FD-Buffer: A Cost-Based Adaptive Buffer Replacement Algorithm for Flash Memory Devices

Sai Tung On, Shen Gao, Bingsheng He, Ming Wu, Qiong Luo, and Jianliang Xu

Abstract—In this paper, we present a design and implementation of FD-Buffer, a cost-based adaptive buffer manager for flash memory devices. Due to flash memory's unique hardware features, it has an inherent read-write asymmetry: writes involve expensive erase operations, which usually makes them much slower than reads. To address this read-write asymmetry, we revisit buffer management and consider the average I/O cost per page access as the main cost metric, as opposed to the traditional miss rate. While there have been a number of buffer management algorithms that take the read-write asymmetry into consideration, most algorithms fail to effectively adapt to the runtime workload or different degrees of asymmetry. In this paper, we develop a new replacement algorithm in which we separate clean and dirty pages into two pools. The size ratio of the two pools is automatically adapted based on the read-write asymmetry and the runtime workload. We evaluate the FD-Buffer with trace-driven experiments on real flash memory devices. Our trace-driven evaluation results show that our algorithm achieves 4.0-33.4 percent improvement of I/O performance on flash memory, compared to state-of-the-art flash-aware replacement policies.

Index Terms—Flash memory, buffer management, solid-state drives, read-write asymmetry

1 Introduction

Flash memory has been widely used for mobile devices and embedded systems and it has a myriad of advantages: high random read performance, high reliability, low power consumption, and so on. Moreover, flash memory is expected to have a sharp increase in market share. The capacity of flash memory has been increasing while its price per GB has been decreasing significantly [27], thanks to demand in mobile and embedded markets. International Data Corporation (IDC) [12] predicted that the total flash memory volume will increase by 54.8 percent in the coming years. Despite its fast speed, flash memory's locality in the main memory will play a key role in overall performance [9], [11]. Since buffer managers are the primary component for capturing memory locality in database and operating systems, this paper studies the design of a cost-based adaptive buffer manager for flash memory devices.

Not all page accesses in a database/operating 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). However, the uniform read-write cost assumption does not hold for flash memory. One inherent feature of flash memory is a *read-write asymmetry*: their random write performance is much lower than their random read performance because of the erase-before-write limitation. As shown in Table 1, random writes can be over two orders of magnitude slower than random reads in current flash memory. Even worse, a recent study [7] showed that this gap would increase 3.5-10X further after flash storage becomes fragmented.

This read-write asymmetry implies that evicting a dirty page costs much more than evicting a clean page, which fundamentally affects the design of buffer-management policies. This results in inconsistency between minimizing buffer miss rate and optimizing I/O performance: a lower miss rate does not necessarily raise I/O performance. Therefore, it breaks the conventional premise of minimizing the buffer miss rate. Thus, 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 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 memory characteristics and read/write patterns in the workload.

Recently, there has been a number of algorithms proposed to address the read-write asymmetry in buffer management. Those algorithms include FAB [14], CFLRU [37] (and its enhanced variant, CFDC [34]) and CASA [33]. Most of those algorithms give priority to choosing clean pages as victims over dirty pages. However, all of these algorithms

- *S.T. On, S. Gao, and J. Xu are with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, KLN, Hong Kong. E-mail: {ston, sgao, xujl}@comp.hkbu.edu.hk.*
- *B. He is with the School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: bshe@ntu.edu.sg.*
- *M. Wu is with Microsoft Research Asia, Haidian District, Beijing, China.*
- *Q. Luo is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: luo@cse.ust.hk.*

Manuscript received 26 July. 2012; revised 26 Dec. 2012; accepted 24 Feb. 2013. Date of publication 6 Mar. 2013; date of current version 7 Aug. 2014. Recommended for acceptance by E. Macii.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.52

TABLE 1
Performance Comparison of Random Accesses on Current Flash Memory (Page Size: 8 KB, Unit: IOPS)

Flash memory	Random reads	Random writes	Ratio of read/write throughput
Kingston SDHC 8GB	680.4	5.0	136.2
Sandisk Cruzer 8GB	666.1	1.4	475.7

have an ad-hoc way of determining the victim based on heuristics. For example, CFLRU always selects a clean page from the tail of the LRU list (i.e., the clean-first region), and it sets the clean-first region to one half of the entire LRU list. CASA [33] proposed a heuristics-based approach to dynamically adjust the ratio of dirty to clean pages.

In general, the existing algorithms fail to fully adapt to different read-write asymmetry levels and workload characteristics, which makes them inefficient. First, the read-write asymmetry varies significantly among flash memory devices because different devices have different hardware and software implementations. Moreover, the degree of read-write asymmetry increases as the flash memory gets fragmented [7]. 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 read-dominant cases, the buffer may evict dirty pages to make room for reads. In some other cases, it would be 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, it can significantly reduce the overhead of writing these pages.

To address these challenges, we propose *FD-Buffer*, a cost-based adaptive buffer-management algorithm for flash memory devices. FD-Buffer divides the buffer pool into two parts: one for clean pages and one for dirty pages. The size ratio of the two pools is dynamically adjusted based on the flash memory characteristics and the runtime workload. In particular, for LRU replacement policies, a cost model is developed to determine the optimal size ratio based on a stack-based model that predicts the buffer miss rate of each pool. With cost-based adaptation, the choice of whether a victim is a clean page or a dirty page is made according to the quantitative gain of the two pools. Moreover, to reduce the overhead of block erasures, we use a write clustering technique that gives priority to evicting pages that reside in the same cluster. All these techniques give the FD-Buffer good I/O performance on flash memory devices.

We evaluate our algorithm with trace-driven simulations on real flash memory devices. The workloads contain three publicly available benchmarks on transactional processing, including TPC-C [42] and file-system traces. The results validate the accuracy of the cost model and the effectiveness of FD-Buffer. FD-Buffer outperforms CFDC [34] and CASA [33], two representative flash-aware algorithms, with an improvement of 4.0–33.4 percent on various traces. Moreover, FD-Buffer adapts well to the dynamics of workload and flash memory fragmentation.

The contributions of this paper are summarized as follows. First, we propose a simple yet effective online algorithm called FD-Buffer, which adapts to the runtime dynamics of flash memory and workloads. Second, we develop a two-stack method for online cost estimation of the two pools in FD-Buffer. Third, we perform extensive performance evaluation with database benchmarks and file-system traces for our algorithm compared to flash-aware buffer-management policies.

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 offer conclusions in Section 6.

2 Preliminaries and Related Work

In this section, we first give a brief introduction to flash memory and data management on flash-based storage devices. Then we review related work on buffer-management policies, followed by buffer miss-rate estimation.

2.1 Flash Memory

Flash memory has been the dominate media in mobile devices and embedded systems because of its low access latency and power consumption. Recently, many manufacturers have packed flash memory chips into solid-state drives (SSDs) for personal computers and servers.

Flash memory is non-volatile storage with unique characteristics. Both its reads and writes are at the granularity of flash pages. The typical size of a flash page is between 512 B and 2 KB. 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 a *block*, which typically contains 16-64 pages. Moreover, the latency of an erase operation is far higher than that of a read or write. As a result, this *erase-before-write* limitation leads to inferior write performance, especially for random writes—hence the read-write asymmetry. In addition, each flash block can only be erased a finite number of times before it gets worn out.

To emulate a traditional hard disk interface that has no erase operations, flash memory employs a firmware layer—called the *flash translation layer* (FTL)—to implement page mappings for out-of-place updates, *garbage collection*, and *wear leveling*. A lot of research has been conducted to improve the effectiveness of FTL [35], [45].

The internal structure of flash memory has been well studied [2]. More recently, a performance study was conducted to analyze the system issues of flash memory [7]; and the uFLIP benchmark [5] was proposed to explain the performance characteristics of flash memory by evaluating the spatial and temporal correlations of flash I/O patterns.

2.2 Data Management on Flash-Based Storage Devices

Data management on flash-based storage devices has been extensively studied in recent years. Lee et al. [19] have shown that several components and operations of databases (such as logging, MVCC, and merge joins) are naturally suited for the I/O characteristics of flash memory. Additionally, basic database constructs of flash memory, such as indexing and joins, have been studied [28]. Both studies

conclude that using flash-based storage devices provides better performance for database applications than using magnetic hard disks.

