R}_0 \cdot ((i+1) \bmod 2)$, where $\mathbb{R}_i$ is the asymmetry value when executing the $i+1$th epoch of trace, and $\mathbb{R}_0$ is set as 32 by default. The first model simulates the increasing asymmetry factor as the flash disk becomes fragmented, and the second model simulates that the fragmented disk is periodically cleaned (such as with the latest *Trim* command), and is restored to the state with a relatively low asymmetry factor.

Implementation on flash disks. To evaluate FD-Buffer on real flash disks, we implement a buffer manager on top of standard OS file system facilities. The buffer manager takes a trace as input, and performs I/O requests to the flash disk. Thus, we can obtain the average response time for a buffer page request on the flash disk. We also evaluate the write clustering technique for both FD-Buffer and CFDC [27] on flash disks.

	TPC-C		TPC-B		TM1	
	w/ PR	w/o PR	w/ PR	w/o PR	w/ PR	w/o PR
P_c	5.1%	186.9%	18.6%	108.3%	16.6%	59.7%
P_d	7.5%	8.2%	1.7%	2.0%	0.9%	1.0%
P_d^w	4.6%	4.6%	17.5%	19.5%	3.6%	6.6%

Table 4: Miss rate estimation with and without present conditions in FD-Buffer

The indexes and tables for the benchmarks and synthetic workloads are stored in files in the file system. We set the page size to be consistent with that in PostgreSQL ($8K$ bytes).

For all the replacement policies, a page written in the buffer pool is marked as dirty, and will be written to disk when it is evicted. To avoid the interference between the virtual memory of the operating system and our buffer manager, we disabled the buffering functionality of the operating system using Windows APIs.

We have made two optimizations to reduce the overhead of the runtime overhead in FD-Buffer. First, we tune the q value for calculating the window size and determine $q = 3\%$ so that the overhead of cost estimation in the policy advisor does not exceed 3%. Second, in order to improve the efficiency of our stack-based estimation, we have used an LRU stack for a sub-pool, and adopted the group-based optimization in the previous study [15]. The basic idea of the optimization is to parition the pages by their access recency into g groups G_0, G_1, ..., G_{g-1}, with each group consisting of I pages. G_0 consists of the most recently accessed pages. The pages within a group G_k is approximately treated with equal distances $(k \times I)$. Pointers to the pages are maintained in group headers. Upon each page access, only the affected group headers are scanned, instead of the affected pages. The group-based optimization can greatly reduce the overhead with insignificant accuracy loss [15]. In our implementation, we use the group size of 64 pages, and the accuracy loss in the miss rate estimation is less than 5%.

In both simulations and real executions, we examine the impact of buffer pool size. We vary Buffer_DB_Ratio (i.e., the ratio of buffer pool size to the database size) with the range of 0.1%–6.25% with 3.1% as the default value. Thus, our results are presented in the default setting: Buffer_DB_Ratio=3.1% and $\mathbb{R} = 32$, unless specified otherwise.

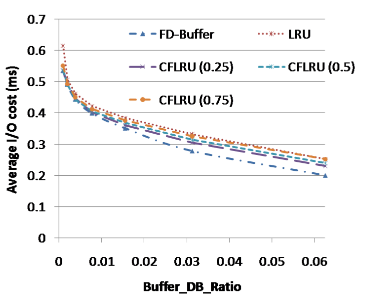
5.2 Simulation Results

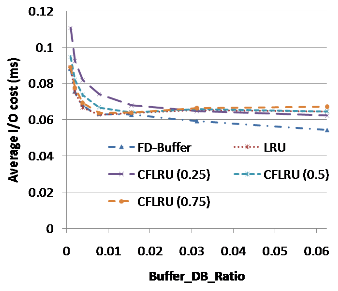
We first evaluate the effectiveness of our adaptation in FD-Buffer. Next, we evaluate FD-Buffer in comparison with LRU and CFLRU.

5.2.1 Results on Benchmarks

Evaluating miss rate estimations. Table 4 shows the average errors between estimations and measurements on P_c, P_d, and P_d^w values for the three benchmarks under default settings. The error is defined to be $e = \frac{|v-v'|}{v} \times 100\%$, where v and v' are the measured and the estimated values, respectively. We compare the estimations with and without present conditions, denoted as "w/ PR" and "w/o PR", respectively. The extension of present conditions significantly reduces the average error. The improvement is the most significant for P_c, since P_c is affected most by the page movement between the two stacks in the Two-Stack algorithm. The average errors with present conditions are less than 20%.

