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 to the read-write asymmetry and the runtime workload. We evaluate FD-Buffer with trace-driven experiments on real flash disks. Our evaluation results show that our algorithm achieves up to 33% improvement on the overall performance on commodity flash disks, in comparison with the state-of-the-art flash-aware replacement policy.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Buffer management, flash disks, solid state drive, read-write asymmetry, buffer 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) [3]. 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 storage market share. IDC [4] predicted that the total flash disk volume will increase by 54.8% per year from 2008 to 2012. To optimize the performance of a database system for flash disks, we design a new buffer management algorithm, *FD-buffer*, to reduce the total cost of accessing 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 identical (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 [1]) selects the victim page that will not be needed for the longest time in the future.

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. Even worse, a recent study [2] 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.

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. 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.

Our FD-Buffer is designed to reduce the I/O cost on flash disks. FD-Buffer divides the buffer pool into two parts: one for clean pages and the other for dirty pages. Unlike the existing flash-aware policies such as CFLRU [12] and CFDC [11] that highly depend on a specific existing replacement policy, each pool of FD-Buffer

CIKM'10, October 26–30, 2010, Toronto, Ontario, Canada.

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 trace-driven executions on real flash disks. The workloads contain transactional processing benchmarks such as TPC-C. It is shown that FD-Buffer is 70% and 33% faster than LRU and CFDC [11], respectively.

2. RELATED WORK

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 [5, 6]. Recently, there have been several proposals of buffer management policies to address the read-write asymmetry of flash disks [12, 11]. The techniques used in these proposals can be categorized into two: giving priority to choosing clean pages as victims over dirty pages [12, 11, 8], and improving the locality of writes [11]. The first category of techniques is more relevant to our study. CCF-LRU [8] considers the access frequency for clean pages in their clean-page first policy. CFLRU (Clean-First LRU) [12] 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 firstly with clean pages in the clean-first region, then with dirty pages in the clean-first region and finally with pages in the working region.

Write clustering has been considered an effective technique in reducing the total cost [11, 13, 7]. It combines multiple writes with proximity into a single write request to the flash disk. CFDC [11] enhances CFLRU by clustering dirty pages and evicting them as a batch. Similar techniques are used in buffer management [13], or write I/O [7].

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. 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). Eq. (1) gives the average I/O cost:

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

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 costs for reading and writing a page from the flash disk, respectively.

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. (1) by C_{read}, we obtain the *normalized average I/O cost* in Eq. (2):

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

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 1 gives an example to show that Beledy's algorithm [1] 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 $\mathbb{R} < 2$, Beledy's algorithm wins. They have the same cost when $\mathbb{R} = 2$. If $\mathbb{R} > 2$, the alternative algorithm wins.

	Belady [1]			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 1: Examples of Belady's algorithm and FD-Buffer

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 still be 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.

4.1 Overview

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 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 *adaptively* 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. The adaptation is made in a cost-based way combining the asymmetry factor and workload localities. Due to the limited space, we focus on the replacement policy in the buffer manager, and refer the reader

to the technical report [10] for the details on the cost estimation and adaptation.

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.

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, since it is widely used and evaluated in the traditional buffer management.

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;

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 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 the page in the frame; (2) placing a new page into the frame and writing in the page. In contrast, a frame moves from the dirty pool to the clean pool when the dirty page in the frame is replaced by a new page for reading. 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 1. 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.

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;

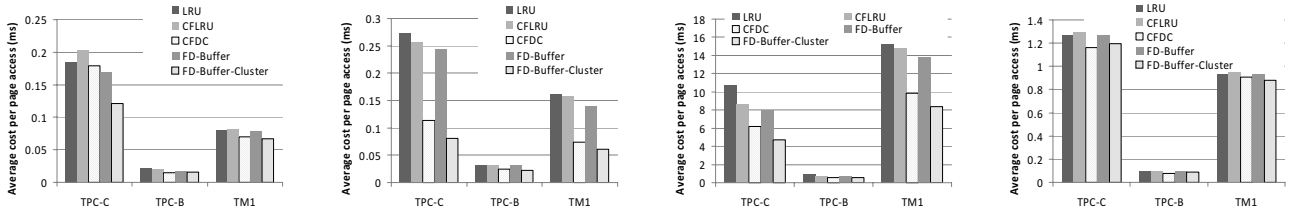
5. EVALUATION

In this section, we mainly present the results on evaluating FD-Buffer in comparison with LRU and CFLRU on real flash disks. More detailed results including simulations and the evaluation on the effectiveness of our adaptation in FD-Buffer are presented in our technical report [10].

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 has a speed of 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. 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. We use three benchmarks on transactional processing, namely TM1 [9], TPC-B, and TPC-C. 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.



(a) Intel SSD (b) Mtron SSD (c) SD card (d) Hard disk

Figure 1: The average time per page access of FD-Buffer on benchmarks with three flash disks and the hard disk.

	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

Table 2: Specification on the traces in the experiment

To evaluate FD-Buffer on real flash disks, we implement a buffer manager, taking a trace as input, and performing I/O requests to the flash disk. Thus, we can obtain the average response time for a buffer page request on the flash disk.

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 [11], as the representatives for traditional disk-based and flash-based replacement policies, respectively. We also evaluate the write clustering technique for both FD-Buffer and CFDC [11] on flash disks. 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.

5.2 Results on Real Disks

Figure 1 shows the average time per page access of the replacement policies on the three flash disks and the hard disk. We explicitly configure the buffer size to be smaller than the database size in order to exercise disk I/O operations. The ratio of buffer pool size to the database size is fixed to be 3.1%. 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 hardware settings), which enables FD-Buffer(with write clustering) to outperform CFDC by up to 33%.

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 proposes a new buffer management algorithm FD-Buffer for flash-based databases, with a focus on addressing the read-write asymmetry and workload dynamics. Our trace-driven experimental studies show that FD-Buffer outperforms two recent flash-aware algorithms. As for future work, we are interested in 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. This work was supported in partial by RGC/GRF projects 210808 and 211510.

7. REFERENCES

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

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

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

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

[5] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD*, 2007.

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

[7] Y. Li, J. Xu, B. Choi, and H. Hu. Stablebuffer: Optimizing write performance for dbms applications on flash devices. In *CIKM*, 2010.

[8] 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.

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

[10] S. T. On, Y. Li, B. He, M. Wu, Q. Luo, and J. Xu. FD-Buffer: A buffer manager for databases on flash disks. Technical report, Microsoft Research, 2010.

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

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

[13] D. Seo and D. Shin. Recently-evicted-first buffer replacement policy for flash storage devices. *IEEE Transactions on Consumer Electronics*, 54(3):1228–1235, Oct. 2008.