Specialized data structures and algorithms have been designed to address the poor performance of random writes in flash memory. Lee and Moon [18] proposed the in-page logging (IPL) to improve update performance. In contrast to 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 [1], [20], [21] have been proposed to address the read-write asymmetry with lazy and batched updates. Tsirogiannis et al. [44] demonstrated that the column-based layout within a page can leverage fast random reads of flash memory to speed up different query operators. Li et al. [22] proposed new flash-aware algorithms to optimize the non-indexed join processing of flash memory. Chen [8] proposed a synchronous logging solution by exploiting the fast sequential writes of flash devices. In a previous study [31], we proposed a new commit scheme called *flag commit* for supporting efficient transaction recovery in flash-based databases. Unlike the aforementioned studies that developed flash-optimized structures and algorithms for data access and transaction management, this paper investigates flash-optimized buffer-management policies for systems running on flash memory devices.

2.3 Buffer-Management Policies

Buffer management is an active research area in the study of databases and operating systems. The theoretical 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. The policy most widely used by commercial systems is LRU and its variants [15], [32]. LRU always evicts the least recently used page. 2Q [15] is a clock-based approximation of LRU, supporting higher concurrency. LRU-K [32] 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. Cost-based victim selection has also been studied in different scenarios [36]. In contrast, our work develops a cost-based victim-selection algorithm for flash memory devices.

Recently, several buffer-management policies have been proposed to address the read-write asymmetry of flash memory [34], [37]. FAB [14] and BPLRU [16] are two block-level buffer-management policies. They are both designed for the small buffers in embedded flash devices. The techniques used in these proposals can be categorized into two types: giving priority to choosing clean pages as victims over dirty pages [23], [24], [34], [37], and improving the locality of writes [16], [34]. The first category of techniques is more relevant to our study. CCF-LRU [23] considers the access frequencies of clean pages in their clean-page first policy. Lv et al. [24] analyzed the locality and the cost of read/write operations and they adjusted the victim-selection mechanism based on their cost analysis. AD-LRU [13] separates the buffer pool into a *cold* LRU queue and a *hot* LRU queue, based on reference frequencies. The sizes of the two queues are adjusted according to the access pattern. In our cost model, we consider not only the workload but also read-write asymmetry.

To address the read-write asymmetry, Clean-First LRU (CFLRU) [37] maintains the LRU list in two regions: the *working region* and the *clean-first region*. The working region consists of the most recently used pages that are placed at the head of the LRU list, and the clean-first region is at the tail of the LRU list. Victims are identified in the following order: first clean pages in the clean-first region, then dirty pages in the clean-first region, and finally the working region. The size of the working region is a parameter in CFLRU. Based on CFLRU, CFDC [34] further splits the clean-first region into a clean queue and a dirty queue, and it avoids scanning extra dirty pages in the clean-first region of CFLRU. In both studies clean pages are always given a higher priority for replacement than dirty pages.

Recent adaptive algorithms like CASA [33] and ACR [40] adjust the clean-first region with a heuristics-based approach. Suppose the number of clean and dirty pages in the buffer is C and D, respectively. If there is a clean page hit, CASA heuristically assigns a weight to the page of $\frac{D}{C}$. Similarly, in the case of a dirty page hit, CASA assigns a weight of $\frac{C}{D}$. The victim selection is based on the weight. However, the reasons of using this way to assign weight were not explained in the previous study [33].

In contrast, the approach proposed in this paper adaptively determines their priority based on a well-developed cost model. Write clustering has been considered an effective technique to reduce the total cost [34], [38]. It groups dirty pages with locality into clusters to take advantage of efficient sequential writes of flash memory. CFDC [34] further enhances CFLRU by clustering dirty pages and evicting them consecutively. Similar techniques have been used in the recently-evicted-first algorithm [38].

Several previous studies have explored partitioning the buffer pool into two or multiple separate regions for different purposes. DBMIN [10] was proposed to allocate a separate buffer pool for each query. In order to capture both recency and frequency, ARC [26] and its clock-based approximation CAR [3] divide the buffer pool into two parts: one region contains frequent pages, the other contains recent pages. Cesana and He [6] developed a multi-buffer scheme for saving energy consumption of accessing flash memory. Unlike 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 memory.

Our previous work has given preliminary test results for the FD-Buffer [30]. This study goes beyond the previous work by (a) providing a novel cost model to estimate the ratios of the clean pool and the dirty pool in the buffer; and (b) reporting more extensive experiments on real flash memory devices.

2.4 Buffer Miss-Rate Estimation

Buffer miss-rate estimation is important for the effectiveness of buffer management. Tran et al. [43] used curve-fitting techniques to estimate the buffer miss rate of several replacement policies, including FIFO and LRU.

TABLE 2
Parameters and Notations

Para.	Description
C_{read}	The average cost for reading a page from flash memory
C_{write}	The average cost for writing a page from flash memory
$\mathbb{R}$	The asymmetry factor of flash memory ($\frac{C_{write}}{C_{read}}$)
$Cost_{io}$	The average I/O cost per page access
$Cost'_{io}$	The normalized average I/O cost (normalized to C_{read})
P_{total}	The buffer miss rate of the entire buffer
E_{dirty}	The ratio of evicting dirty pages to all evictions
M	The total buffer size (in pages)
Parameters for FD-Buffer	
M_c, M_d	The clean and dirty pool threshold sizes, respectively
P_c, P_d	The miss rates in clean and dirty pools, respectively
P_d^w	The miss rate of writes in the dirty pool

Mattson et al. [25] proposed a stack-based algorithm to estimate the buffer miss rate for LRU and its variants of any buffer size. This is done with a single-pass scan on the page references. Kim et al. [17] reduced the computational overhead of the stack-based algorithm, which has been widely studied in different replacement policies such as LRU, LIRS, and MRU. In this paper, we will extend their technique to estimate the cost of two-pool management in our cost-based buffer replacement algorithm (Section 4.3).

3 Problem Definition

The buffer miss rate is not consistent with I/O performance because of the read-write asymmetry. Therefore, we use the average I/O cost as the primary metric. In particular, we consider the average I/O cost per page access (*the average I/O cost* in short) after the warm-up (i.e., after the buffer is filled). This excludes the cost of buffer misses during warm-up. Thus, the average I/O cost in our model includes two parts: the cost of fetching a page from flash memory upon a buffer miss and the cost of writing a page back to flash memory upon evicting a dirty page. Eq. (1) gives the average I/O cost:

$$Cost_{io} = P_{total} \cdot C_{read} + P_{total} \cdot E_{dirty} \cdot C_{write}, \quad (1)$$

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 average costs for reading and writing a page from the flash memory, respectively. For simplicity, we assume each read operation on the flash memory 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 flash memory devices justify the effectiveness of this simple cost model (see Section 5 for more details). Table 2 summarizes the notations used throughout the paper.

To quantify the read-write asymmetry of the flash memory, we further define the asymmetry factor as $\mathbb{R} = \frac{C_{write}}{C_{read}}$. On a real flash memory device, $\mathbb{R}$ can be dynamic. Its value is usually estimated according to its history [7], [33]. For the ease of presentation, we treat $\mathbb{R}$ as a constant, and the formula can easily be extended to be the cases of dynamic $\mathbb{R}$ values. Normalizing Eq. (1) by C_{read}, we obtain the *normalized average I/O cost* in Eq. (2):

TABLE 3
Examples of Belady's Algorithm and FD-Buffer

	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

$$Cost'_{io} = P_{total} \cdot (1 + E_{dirty} \cdot \mathbb{R}). \quad (2)$$

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

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

Due to the read-write asymmetry of flash memory, the traditional miss-rate-optimized replacement policies are no longer cost-optimal. Table 3 gives an example to show that Belady'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. (2), 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 $\mathbb{R} < 2$, Belady's algorithm wins. When $\mathbb{R} = 2$, they have the same cost. If $\mathbb{R} > 2$, the alternative algorithm wins, and the performance gap will increase as $\mathbb{R}$ increases.