Effectiveness of adaptation. We examine the effectiveness of the adaptation in comparison with the algorithms with static clean


(a) TPC-C


(b) TPC-B

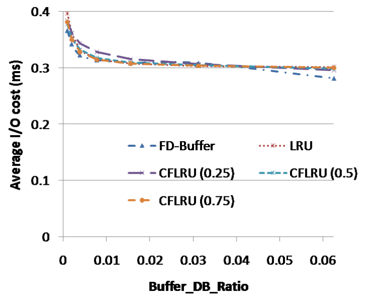

(c) TM1

Figure 3: Average I/O cost of online algorithms on benchmarks.

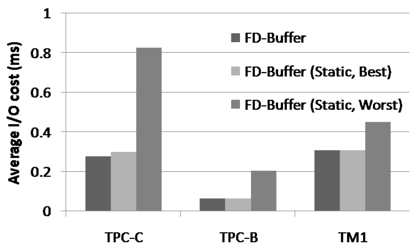


Figure 2: The comparison of FD-Buffer and the best and worst of FD-Buffer with static clean pool sizes.

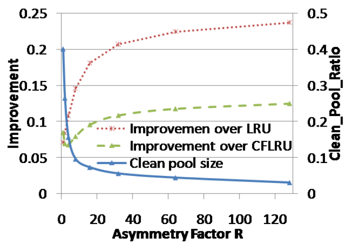
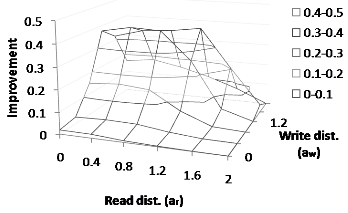


Figure 4: The average I/O cost improvement of FD-Buffer over LRU and CFLRU with $\mathbb{R}$ varied.


(a) Over LRU

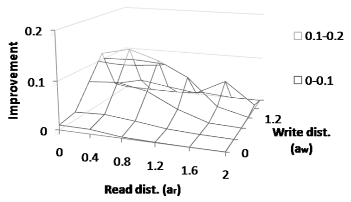

(b) Over CFLRU

Figure 5: The average I/O cost improvement of FD-Buffer over LRU and CFLRU.

pool sizes being $(M-1)$ and 2^i pages $(0 \leq i \leq (\log_2 M-1))$. Figure 2 compares their average I/O costs. FD-Buffer (with adaptation) is comparable to or better than the best of static algorithms. Under the default buffer setting, the average I/O cost of FD-Buffer is 9% less than that of the best static algorithm on TPC-C, and is within 1% greater than those on TPC-B and TM1. The accurate miss rate estimation contributes to the effectiveness of our adaptation. On the other hand, the best static clean pool ratios are different, with 1.5%, 25% and 50% on TPC-C, TPC-B and TM1, respectively. Moreover, the gap between the best and the worst of the static algorithms indicates the significant performance loss with wrong settings.

Overall comparison. Figure 3 shows the average I/O cost of online algorithms on the three benchmarks, with the Buffer_DB_Ratio varied. We also show CFLRU with different WRR values, i.e., 0.25, 0.5 and 0.75. Note, the previous study [27] used 0.5. For all of the three benchmarks, FD-Buffer is the best, CFLRU with a suitable setting is the second, and LRU is the worst. When the ratio increases from 0.1% to 6.25%, the I/O cost of FD-Buffer decreases. The improvement of FD-Buffer over LRU is up to 21%, 15%, 7% on TPC-C, TPC-B and TM1 benchmarks, respectively, and the improvement over CFLRU(0.5) is up to 15%, 17%, and 7%, respectively.

CFLRU can be worse than LRU (up to 18%) under an unsuitable setting. However, the suitable setting for CFLRU varies with buffer sizes and benchmarks. Among the three CFLRU variants, there is not a clear winner for all the benchmarks and buffer sizes. Thus, following the previous study [27], we use 0.5 as the suitable WRR setting for CFLRU, and denote "CFLRU(0.5)" with "CFLRU" for short in the rest of the experiments.

