AD-LRU: An Efficient Buffer Replacement Algorithm for Flash-Based Databases

Peiquan Jin[a], Yi Ou[b], Theo Härder[b], Zhi Li[a]

[a] *School of Computer Science and Technology, University of Science and Technology of China*
[b] *Department of Computer Science, University of Kaiserslautern, D-67663 Kaiserslautern, Germany*

Abstract

Flash memory has characteristics of out-of-place update and asymmetric I/O latencies for read, write, and erase operations. Thus, the buffering policy for flash-based databases has to consider those properties to improve the overall performance. This article introduces a new approach to buffer management for flash-based databases, called AD-LRU (Adaptive Double LRU), which focuses on improving the overall runtime efficiency by reducing the number of write/erase operations and by retaining a high buffer hit ratio. We conduct trace-driven experiments both in a simulation environment and in a real DBMS, using a real OLTP trace and four kinds of synthetic traces: random, read-most, write-most, and Zipf. We make detailed comparisons between our algorithm and the best-known competitor methods. The experimental results show that AD-LRU is superior to its competitors in most cases.

Keywords: flash memory, database, buffer management, replacement policy, flash-based DBMS

1. Introduction

1.1. Problem Statement

In recent years, flash memory greatly gained acceptance in various embedded computing systems and portable devices such as PDAs (personal digital assistants), HPCs (handheld PCs), PMPs (portable multimedia players), and mobile phones because of low cost, volumetric capacity, shock resistance, low-power consumption, and non-volatile properties [1, 2, 3]. Among the two types of flash memory, NAND and NOR, the NAND flash memory is more often used for mass-storage devices[1]. In the following text, we just use the term *flash memory* or *flash* to indicate the *NAND flash memory*.

[1] In some special use cases, e. g., in PMPs, it is even desirable to store program code on NAND flash [4], but, in this article, we consider the more common use of flash as a data storage device.

However, flash memory has many properties differing from magnetic disk, e.g. write-once, block erasure, asymmetric read/write speed and limited block erase count. Flash memory usually consists of many blocks and each block contains a fixed set of pages [5]. Read/write operations are performed on page granularity, whereas erase operations use block granularity. Write/erase operations are relatively slow compared to read operations. Typically, write operations are about ten times slower than read operations, and erase operations are about ten times slower than write operations [6]. Data in a page cannot be updated in-place, i.e., when some data in a page has to be modified, the entire page must be written into a free page slot and the old page content has to be invalidated. Hence, flash always requires *out-of-place updates*. Furthermore, updating a page will cause costly erase operations performed by some garbage collection policy [7], in case that no enough free pages exist in flash memory. Hence, increasing the number of writes will accompany even more erase operations, as shown in previous experimental studies [6].

Since flash memory has become a serious disk alternative, traditional DBMSs should support flash storage devices and provide efficient techniques to cope with flash I/O properties. Among those techniques, DBMS buffering has first received much attention from the research community because of its effectiveness in reducing I/O latencies and thus improving the overall DBMS performance. Traditional (magnetic-disk-based) buffering algorithms do not consider the differing I/O latencies of flash memory, so their straight adoption would result in poor buffering performance and would demote the development of flash-based DBMSs [8]. The use of flash memory requires new buffer replacement policies considering not only buffer hit ratios but also replacement costs incurring when a dirty page has to be propagated to flash memory to make room for a requested page currently not in the buffer. As a consequence, a replacement policy should minimize the number of write and erase operations on flash memory and, at the same time, avoid to worsen the hit ratio which otherwise would lead to additional read operations.

The above flash challenges of DBMS buffering are not met by traditional buffer replacement algorithms. Most of them focus on hit-ratio improvement alone, but not on write costs caused by the replacement process. Recently, LRU-WSR [6] and CFLRU [8] were proposed as the new buffering algorithms for flash-based DBMSs. These algorithms favor to first evict clean pages from the buffer so that the number of writes incurring for replacements can be reduced. Of course, this is a very important idea concerning flash DBMS buffering, and we will also partially observe this policy but with a critical revision of the replacement process. However, CFLRU and LRU-WSR do not exploit the frequency of page references, which will result in an increase of both write count and runtime. Furthermore, since both algorithms exploit the LRU concept and use only a single LRU queue, both are not scan-resistant [9], i.e., the buffer will be totally polluted with sequentially referenced pages when some kind of scan operation is performed.

1.2. Our Contributions

In this article, we present an efficient buffer replacement policy for flash-based DBMSs, called AD-LRU (Adaptive Double LRU), which focuses on reducing the number of write/erase operations as well as maintaining a high buffer hit ratio. The specific contributions of our article can be summarized as follows:

(1) We present the novel AD-LRU algorithm for the buffer management of flash-based DBMSs (see Section 3), which not only considers the frequency and recency of page references but also takes into account the imbalance of read and write costs of flash memory when replacing pages. Moreover, AD-LRU is self-tuning to respond to changes in reference patterns, as frequency and recency of page references may fluctuate in varying workloads.
(2) We run experiments both in a flash simulation environment and in a real DBMS to evaluate the efficiency of AD-LRU by using different types of workloads (see Section 4). The experimental results show that AD-LRU maintains a higher hit ratio for the Zipf workload, whereas, for the other workloads, its results are comparable to those of the three competitor algorithms. In both types of experiments, our algorithm outperforms them considering both write count and overall runtime.