This definition suggests some guidelines in the design of a cost-optimal buffer-management algorithm for flash memory devices. An ideal buffer-management algorithm should minimize both P_{total} and E_{dirty}. In a fixed-size buffer, these two sub-goals may conflict with 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. The key issue is how to achieve a

balance between these two sub-goals so that 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 manager that attempts to minimize the average I/O cost in flash memory.

Without knowledge of future page references, FD-Buffer follows 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 of each other, with each being managed by a traditional buffer-management policy to exploit locality. This design allows us to utilize the previous research results of buffer-management policies for each sub-pool. Second, the relative size of the clean pool and the dirty pool affect the global localities of reads and writes in the entire buffer pool. We *dynamically* adjust the size ratio of the two sub-pools by comparing their localities. Since the total size of the buffer pool is fixed, increasing the size of one sub-pool will reduce misses in it, but increase misses in the other sub-pool. We further apply the write-clustering technique to the dirty pool so that dirty pages belonging to the same erase block are consecutively written back to the flash memory.

Unlike the existing policies, such as CFLRU [37] and CFDC [34], which depend highly on a specific existing replacement policy, each pool of FD-Buffer can be managed by an independent policy. This flexibility enables FD-Buffer to integrate various traditional buffer-management policies with little modification.

4.1 Overview

FD-Buffer has three main components: a *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 $Policy_d$ for 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 memory characteristics and the runtime workload.

The cost estimator is responsible for estimating the cost for different M_c values, with the runtime statistics and flash memory 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 of the flash memory includes the average latency for reads and writes, as well as, the asymmetry factor, all of which are obtained through measurements at runtime. The reason for runtime measurement as opposed to offline calibration is that these characteristics of the flash memory can change dynamically over time [7].

The policy advisor is used to adaptively recommend the optimal setting to the buffer manager. It determines the optimal setting by choosing the clean pool ratio that minimizes the I/O cost of flash memory.

4.2 Replacement Algorithms in FD-Buffer

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

Our FD-Buffer algorithm uses APIs including *Lookup*, *Update*, *Add*, *Remove*, and *GetVictim*, which are commonly used in other buffer algorithms. *Lookup* locates a page in the pool and returns the frame that contains the page. *Update* updates the book-keeping data structure to record that the page is referenced. *Add* adds a page frame to the pool. *Remove* removes a page frame from the pool. *GetVictim* gets a victim frame for replacement.

To illustrate these commands, 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 the metadata of all unused page frames. *Add* adds the metadata of a new page frame to the tail of the queue. *Remove* removes the metadata of the page frame from the queue. *Lookup* locates a page frame in the queue. *Update* moves the metadata of the page to the tail of the queue. *GetVictim* selects the page frame at the head of the queue as the victim.

Algorithm 1 illustrates our FD-Buffer algorithm. The entire buffer pool is divided into the clean pool, C, and the dirty pool, D. For both read and write operations, FD-Buffer first checks the clean pool and then the dirty pool. Maintaining the locality of each individual pool is the responsibility of each corresponding replacement policy, whereas FD-Buffer is in charge of adjusting the sizes of both pools. We define the number of frames in the clean and the dirty pool as $|C|$ 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 them. A frame moves from the clean pool to the dirty pool in two scenarios: (1) while writing a page in the clean pool or (2) while 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 $|D|$ dynamically approach M_c and M_d, respectively.

Table 3 shows an example of running FD-Buffer. 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_I comes, D grows to one page. The dirty page J stays in the dirty

pool for its next write, since no other writes occur during this period. In this example, FD-Buffer can achieve a lower cost than Belady's algorithm.

Algorithm 1 Reads and Writes on FD-Buffer.

Proc.: Read (Page p)

```
1: Frame v = C.Lookup(p);
2: if v ≠ NULL then
3:    C.Update(v);
4: else
5:    v = D.Lookup(p);
6:    if v ≠ NULL then
7:       D.Update(v);
8:    else
9:       v=FindVictimForClean (C, D).   // Algorithm 2.
10:      Fetch page p from the flash memory to frame v;
11: Return frame v;
```

Proc.: Write (Page p)

```
1: Frame v = C.Lookup(p);
2: if v ≠ NULL then
3:    C.Remove(v);
4:    D.Add(v);
5: else
6:    v = D.Lookup(p);
7:    if v ≠ NULL then
8:       D.Update(v);
9:    else
10:      v=FindVictimForDirty (C, D).   // Algorithm 2.
11:      Fetch page p from the flash memory to frame v;
12: Return frame v;
```

Algorithm 2 Page evictions in FD-Buffer.

Proc.: 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 flash memory;
6: else
7:    v = C.GetVictim();
8: Return frame v;
```

Proc.: 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 flash memory;
8: Return frame v;
```

Next we analyze the average I/O cost of FD-Buffer based on the miss rate. A buffer miss occurs in FD-Buffer only when the miss occurs in both the clean pool and the dirty pool. We also note that, for every access, FD-Buffer starts by checking the clean pool; hence, not every access goes to the dirty pool. Given the miss rates 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 define this miss rate as P_d^w. By definition, $E_{dirty} = P_d^w$. Substituting P_{total} and E_{dirty}, we get the normalized average I/O cost for FD-Buffer as expressed in Eq. (3). Note, to adapt to the dynamics of flash memory, we estimate the $\mathbb{R}$ value according to the historical read and write performance. In particular, we estimate $\mathbb{R}$ to be the ratio of the average latency of the previous m writes and the average latency of the previous m reads (m is set at 32,768 by default, as in a previous study [33]).

$$Cost'_{io} = P_c \times P_d(1 + P_d^w \cdot \mathbb{R}). \tag{3}$$

P_c and P_d are two independent metrics, although the clean pool size and the dirty pool size are correlated (i.e., their sum is equal to the total buffer size). Note that when a page access request arrives at the FD-Buffer, the probability that the page is not found in the clean pool is independent of the probability that the page is not found in the dirty pool, and vice versa.

4.3 Cost Estimation

In order to adapt the size of the clean pool to the characteristics of the flash memory 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 motivates us to perform online estimation of P_c, P_d, and P_d^w.

We estimate these miss rates based on Mattson's stack-based algorithm [25]. Mattson's algorithm uses a stack to store page accesses, and it estimates the miss ratio based on the calculation of stack distances. Since the traditional Mattson's stack algorithm only works for a single buffer pool, we propose the *Two-Stack* algorithm, which extends the Mattson's algorithm for the two pools in FD-Buffer.

Before we present our *Two-Stack* algorithm, let us briefly review the basic ideas of Mattson's algorithm. Recall that we use LRU and its variants as buffer-replacement policies in this paper. Mattson's algorithm is based on the inclusion property of LRU: for any sequence of memory accesses, the contents of a buffer with k pages should be a subset of the contents of a buffer of size $k+1$ or larger. To estimate the buffer miss rate, Mattson's algorithm uses an LRU stack to store the accessed pages, such that the most recent page is on the top of the stack. The algorithm maintains an array of hit counters, $Hit[1, 2, \ldots, \infty]$, to keep track of the miss counts under different buffer sizes.

Upon each page access, the algorithm finds the referenced page in the simulated stack and updates the stack to reflect the memory contents based on the replacement policy. For example, under the LRU policy, the referenced page will be moved to the top of the stack. Before the movement, if the referenced page is the dth element from the top of the stack, the *stack distance* of the current access is d; otherwise if the referenced page is not found in the current stack, its distance is considered to be ∞. This reflects the fact that if the buffer size M is between 1 and $d-1$ pages, this access will result in a page miss. On the other hand, if M is larger than or equal to d, the access will result in a page hit. For a page

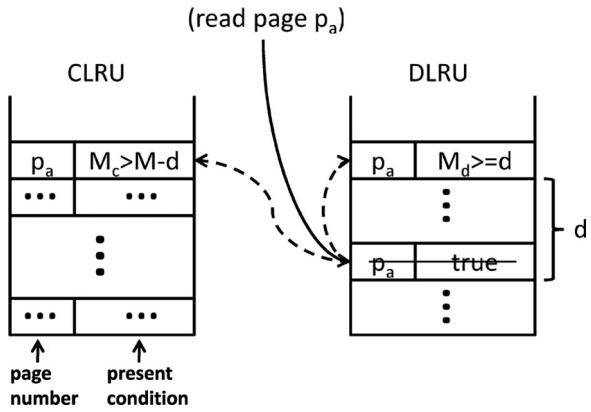


Fig. 1. An example of Two-Stack simulation for page read access.

access of distance d, the hit counter $Hit[d]$ will be increased by 1. Thus, given a sequence of page accesses, the buffer miss rates for various memory sizes can be calculated according to the hit counters. Specifically, the miss rate of $P(M)$ for an M-page buffer on an access sequence of N distinct pages is given by Eq. (4):

$$P(M) = 1 - \frac{\sum_{i=1}^{M} Hit[i]}{\sum_{i=1}^{N} Hit[i] + Hit[\infty]}. \quad (4)$$

Now we explain our *Two-Stack* algorithm, which is formally described in Algorithms 3 and 4. The extension of Mattson's algorithm to *Two-Stack* is a non-trivial task, since the clean pool C and the dirty pool D interact with each other. *Two-Stack* uses two LRU stacks named $CLRU$ and $DLRU$ for C and D, respectively. $CLRU$ and $DLRU$ store the accessed pages according to their access recency.