Varying $\mathbb{R}$. Figure 4 shows the average I/O cost improvement of FD-Buffer over LRU and CFLRU on TPC-C with $\mathbb{R}$ varied. As the asymmetry factor value increases, the improvement over LRU increases. The improvement over CFLRU slightly decreases and then increases, between 7% and 13%. The improvement is more significant when $\mathbb{R}$ is small, because reads and writes are at almost the same cost, and the clean-first policy in CFLRU hurts its performance. When $\mathbb{R}$ is large, the improvement of FD-Buffer over CFLRU increases. More pages should be allocated to the dirty ones, but the static working region ratio in CFLRU makes it contain fewer dirty pages than FD-Buffer.

The decreasing clean pool sizes in FD-Buffer reveal the effectiveness of the cost-based adaptation, since writes are becoming more costly and the clean pool shrinks for holding more dirty pages in the buffer. This indicates that our cost-based adaptation is adapted for different flash disks, and also for the changing asymmetry factor on the same flash disk.

5.2.2 *Results on Synthetic Workloads*

Varying read/write distributions. Figure 5 shows the average I/O cost of FD-Buffer over LRU and CFLRU. Under all the combinations, FD-Buffer has a lower average I/O cost than both LRU and CFLRU, with an improvement between 0.1% and 39% over LRU, and between 0.1% and 13% over CFLRU. The trend of the improvement is similar on both LRU and CFLRU. For a fixed α_r, the improvement is a concave curve as α_w increases. The performance difference is due to the difference in the capability of capturing write localities among the algorithms. For a fixed α_w value, the improvement becomes smaller as α_r increases.

We further examine the average size ratio of clean pool in FD-Buffer and find that the clean pool size adapts well to the read/write localities (Figure 6). For a fixed α_r, the clean pool ratio increases as α_w increases, since a larger clean pool size results in a smaller

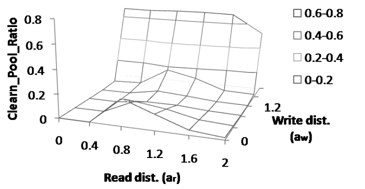
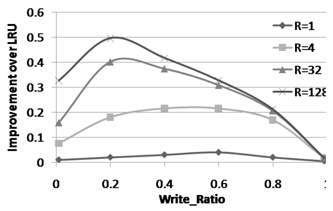
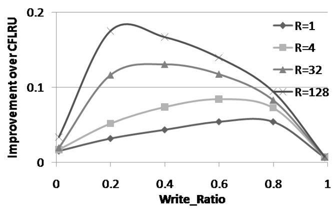


Figure 6: The average ratio of clean pool in FD-Buffer.



(a) Over LRU

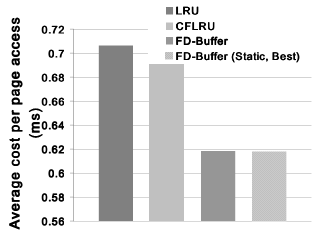


(b) Over CFLRU

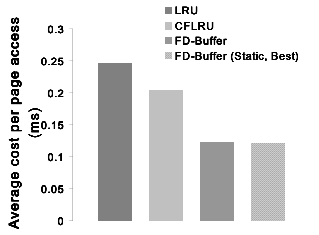
Figure 7: The average I/O cost improvement of FD-Buffer over LRU and CFLRU varying write ratios.

average I/O cost. For a fixed α_w value, the clean pool ratio is a concave curve. When α_r is near zero, the read locality is low. Allocating pages to the clean pool does not significantly reduce the read misses. When α_r is near two, the read locality is so high that a small number of pages is sufficient for achieving a low miss rate. Between these two ends, a reasonable number of pages allocated to the clean pool can minimize the cost.

Varying write ratios. Figure 7 shows the average I/O cost of FD-Buffer over LRU and CFLRU with the write ratio varied. We fix α_r and α_w to 0.4 and 1.2, respectively. Overall, the improvement of FD-Buffer is 0.5–48% over LRU, and 0.5–18% over CFLRU. The improvement is more significant for a high asymmetry factor value. As the write ratio increases, the improvement on a fixed asymmetry factor value is a concave curve. FD-Buffer achieves a higher improvement on workloads with a mix of reads and writes than those with reads or writes only.



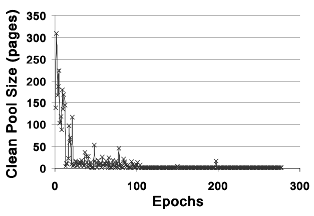
(a) WM_1



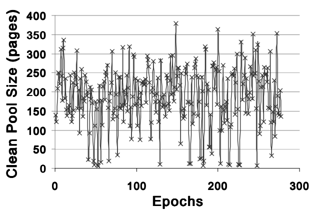
(b) WM_2

Figure 8: Average I/O cost of online algorithms under dynamic workload.