1.3. A Brief Outline of the Paper

The remainder of this article is organized as follows: In Section 2, we sketch the related work. In Section 3, we present the basic concepts of the AD-LRU approach to flash DBMS buffering. Section 4 describes the details about the experiments and the performance evaluation results. Finally, Section 5 concludes the article and outlines our future work.

2. Related Work

In this section, we briefly introduce flash storage systems (see Section 2.1) and then review the replacement algorithms for magnetic-disk-based (see Section 2.2) and those for flash-based DBMSs (see Section 2.3).

2.1. Flash Memory and Flash Storage System

Flash memory is a type of EEPROM, which was invented by Intel and Toshiba in 1980s. Unlike magnetic disks, flash memory does not support update in-place, i.e., previous data must be first erased before a write can be initiated to the same place. As another important property of flash memory, three types of operations can be executed: read, write, and erase. In contrast, magnetic disks only support read and write operations. Moreover, all granularities and latencies of read and write operations differ for both device types.

Compared to magnetic disks, flash memory has the following special properties:

(1) It has no mechanical latency, i.e., seek time and rotational delay are not present.

(2) It uses an out-of-place update mechanism, because update in-place as used for magnetic disks would be too costly.
(3) Read/write/erase operations on flash memory have different latencies. While reads are fastest, erase operations are slowest. For example, a MICRON MT29F4G08AAA flash chip needs 25 μs/page, 220 μs/page, 1.5 ms/block for the read/write/erase latencies, respectively [10].
(4) Flash memory has a limited erase count, i.e., typically 100,000 for SLC-based NAND flash memory. Read/write operations become unreliable when the erase threshold is reached.

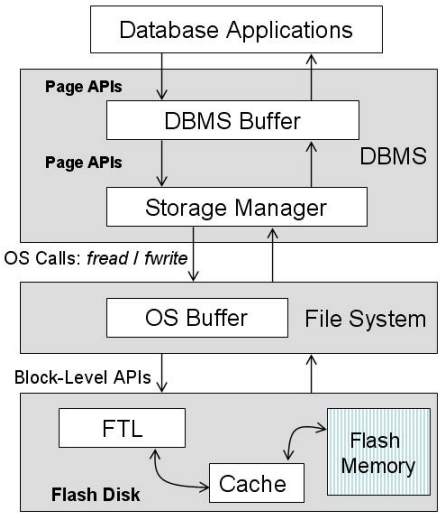


Figure 1: Simple architecture of a flash-based storage system

NAND flash memory can be categorized into two types, which are Single-Level-Cell (SLC) and Multi-Level-Cell (MLC) flash memory. SLC stores one bit in a memory cell, while MLC represents two or more bits in a memory cell. SLC is superior to MLC both in read/write performance and durability. On the other hand, MLC can provide larger capacity with lower price than SLC [11]. However, the special features of SLC/MLC are usually transparent to file and database systems, because most flash-based storage devices use a flash translation layer (FTL) [12, 13] to cope with the special features of flash memory, which maps logical page addresses from the file system to physical page addresses used in flash memory devices. FTL is very useful because it enables a traditional DBMS to run on flash disks without any changes to its kernel. In other words, A DBMS-level algorithm, e.g., buffer management, is independent of the fundamental SLC or MLC flash chips, due to the FTL layer.

Figure 1 shows the typical architecture of a flash-based storage system using FTL. It indicates that the file system regards flash disks as block devices. Page rewrites and in-place updates can be logically done at the file system layer.

However, updating a page will cause a physical rewriting to a different page or even a different block. Thus reducing the page rewrites at the file system layer is helpful to reduce the number of physical write and erase operations.

2.2. Traditional Buffer Replacement Algorithms

Buffer management is one of the key issues in DBMSs. Typically, we assume a two-level storage system: main memory and external (secondary) storage. Both of them are logically organized into a set of pages, where a page is the only interchanging unit between the two levels. When a page is requested from modules of upper layers, the buffer manager has to read it from secondary storage if it is not already contained in the buffer. If no free buffer frames are available, some page has to be selected for replacement. In such a scheme, the quality of buffer replacement decisions contributes as the most important factor to buffer management performance.

Traditional replacement algorithms primarily focus on the hit ratio [9], because a high hit ratio will result in a better buffering performance. Many algorithms have been proposed so far, either based on the recency or frequency property of page references. Among them, the best-known ones are LRU, CLOCK [14], LRU-2 [15], and ARC [9].