For each page access, there are two steps involved in the *Two-Stack* algorithm:

Step 1: We first move the referenced page to the top of its corresponding stack(s). Since a page in C can be moved to D (and vice versa) in FD-Buffer, we also allow page movement between $CLRU$ and $DLRU$. For example, a write access to a page in $CLRU$ may move the page to $DLRU$, since the page becomes dirty afterwards. However, it is a challenge to handle the page movement from $DLRU$ to $CLRU$. Consider a case where a read access arrives, and the referenced page exists in $DLRU$ but not in $CLRU$. After the read access, the page may remain in $DLRU$ if it is a hit in the dirty pool. Otherwise, the read access is 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. To determine the residence pool of each page, we introduce the notion of *present condition*, *PCond*. In the scenario mentioned above, we will maintain a pair of complementary conditions $\{PCond_c, PCond_d\}$ for the corresponding entries in $CLRU$ and $DLRU$. Given the size of the dirty pool M_d (or the size of the clean pool M_c), only one *present condition* can be true, which implies that a page can exist in either C or D. Fig. 1 shows an example of page movement from $DLRU$ to $CLRU$. Assume that page p_a exists only in $DLRU$, and its *stack distance* is d. Upon a new read access to p_a, we need to move the page to the top of $DLRU$ and copy it to the top of $CLRU$. Accordingly, we will update $PCond_d[p_a] = (M_d \geq d)$ and $PCond_c[p_a] = (M_c > M - d)$ for the corresponding entries in $DLRU$ and $CLRU$, respectively. Since $M_c + M_d = M$, only one condition can be true for any size of M_c or M_d.

Algorithm 3 Handling a read in Two-Stack.

Proc.: 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 by $PCond_c[e_c]$,
in either of the following two ways:
a) if $PCond_c[e_c] = true$ then $i = d_c$;
b) if $PCond_c[e_c] = (M_c \geq d_c')$ then $i = Max\{d_c, d_c'\}$;
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 by $PCond_c[e_c]$,
in either of the following two ways:
a) if $PCond_c[e_c] = true$ then $i = d_c$;
b) if $PCond_c[e_c] = (M_c \geq d_c')$ then $i = Max\{d_c, d_c'\}$;
21: Compute j by $PCond_d[e_d]$,
in either of the following two ways:
a) if $PCond_d[e_d] = true$ then $i = d_c$;
b) if $PCond_d[e_d] = (M_d \geq d_d')$then $i = Max\{d_d, d_d'\}$;
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: $PCond_d[e_d'] = (PCond_d[e_d] \wedge M_d \geq d_d)$;

Step 2: Next we increment the hit counter for an element at position i for $CLRU$ and $DLRU$, respectively. In Mattson's algorithm, i has the same value as the *stack distance* d of the current access, since d reflects the actual position of the referenced page in the pool. However, this is not the case in our *Two-Stack* algorithm, since d may miscount some pages that actually do not exist in the pool under a given buffer size. To address this issue, we derive the value of i in the following way: given a referenced page p_e, if $PCond[p_e]$ is *true*, then $i = d$; otherwise, $PCond[p_e]$ must be in the form of $M_c > d'$, and i is set to the larger of d and d'. This is due to the inclusion property of LRU. Specifically, when $PCond[p_e] = true$, although the distance i may count the pages that do not exist in the pool, it guarantees that the current access is a hit for any

buffer with a size greater than or equal to i. If $PCond[p_e]$ is in the form of $M_c > d'$, the larger of d and d' would have the same guarantee. After deriving the value of i, the corresponding hit counter is incremented. The *Two-Stack* algorithm maintains the hit counters for both $CLRU$ and $DLRU$. It also distinguishes the hit counters for reads and writes, since the computation of P_d^w depends on write accesses only. We denote the hit counters for reads and writes in the clean (dirty) pool as by Hit_c^r and Hit_c^w (Hit_d^r and Hit_d^w), respectively.

Finally, given an access sequence on N distinct pages, P_c, P_d, and P_d^w are computed by Eqs. (5)-(7), respectively

$$P_c(M_c) = 1 - \frac{\sum_{i=1}^{M_c} \left(Hit_c^r[i] + Hit_c^w[i]\right)}{\sum_{i=1}^{N} \left(Hit_c^r[i] + Hit_c^w[i]\right) + Hit_c^r[\infty] + Hit_c^w[\infty]}, \tag{5}$$

$$P_d(M_d) = 1 - \frac{\sum_{i=1}^{M_d} \left(Hit_d^r[i] + Hit_d^w[i]\right)}{\sum_{i=1}^{N} \left(Hit_d^r[i] + Hit_d^w[i]\right) + Hit_d^r[\infty] + Hit_d^w[\infty]}, \tag{6}$$

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

Algorithm 4 Handling a write in Two-Stack.

Proc.: 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 by $PCond_c[e_c]$,
in either of the following two ways:
a) if $PCond_c[e_c] = true$ then $i = d_c$;
b) if $PCond_c[e_c] = (M_c \geq d'_c)$ then $i = Max\{d_c, d'_c\}$;
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 by $PCond_d[e_d]$,
in either of the following two ways:
a) if $PCond_d[e_d] = true$ then $i = d_d$;
b) if $PCond_d[e_d] = (M_d \geq d'_d)$then $i = Max\{d_d, d'_d\}$;
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: $PCond_d[e''_d] = true$;

A direct implementation of the cost estimation in FD-Buffer might incur a high runtime overhead because the stacks are accessed randomly. In order to improve the efficiency of our stack-based estimation, we use an LRU stack for each sub-pool, and we use the group-based optimization suggested in a previous study [17]. The basic idea of the optimization is to group the pages by their access recency into g groups $G_0, G_1, \ldots, G_{g-1}$, with each group consisting of z pages. G_0 consists of the most recently accessed pages. The stack distances for all the pages within a group G_k are set to approximately the same value (i.e., $k \cdot z$). We maintain the pointers to the pages in the header of each group. Upon each page access, only the affected group headers are updated, instead of the affected pages. As such, for each page access, the stack-access complexity is reduced to $O(D)$, where D is the number of groups. As demonstrated in a previous study [17], group-based optimization can greatly reduce overhead without significantly reducing accuracy. In our performance evaluation, the accuracy loss in the miss rate estimation is less than 5 percent.

4.4 Policy Advisor

The policy advisor predicts the optimal replacement for future page references. Since workload estimation is generally difficult, we use a simple window-based estimation method, where a window is defined as a predefined number of consecutive page references. We denote the previous window and the current window as Win and Win', respectively. The policy advisor decides the M_c value based on the statistics collected from Win, and it uses the M_c value for Win'.

Based on the hit counters in Win, we iterate M_c from 1 to $(M-1)$, and use Eqs. (5)-(7) to compute P_c, P_d, and P_d^w values. With P_c, P_d, and P_d^w values, we use Eq. (3) to compute the normalized average I/O cost for each M_c value. Thus, we obtain the optimal M_c value of Win as the value that minimizes the average I/O cost among the estimated costs.