Adaptation to workload dynamics. Figures 8(a) and 8(b) show the average I/O cost of FD-Buffer (with adaptation), FD-Buffer (with a best static clean pool ratio), LRU and CFLRU under the workloads with dynamic models WM_1 and WM_2, respectively. We can make the following observations: 1) the performance of FD-Buffer(with adaptation) is comparable to that of the best static FD-Buffer algorithm, which validates the effectiveness of our adaptation; 2) FD-Buffer(with adaptation) sufficiently outperforms LRU and CFLRU by 12% and 11% on WM_1 respectively, and the improvements increase to 50% and 40% on WM_2. This is because FD-Buffer can efficiently adapt to the workload by adjusting the clean pool size, as shown in Figure 9. For example, if a workload has a higher write ratio, more pages are allocated to hold the dirty pages and hence more I/O cost savings are obtained.

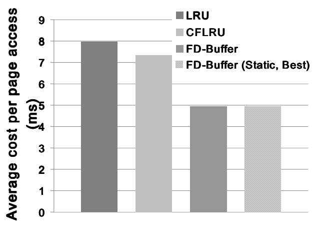


(a) WM_1

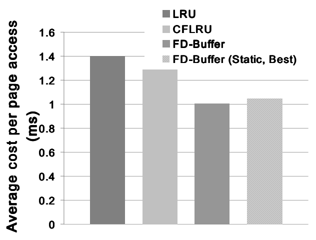


(b) WM_2

Figure 9: The average clean pool size of each epoch in FD-Buffer under dynamic workload.



(a) RM_1



(b) RM_2

Figure 10: Average I/O cost of online algorithms under dynamic asymmetry factor.

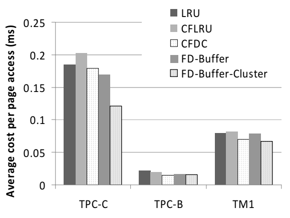
5.2.3 *Results on Asymmetry Factor Dynamics*

Figures 10(a) and 10(b) show the average I/O cost of FD-Buffer(with adaptation), FD-Buffer(with a best static clean pool ratio), LRU and CFLRU under asymmetry factors with dynamic models RM_1 and RM_2, respectively. We can observe that, the performance of FD-Buffer (with adaptation) is comparable to or better than that of the best static FD-Buffer algorithm. Furthermore, it outperforms LRU and CFLRU by 38% and 33% on RM_1, respectively, and the improvements on RM_2 are 28% and 22%, respectively. The clean pool size well adapts to the dynamics in the asymmetry factor, as shown in Figure 11.

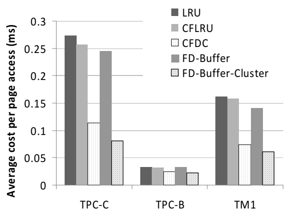
5.3 Results on Real Disks

Figure 12 shows the average time per page access of the replacement policies on the three flash disks and the hard disk. The average time per page access is given by the total execution time after warm-up till the end of the trace divided by the number of buffer page accesses. The performance comparison varies with the asymmetry factor. On the hard disk ($\mathbb{R} = 1.1$), FD-Buffer (without write clustering) has little improvement (less than 1%) over LRU and CFLRU on all three benchmarks. However, on the Intel SSD ($\mathbb{R} = 5.8$), the improvement of FD-Buffer becomes larger, with up to 9% faster than LRU. Such an improvement continues to increase to 13% and 26% on the Mtron SSD ($\mathbb{R} = 39.6$) and the SD card ($\mathbb{R} = 121.3$), respectively. Meanwhile, FD-Buffer outperforms CFLRU by up to 17% on these three flash disks. Two factors contribute to the performance improvement of FD-Buffer. First, the grouping optimization significantly reduces the computation overhead. Second, the reduction in the I/O cost is the major factor for improvement, which has been demonstrated in simulations.

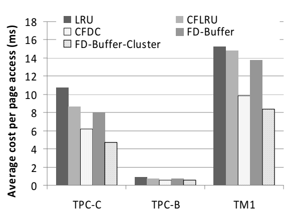
Furthermore, write clustering also helps to improve the performance of FD-Buffer. As can be observed from the experiment results, with writing clustering, FD-Buffer achieves an improvement of 5%∼67% (the improvement varies under different workloads and hardwares), which enables FD-Buffer(with write clustering) to outperform CFDC by up to 33%.