The window size is a tuning parameter for the policy advisor. Ideally, it should balance the gain of adaptation in workload changes and the overhead of running the policy advisor. If the window size is too small, the policy advisor will be executed too often and its computational overhead will be high. On the other hand, if the window size is too large, the policy may not adapt well 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 cost incurred in the current window. We define the computation time of the cost estimation in the policy advisor as $Comp$, and we set the window size so that the total I/O cost reaches $\frac{Comp}{q}$. We demonstrate the effectiveness of this simple method in our experiments.

4.5 Write Clustering

Write clustering is an effective technique for exploiting the locality of writing multiple consecutive pages to the same erase block. The basic idea is that, since the cost of erasing an erase block is a bottleneck for writes to flash memory, writing multiple pages belonging to the same erase block could reduce multiple erase operations to one. Specifically, we group dirty pages in the buffer pool into clusters according to their logical addresses. On the first eviction request, we choose a victim cluster based on the metric function developed in a previous study [34], and then we flush one

page from the victim cluster. Each subsequent eviction request will flush the next page from the same cluster until the whole cluster is emptied; after that, another victim cluster will be chosen, and the same process is followed. In this way, multiple writes are likely directed to a limited number of blocks. Although FTL hides the real logical-to-physical address mappings, the effectiveness of this simple scheme has been evaluated and validated for different flash memory devices [34], [39].

Compared with existing buffer-management policies (such as CFDC), FD-Buffer has several advantages of using the write-clustering technique. First, our cost model prefers that the set of hot dirty pages is kept in the buffer, and the two-pool design of FD-Buffer usually keeps more dirty pages in the buffer, as we observed in our experiments. This offers more opportunities to increase the number of dirty pages in clustered writing. Second, FD-Buffer has less computational overhead in finding a victim cluster, since all the dirty pages are already maintained in a localized dirty pool.

There are two issues worth noting. First, the write-cluster size (i.e., the number of pages in a victim cluster) should be adapted to flash memory characteristics. To make the clustered write fit into one erase block, we set the cluster size as $\lfloor \frac{e}{p} \rfloor$, where e is the erase block size and p is the page size of the flash memory. The e value can usually be found in product specifications, and the p value depends on the application.

Second, to minimize the modification of our cost model, we carefully design the conversion from dirty pages to clean pages. In particular, during page replacement, we move only the evicted dirty page to the clean pool, and we keep all other dirty pages of the same cluster in the dirty pool. This is to reduce the number of page movements between the two pools. In practice, this is useful because dirty pages usually have temporal locality and will be written again soon. With this careful design, our cost model achieves good accuracy, as will be demonstrated in the performance evaluation.

5 Performance Evaluation

In this section, we evaluate our algorithms with trace-driven experiments on real flash memory devices.

5.1 Experimental Setup

We ran our simulation experiments on a Windows workstation with an Intel 2.4 GHz Quad-Core CPU, with 4 GB main memory, a 160 GB 7200 rpm SATA magnetic hard disk, and two flash memory devices. The hard disk supports 109 random reads and 100 random writes on 8 KB pages per second. To evaluate the impact of different asymmetry factors, we used two flash memory devices: a Kingston SDHC 8 GB and a Sandisk Cruzer USB 8 GB. Without on-device caches, the data access characteristics of flash memory are well captured by the SD card and the USB. Because they have asymmetry factors of 136 and 475, we denote them as "Flash-136" and "Flash-475," respectively. We also use one 16 GB Mtron MOBI3500 SSD (denoted as "Flash-25" for its asymmetry factor) to investigate the performance of FD-Buffer on SSDs.

TABLE 4
Specification of the Traces in the Experiment

	Dataset size (GB)	#References (millions)	Write ratio	Description
TPC-C	2.4	16.8	15.0%	20 warehouses
TPC-B	2.2	12.7	3.5%	150 branches
TM1	2.4	10.2	4.6%	1 million subscribers
File_System	4.5	15.9	3.4%	System-call I/O traces to file system

Workloads. The workloads include publicly available benchmarks on transactional processing (TM1 [29], TPC-B [41], and TPC-C [42]) and a file-system trace (see Table 4). The file-system trace includes system-call I/O accesses to a file system, named LASR.[1] As for the three benchmarks, TM1 is a telecom workload benchmark, TPC-B simulates transactions on a hypothetical bank, and TPC-C is for online transaction processing. To get the traces for buffer page accesses, we ran the benchmarks on PostgreSQL with default settings (e.g., a page size of 8 KB). For each benchmark, we ran the test for a sufficiently long period of time (around 3 hours), including a 30-minute warm-up period. The number of clients was set at 20 for all benchmarks. We explicitly configured the buffer size to be smaller than the data set size in order to exercise disk I/O operations. Moreover, the trace includes all of the accesses to disk pages (including those hit in the buffer), while the logging I/Os are not included in the trace. In addition to the default data sets listed in Table 4, we also used a larger TPC-C data set of 12 GB.

In order to evaluate the performance impact of workload dynamics, we simulated the dynamics in the read/write ratio. In particular, we divided the TPC-C trace into epochs, with each epoch consisting of around 5,000 page references. We dynamically changed the write ratio w in these epochs by changing reads to writes in the trace. We used two models for simulating workload 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, and w_0 is obtained in the first epoch of the original TPC-C trace. The first model simulates the case when write requests become dominant in the workload, and the second model simulates the case where the workload periodically changes.

Implementation for buffer manager. We implemented a buffer manager on top of the standard OS file system facilities. The buffer manager takes a trace as input and performs I/O requests for the flash memory. Thus, we can obtain the response time for each buffer page request for the flash memory.

Both of FD-Buffer's two pools are managed by LRU. We set the group size to be 64 in the group-based stack-distance calculation. We evaluate the effectiveness of FD-Buffer in comparison with LRU, CFDC [34], and CASA [33], the representatives of traditional disk-based and flash-based replacement algorithms. We also implemented the write-clustering technique for all of the buffer-replacement algorithms. We use LRU as the baseline for comparison, since LRU and its variants are the

1. http://iotta.snia.org/tracetypes/1.

TABLE 5
Miss Rate Estimation with and without *Present Conditions* in FD-Buffer

		P_c	P_d	P_d^w
TPC-C	w/ PR	5.10%	7.50%	4.60%
	w/o PR	186.90%	8.20%	4.60%
TPC-B	w/ PR	18.60%	1.70%	17.50%
	w/o PR	108.30%	2.00%	19.50%
TM1	w/ PR	16.60%	0.90%	3.60%
	w/o PR	59.70%	1.00%	6.60%
File_System	w/ PR	4.10%	12.10%	0.60%
	w/o PR	93.30%	20.70%	24.10%

most widely used replacement policies in traditional buffer managers. By default, we set the ratio of the clean-first region in CFDC at 0.5, as in a prior study [34]. CASA is a heuristic-based adaptive algorithm for victim selection. The comparison with CASA is meant to assess the effectiveness of our cost-based adaptation.

To evaluate the impact of different read-write asymmetry levels, we intentionally add latency to the read or write operations to increase their latency. A high asymmetry level 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 of approximately 1, and a low-end flash memory may have a high asymmetry factor. Furthermore, due to fragmentation, the asymmetry factor may significantly increase on the same flash memory over time. We simulated the dynamics in the asymmetry level in a way similar to workload dynamics. In particular, we used 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 the initial $\mathbb{R}$ value of the specific device. The first model simulates the increasing asymmetry as the flash memory becomes fragmented, and the second model simulates the scenario where the fragmented flash memory is periodically cleaned (such as by using the latest *Trim* command) and restored a relatively low asymmetry.

For all of the replacement algorithms, a page that was updated in the buffer pool is marked as dirty, and it will be written to flash memory when it is evicted. To avoid 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 measure the read/write speed at runtime and estimate the asymmetry level online. To smooth out short-term fluctuations in I/O speeds, we use the same method as CASA [34], which is an n-point moving average of the measured values. By default, the size of each workload estimation window is set to 5,000 page references so that the overhead of the policy advisor does not exceed 1 percent. We also experimentally evaluate the impact of different window sizes.

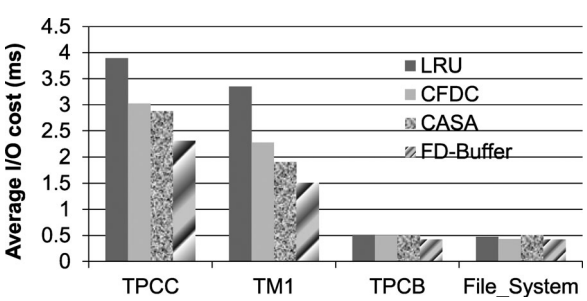


Fig. 2. Avg. I/O cost of different algorithms (Flash-136).

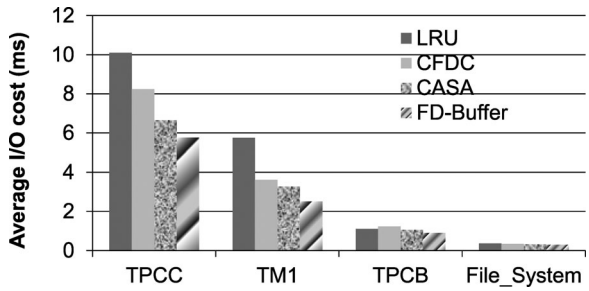


Fig. 3. Avg. I/O cost of different algorithms (Flash-475).

5.2 Results

Now we present the results of trace-driven simulations on the flash memory devices. We vary the buffer size and the simulated asymmetry factor in the evaluation. By default, the buffer is set at 3 percent of the data set size, and we do not add any latency to read/write operations.

Evaluating miss-rate estimations. Table 5 shows the average errors between the estimations and measurements on P_c, P_d, and P_d^w values for running all of the traces under the default settings on Flash-136. The results on Flash-475 are similar and hence omitted to save space.

The error is defined as $e = \frac{|v-v'|}{v} \times 100\%$, where v is the measured value and v' is the estimated value. We compare the estimations with and without *present conditions*, denoted as "w/ PR" and "w/o PR." As the results show, the extension of using *present conditions* significantly reduces the estimation error. In particular, the improvement is most significant for P_c, since P_c is affected most by page movement between the two stacks in the Two-Stack algorithm. Such a small error is important for the effectiveness of our adaptation.

Overall comparison. Figs. 2 and 3 show the average I/O cost of FD-Buffer compared to LRU, CFDC, and CASA on the two flash memory devices. All algorithms are with write clustering enabled. According to the device specifications, we set the write cluster size to 256 on both Flash-136 and Flash-475. FD-Buffer significantly outperforms the other three algorithms in most cases. Compared with LRU, CFDC, and CASA, the average improvement of FD-Buffer on the four traces is 30.4, 18.1, and 18.1 percent on Flash-136 and 33.4, 24.2, and 15.4 percent on Flash-475, respectively.

As for computational overhead, FD-Buffer and CASA have a similar cost, which is only about 0.2 percent of total I/O time. Such a small overhead is negligible to FD-Buffer, thanks to the group-based optimization for deciding stack distances in the Two-Stack algorithm. Moreover, the estimation of buffer miss rates is performed only at the beginning of each workload estimation window. As a result, it achieves a computational overhead comparable to CASA, where each buffer hit may incur a buffer-pool adjustment.

TABLE 6
Disk Read and Write Operations for Different Algorithms (TPC-C, Flash-136)

	Reads	Writes	Total
LRU	56,121	35,477	91,598
CFDC	61,533	30,394	91,927
CASA	60,995	30,944	91,939
FD-Buffer	72,923	27,863	100,786

To better understand the performance improvement of our algorithm, we next examine the number of buffer misses, the effectiveness of adaptation, and the impact of write-cluster size. Since we observed similar results of running the four traces on the two flash memory devices, in the following part, we present the results of running TPC-C on Flash-136 only.

Table 6 compares the number of reads and writes and the total number of disk operations for different algorithms running TPC-C on Flash-136. Being aware of the read-write asymmetry, CFDC, CASA, and FD-Buffer all have a smaller number of writes than LRU, with the cost of incurring a larger number of reads. This results in a higher buffer miss rate for these three algorithms in comparison with LRU. Nevertheless, due to the read-write asymmetry of flash memory, they have better overall performance than LRU. Among the flash-aware algorithms, CFDC is static, and CASA and FD-Buffer are both adaptive. The cost model in FD-Buffer guides the buffer management to adaptively give priority to dirty pages. Consequently, FD-Buffer has a smaller number of writes than CFDC and CASA, and FD-Buffer achieves the best overall performance.

Fig. 4 shows the number of clean pages in FD-Buffer and CASA while they run TPC-C. FD-buffer has a more fluctuated pattern on the clean pool size. To gain more insight into this pattern, we also plot the ratio of the total read cost to the total I/O cost for each workload estimation window. We can see that this ratio also fluctuates, and FD-Buffer more closely follows this fluctuation. In contrast, CASA is not very sensitive to these fluctuations. This demonstrates that FD-buffer is more adaptive to the read/write cost fluctuations in the workload.

Fig. 5 shows the average number of clean pages for different algorithms running TPC-C. FD-Buffer has the smallest average number of clean pages. Since LRU does not separate the clean and dirty pages, it has the worst combination of read/write counts. Although CFDC separates the clean and dirty pools, it fails to manage them adaptively. With adaptive adjustments of the clean/dirty pool size, FD-Buffer has a relatively large number of dirty pages in its buffer pool. This enables it to take greater advantage of clustered writing. To verify this, Fig. 6 shows the average

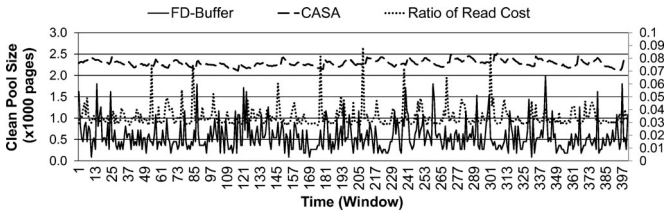


Fig. 4. Number of clean pages during the execution time of TPC-C for FD-Buffer and CASA (Flash-136).

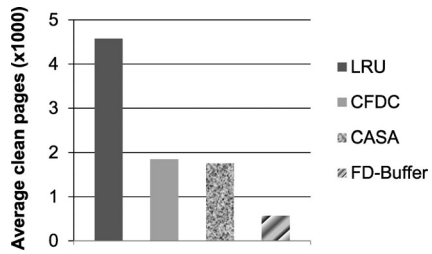


Fig. 5. Average number of clean pages (TPC-C, Flash-136).

number of dirty pages in a write cluster for the four algorithms. As expected, FD-Buffer has the largest number of dirty pages—12.3 percent more than that of CASA.

Finally, Fig. 7 shows the experimental results on Flash-25 with a larger TPC-C data set. We set the data set size to be 12 GB, which is close to the storage capacity of Flash-25, and the ratio of buffer size to data set size is kept at 3 percent. We can see that FD-Buffer significantly outperforms LRU by 37.4 percent and CFDC by 32.2 percent. The improvement over CASA is slightly smaller (10.9 percent), due to a small asymmetry factor.

Comparison under various settings. We now present parametric studies that vary the buffer size, the asymmetry level, and workload dynamics. Since we obtained similar results on the four traces, in the interest of space, we focus our discussion on the TPC-C trace with Flash-136 only. Fig. 8 shows the average I/O cost for the four algorithms when the ratio of buffer size to data set size is varied from 0.75, 1.5, 3, to 6 percent. As the buffer size increases, the improvement of FD-Buffer over CASA increases from 9 to 24 percent. This is mainly because FD-Buffer considers the buffer size in the cost model, whereas CASA adjusts the clean pool based on heuristics only.

Fig. 9 compares the average I/O cost for the four algorithms using models with different asymmetry levels $\mathbb{R}$. In the RM_1 model, where the $\mathbb{R}$ value continuously increases, FD-Buffer achieves a higher improvement than the static algorithms, such as LRU and CFDC. This is because FD-Buffer has the capability to adaptively adjust the clean/dirty pool size. A similar trend is observed in the RM_2 model, where the $\mathbb{R}$ value changes periodically. Compared to the adaptive algorithm—CASA, the improvement of FD-Buffer is higher in the RM_2 model than in the RM_1 model, which suggests that FD-Buffer is able to capture the periodic changes in $\mathbb{R}$ values well.