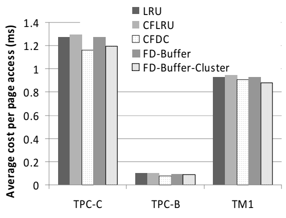
(a) Intel SSD



(b) Mtron SSD

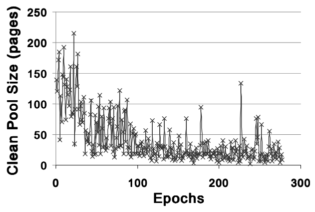


(c) SD card

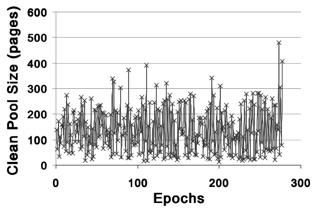


(d) Hard disk

Figure 12: The average time per page access of FD-Buffer on benchmarks with three flash disks and the hard disk.



(a) RM_1



(b) RM_2

Figure 11: The average clean pool size of each epoch in FD-Buffer under dynamic asymmetry factor.

6. CONCLUSIONS

As flash disks have become competitive storage media for database systems and they possess distinct hardware features from traditional hard disks, there is an urgent need to re-visit the database management techniques. This paper studies the buffer management for flash-based databases, with a focus on addressing the read-write asymmetry and workload dynamics. We have developed FD-Buffer that automatically adapts to the flash disk characteristics and the runtime workload. This automacity significantly reduces the ownership cost of database systems running on flash disks. Our trace-driven experimental studies show that FD-Buffer outperforms both LRU and a recent flash-aware algorithm. As for future work, we are interested in further improving the efficiency of FD-Buffer with other traditional replacement polices such as 2Q [13], and with optimizations on write patterns to the flash disk. Moreover, we are integrating our buffer management techniques into open-source DBMSs such as PostgreSQL.

Acknowledgement

The work of Sai Tung On, Yinan Li and Bingsheng He was done when they were in Microsoft Research Asia. The authors would like to thank Ippokratis Pandis for providing his source code on implementing the TM-1 benchmark.

7. REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. In *VLDB*, 2009.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigraphy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, 2008.

[3] S. Bansal and D. S. Modha. Car: Clock with adaptive replacement. In *FAST*, 2004.

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems*, 1966.

[5] L. Bouganim, B. T. Jónsson, and P. Bonnet. uflip: Understanding flash io patterns. In *CIDR*, 2009.

[6] F. Chen, D. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS*, 2009.

[7] S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD Conference*, 2009.

[8] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB*, 1985.

[9] J. Gray. *Tape is Dead Disk is Tape Flash is Disk RAM Locality is King*, 2006.

[10] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *ACM Queue*, 6(4):18–23, 2008.

[11] IDC. *Worldwide Solid State Drive 2008–2012 Forecast and Analysis: Entering the No-Spin Zone*, 2008.

[12] 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.

[13] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, 1994.

[14] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *FAST*, 2008.

[15] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. *SIGMETRICS Perform. Eval. Rev.*, 19(1), 1991.

[16] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD*, 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*, 2008.

[18] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *ICDE*, 2009.

[19] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. In *PVLDB*, 2010.

[20] Y. Li, S. T. On, J. Xu, B. Choi, and H. Hu. Digestjoin: Exploiting fast random reads for flash-based joins. In *MDM*, 2009.

[21] 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), 2009.

[22] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2), 1970.

[23] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, 2003.

[24] D. Myers. On the use of nand flash memory in high-performance relational databases. *MIT Msc Thesis*, 2008.

[25] Nokia. *Network Database Benchmark*. http://hoslab.cs.helsinki.fi/homepages/ndbbenchmark/.

[26] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD*, 1993.

[27] Y. Ou, T. Härder, and P. Jin. CFDC: a flash-aware replacement policy for database buffer management. In *DaMoN*, 2009.

[28] S. Y. Park, D. Jung, J. U. Kang, J. Kim, and J. W. Lee. CFLRU: a replacement algorithm for flash memory. In *CASES*, 2006.

[29] 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.

[30] TPC. *TPC Benchmark B: Standard Specification*. http://www.tpc.org/tpcb/spec/tpcb_current.pdf.

[31] TPC. *TPC Benchmark C: Standard Specification*. http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[32] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*, 2009.