Fig. 10 shows the average I/O cost for the four algorithms with dynamic workloads. For WM_1, since there

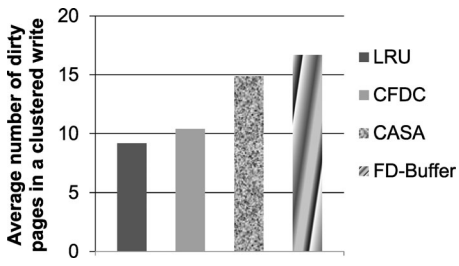


Fig. 6. Average number of dirty pages in a write cluster (TPC-C, Flash-136).

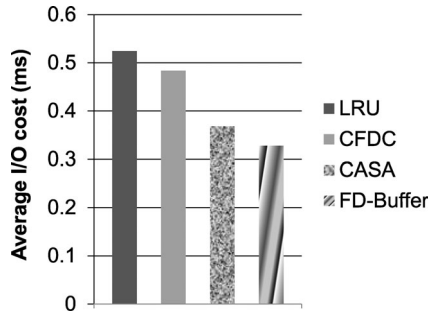


Fig. 7. Experimental results on the SSD (TPC-C, Flash-25).

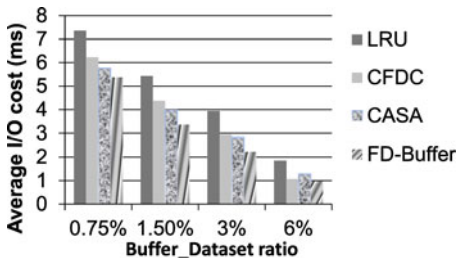


Fig. 8. Average I/O cost with different buffer sizes (TPC-C, Flash-136).

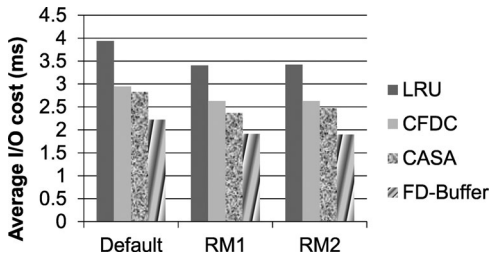


Fig. 9. Average I/O cost under the models with varying $\mathbb{R}$ (TPC-C, Flash-136).

are increasingly more writes, the effect of clustered writing is significant, which narrows the performance gaps between the algorithms. This is consistent with the findings of a previous study [34]. For WM_2, since the workload is more dynamic, the improvement of FD-Buffer over CASA is increased by 2-9 percent, showing the robust adaptivity of our proposed algorithm.

Next, we examine the effect of window size in performing workload estimation, which determines the frequency of running the policy advisor in FD-Buffer. Fig. 11 shows the results of increasing the estimation window size from 0.25 to 5 times of the default size. The overall performance is almost steady for all workloads when the window size is between 0.25 and 1. The runtime overhead of FD-Buffer increases slightly from 0.2 to 1 percent when the window size is decreased from 1 to 0.25. On the other hand, our adaptation on the clean and dirty pool sizes is more effective, which cancels out the small increase in runtime overhead. When the window size is increased from 1 to 5, the runtime overhead is reduced to below 0.1 percent. Nevertheless, the adaption of buffer management to the workload changes is weakened so that the overall performance is degraded slightly, by 2.7-7.1 percent.

Finally, we investigate the effect of varying the epoch size, which determines the fluctuation frequency of the synthetic workloads. The smaller the epoch size, the more

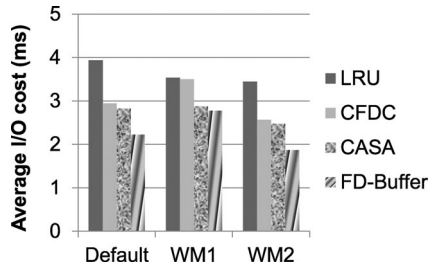


Fig. 10. Average I/O cost under different synthetic workloads (TPC-C, Flash-136).

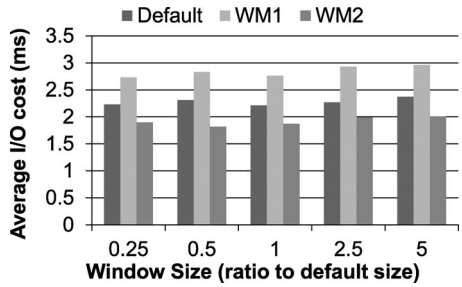


Fig. 11. Varying the estimation window size under synthetic workloads (TPC-C, Flash-136).

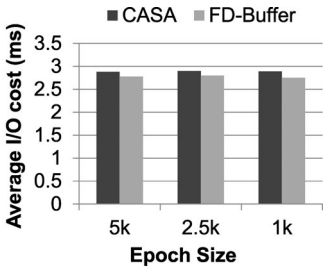
frequent the workload changes. Fig. 12 compares the results of the two adaptive algorithms (CASA and FD-Buffer) under smaller epoch sizes of 1,250 and 2,500 page references (5,000 is the default setting). For the WM1 workload in Fig. 12a, the performance improvement of FD-Buffer over CASA remains at 4 percent, as consistent with Fig. 10. For the WM2 workload in Fig. 12b, when the epoch size is smaller, the performance of FD-Buffer is slightly worse. Nevertheless, the improvement over CASA is still more than 20 percent, which demonstrates the robustness of our FD-Buffer.

In summary, FD-Buffer outperforms the other three algorithms in all aspects, regardless of the buffer size, $\mathbb{R}$ models, and the model of workload dynamics. In particular, its performance gain over the existing algorithms becomes larger when the buffer size is larger, the asymmetry level or the read/write workload is more dynamic.

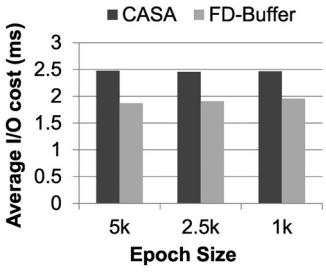
6 Conclusions

This paper studies buffer management in flash memory devices, with a focus on read-write asymmetry and workload dynamics. We have developed a cost-based adaptive buffer replacement algorithm named FD-Buffer, which automatically adapts to the flash memory characteristics and the runtime workload. The atomicity of FD-Buffer significantly reduces the ownership cost of systems running on flash memory devices. Our experimental studies show that FD-Buffer outperforms the existing algorithms, with 4.0-33.4 percent performance improvement under various settings. It is demonstrated to be more effective to dynamic system environments where the asymmetry level and/or read/write workload change over time.

In future work, we are interested in further improving the efficiency of FD-Buffer with other traditional replacement polices (such as 2Q [15]) and with optimizations on write patterns to flash memory.



(a) Performance under WM1 workload



(b) Performance under WM2 workload

Fig. 12. Varying epoch size of the synthetic workloads (TPC-C, Flash-136).

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by a start-up grant (M4080102.020) of Nanyang Technological University, Singapore, the Research Grants Council of Hong Kong SAR, China (Grants 211510 and HKBU211212), and the Natural Science Foundation of China (Grant 60833005).

REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh, "Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices," *Proc. VLDB Endowment*, vol. 2, pp. 361-372, 2009.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigraphy, "Design Tradeoffs for SSD Performance," *Proc. USENIX Ann. Technical Conf.*, 2008.

[3] S. Bansal and D.S. Modha, "CAR: Clock with Adaptive Replacement," *Proc. Third USENIX Conf. File and Storage Technologies (FAST)*, 2004.

[4] L.A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems J.*, vol. 5, pp. 78-101, 1966.

[5] L. Bouganim, B.T. Jónsson, and P. Bonnet, "uFLIP: Understanding Flash IO Patterns," *Proc. Fourth Biennial Conf. Innovative Data Systems Research (CIDR)*, 2009.

[6] U. Cesana and Z. He, "Multi-Buffer Manager: Energy-Efficient Buffer Manager for Databases on Flash Memory," *ACM Trans. Embedded Computing Systems*, vol. 9, article 28, 2010.

[7] F. Chen, D. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives," *Proc. 11th Int'l Joint Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[8] S. Chen, "Flashlogging: Exploiting Flash Devices for Synchronous Logging Performance," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2009.

[9] S. Chen, A. Ailamaki, M. Athanassoulis, P.B. Gibbons, R. Johnson, I. Pandis, and R. Stoica, "TPC-E vs. TPC-C: Characterizing the new TPC-E Benchmark via an I/O Comparison Study," *ACM SIGMOD Record*, vol. 39, pp. 5-10, Feb. 2011.

[10] H.-T. Chou and D.J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. 11th Int'l Conf. Very Large Data Bases (VLDB)*, 1985.

[11] J. Gray. Tape is dead, disk is tape, flash is disk, RAM locality is king. *Pres. at the CIDR Gong Show*, 2007.

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

[13] P. Jin, Y. Ou, T. Härder, and Z. Li, "AD-LRU: An Efficient Buffer Replacement Algorithm for Flash-Based Databases," *Data and Knowledge Eng.*, vol. 72, pp. 83-102, 2012.

[14] 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 Trans. Consumer Electronics*, vol. 52, no. 2, pp. 485-493, May 2006.

[15] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th Int'l Conf. Very Large Data Bases (VLDB)*, 1994.

[16] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," *Proc. Sixth USENIX Conf. File and Storage Technologies (FAST)*, 2008.

[17] Y.H. Kim, M.D. Hill, and D.A. Wood, "Implementing Stack Simulation for Highly-Associative Memories," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 19, no. 1, pp. 212-213, 1991.

[18] S.-W. Lee and B. Moon, "Design of Flash-Based Dbms: An In-Page Logging Approach," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2007.

[19] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2008.

[20] Y. Li, B. He, Q. Luo, and K. Yi, "Tree Indexing on Flash Disks," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, 2009.

[21] Y. Li, B. He, R.J. Yang, Q. Luo, and K. Yi, "Tree Indexing on Solid State Drives," *Proc. VLDB Endowment*, vol. 3, pp. 1195-1206, 2010.

[22] Y. Li, S.T. On, J. Xu, B. Choi, and H. Hu, "Optimizing Non-Indexed Join Processing in Flash Storage-Based Systems," *IEEE Trans. Computers*, vol. 62, no. 7, pp. 1417-1431, July 2013.

[23] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue, "CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory," *IEEE Trans. Consumer Electronics*, vol. 55, no. 3, pp. 1351-1359, Aug. 2009.

[24] Y. Lv, B. Cui, B. He, and X. Chen, "Operation-Aware Buffer Management in Flash-Based Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pp. 13-24, 2011.

[25] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM System J.*, vol. 9, no. 2, p. 78, 1970.

[26] N. Megiddo and D.S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proc. Second USENIX Conf. File and Storage Technologies (FAST)*, 2003.

[27] S.L. Min and E.H. Nam, "Current Trends in Flash Memory Technology: Invited Paper," *Proc. Asia and South Pacific Design Automation Conf.*, pp. 332-333, 2006.

[28] D. Myers, "On The Use of Nand Flash Memory in High-Performance Relational Databases," Msc thesis, MIT, 2008.

[29] Nokia, *Network Database Benchmark*, http://hoslab.cs.helsinki.fi/homepages/ndbbenchmark/, 2014.

[30] S.T. On, Y. Li, B. He, M. Wu, Q. Luo, and J. Xu, "FD-Buffer: A Buffer Manager for Databases on Flash Disks," *Proc. 19th ACM Int'l Conf. Information and Knowledge Management (CIKM)*, 2010.

[31] S.T. On, J. Xu, B. Choi, H. Hu, and B. He, "Flag Commit: Supporting Efficient Transaction Recovery in Flash-Based DBMSs," *IEEE Trans. Knowledge and Data Eng.*, vol. 24, no. 9, pp. 1624-1639, Sept. 2012.

[32] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 1993.

[33] Y. Ou and T. Härder, "Clean First or Dirty First? A Cost-Aware Self-Adaptive Buffer Replacement Policy," *Proc. 14th Int'l Database Eng. and Applications Symp. (IDEAS)*, pp. 7-14, 2010.

[34] Y. Ou, T. Härder, and P. Jin, "CFDC: A Flash-Aware Replacement Policy for Database Buffer Management," *Proc. Fifth Int'l Workshop Data Management on New Hardware (DaMoN)*, 2009.

[35] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A Reconfigurable FTL (Flash Translation Layer) Architecture for Nand Flash-Based Applications," *ACM Trans. Embedded Computing Systems*, vol. 7, article 38, Aug. 2008.

[36] C.-M. Park, K.-Y. Whang, J.-J. Lee, and I.-Y. Song, "A Cost-Based Buffer Replacement Algorithm for Object-Oriented Database Systems," *Information Sciences—Informatics and Computer Science*, vol. 138, pp. 99-118, Aug. 2001.

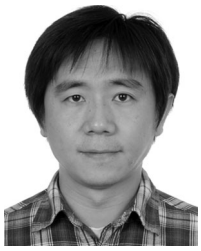
[37] S.Y. Park, D. Jung, J.U. Kang, J. Kim, and J.W. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006.

[38] D. Seo and D. Shin, "Recently-Evicted-First Buffer Replacement Policy for Flash Storage Devices," *IEEE Trans. Consumer Electronics*, vol. 54, no. 3, pp. 1228-1235, Aug. 2008.
[39] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki, "Evaluating and Repairing Write Performance on Flash Devices," *Proc. ACM Fifth Int'l Workshop Data Management on New Hardware (DaMoN '09)*, pp. 9-14, 2009.
[40] X. Tang and X. Meng, "ACR: An Adaptive Cost-Aware Buffer Replacement Algorithm for Flash Storage Devices," *Proc. Int'l Conf. Mobile Data Management (MDM)*, pp. 33-42, 2010.
[41] TPC-B, *TPC Benchmark B: Standard Specification*, http://www.tpc.org/tpcb/spec/tpcb_current.pdf, 2014..
[42] TPC-C, *TPC Benchmark C: Standard Specification*, http://www.tpc.org/tpcc/spec/tpcc_current.pdf, 2014.
[43] D.N. Tran, P.C. Huynh, Y.C. Tay, and A.K.H. Tung, "A New Approach to Dynamic Self-Tuning of Database Buffers," *Trans. Storage*, vol. 4, pp. 3:1-3:25, May 2008.
[44] D. Tsirogiannis, S. Harizopoulos, M.A. Shah, J.L. Wiener, and G. Graefe, "Query Processing Techniques for Solid State Drives," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2009.
[45] C.-H. Wu, "A Self-Adjusting Flash Translation Layer for Resource-Limited Embedded Systems," *ACM Trans. Embedded Computing System*, vol. 9, article 31, 2010.

Sai Tung On received the BEng degree in software engineering from Tsinghua University, Beijing, China. He is currently working toward the MPhil degree in the Department of Computer Science at Hong Kong Baptist University His research interests include data management on novel storage media.

Shen Gao received the BSc degree in computing studies (information systems) from Hong Kong Baptist University where he is currently working toward the MPhil degree in the Department of Computer Science. His research interests include data management for next-generation storage devices. He is a student member of the ACM.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science from the Hong Kong University of Science and Technology (2003-2008). He is currently an assistant professor in the Division of Computer Science, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests include high performance computing, distributed and parallel systems, and database systems.



Ming Wu received the bachelor's degree in computer science from the University of Science and Technology of China (1997-2002), and the PhD degree in computer science from the Institute of Computing Technology, Chinese Academy of Science (2002-2007). He is currently a researcher in the Systems Research Group, Microsoft Research Asia. His research interests include high performance computing, distributed and parallel systems, and transaction processing systems.

Qiong Luo received the PhD degree in computer sciences from the University of Wisconsin-Madison in 2002. She is currently an associate professor in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. Her research interests include database systems, parallel and distributed systems, and scientific computing.

Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998, and the PhD degree in computer science from the Hong Kong University of Science and Technology in 2002. He is currently an associate professor in the Department of Computer Science, Hong Kong Baptist University. He held visiting positions at Pennsylvania State University and Fudan University. His research interests include data management, mobile/pervasive computing, wireless sensor networks, and distributed systems. He has published more than 110 technical papers in these areas. He was a vice chairman of the ACM Hong Kong Chapter and is a senior member of the IEEE.