LRU always evicts the least-recently-used page from an LRU queue used to organize the buffer pages ordered by time of their last reference. It always replaces the page found at the LRU position. An important advantage of LRU is its constant runtime complexity. Furthermore, LRU is known for its good performance in case of reference patterns having high temporal locality, i.e., currently referenced pages have a high re-reference probability in the near future. But LRU also has severe disadvantages. First, it only considers the recency of page references and does not exploit the frequency of references. Second, it is not scan-resistant, i.e. a scan operation pollutes the buffer with one-time referenced pages and possibly evicts pages with higher re-reference probability.

CLOCK uses a reference bit which is set to 1 whenever the page is referenced [14]. Furthermore, it organizes the buffer pages as a circle, which guides the page inspection when a victim is searched. When a page currently inspected has a 1 in the referenced bit, it is reset to 0, but not replaced. The first page found having referenced bit 0 is used as the victim. Obviously, CLOCK does not consider reference frequency. Moreover, it is not scan-resistant, too. The improved CLOCK algorithm, called GCLOCK, can be tailored to workload characteristics by adding a reference count to each buffer page and using page-type-specific weights (as parameters). But experimental results have shown that its performance is highly sensitive to the chosen parameter configuration [16] and that it can be even worse than LRU [17].

LRU-2 is an improvement of LRU, as it captures both recency and approximate frequency of references [15]. It considers the two most-recent references of each page to determine the victim for replacement. Though LRU-2 tends to have a better hit ratio than LRU, its runtime complexity is higher than that of LRU. LRU-2 has to maintain the history of reference in a priority queue, which is controlled by a tunable parameter called *Correlated Information Period* (CIP).

Unfortunately, the (manual) choice of the CIP parameter crucially affects the LRU-2 performance [9]. Experiments have revealed that no single parameter setting works universally well for all buffer sizes and differing workloads.

There are other replacement algorithms that consider both recency and frequency [9], such as LRFU [18], 2Q [19], and FBR [20]. However, they have similar shortcoming as LRU-2, i.e., they all are not self-tuning for different buffer sizes and workloads.

The ARC algorithm is an adaptive buffer replacement algorithm [9], which utilizes both frequency and recency. Moreover, ARC is scan-resistant and has proven a superior performance than other replacement algorithms. It maintains two LRU lists: L_1 and L_2. The L_1 list stores the pages which are referenced only once, while the L_2 list contains those pages accessed at least twice. The ARC algorithm adapts the sizes of the two lists to make the replacement algorithm to suit different buffer sizes and workloads.

All the traditional algorithms mentioned above do not consider the asymmetric I/O properties of flash memory. Yet, reducing the write/erase count for buffer management of flash-based DBMSs is not only useful to improve the runtime, but also helpful to extend the life cycle of flash memory. Hence, buffer replacement algorithms focusing on the avoidance of write operations have been investigated in recent years.

2.3. Buffer Replacement Algorithms for Flash-based DBMSs

To our knowledge, CFLRU is the first algorithm designed for flash-based DBMSs [8]. It modified the LRU policy by introducing a clean-first window W, which starts from the LRU position and contains the least-recently-used $w \cdot B$ pages, where B is the buffer size and w is the ratio of the window size to the total buffer size. When a victim is selected, CFLRU first evicts the least-recently-used clean page in W. Hence, it reduces the number of write operations, because a clean page is not propagated to flash memory. If no clean page is found in W, CFLRU acts according to the LRU policy. However, the following problems occur:

(1) Its clean-first window size has to be tuned to the current workload and can not suit differing workloads. For this reason, [8] proposed a method to dynamically adjust w, which periodically computes the ratio of writes to reads and decides to increase or decrease w. However, it is not sufficient in real scenarios, because the locality of references as well as the buffer size have substantial impact on the optimality of w.
(2) It always replaces clean pages, which causes cold dirty pages residing in the buffer for a long time and, in turn, resulting in suboptimal hit ratios.
(3) It has to search the clean-first window during each page replacement, which brings additional runtime costs.
(4) Just like LRU, CFLRU is also not scan-resistant and does not exploit frequency of references.

Based on the CFLRU policy, Yoo et al. presented a number of upgrades to the CFLRU algorithm, called CFLRU/C, CFLRU/E, and DL-CFLRU/E [21].

These algorithms explored strategies for evicting pages based on lowest write reference count, lowest block erase count, frequency of access and wear-leveling degree. In the CFLRU/C and CFLRU/E algorithms, the buffer list structure is the same as that of CFLRU, the least-recently-used clean page is selected as the victim within the pre-specified window of the LRU list. If there is no clean page within the window, CFLRU/C evicts the dirty page with the lowest write reference count, while CFLRU/E evicts the dirty page with the lowest block erase count. DL-CFLRU/E maintains two LRU lists called clean page list and dirty page list, and first evicts a page from the clean page list. If there is no clean page in the clean page list, DL-CFLRU/E evicts the dirty page with the lowest block erase count within the window of the dirty page list.

Unlike CFLRU, CFLRU/C only delays to flush dirty pages with a high write reference count, which can effectively reduce the number of write operations and hence the number of erase operations to some extent. However, it does not consider the reference frequency, just like the original CFLRU algorithm. The CFLRU/E and DL-CFLRU/E algorithms need to know exactly the erase count of each block, which is usually hidden in the FTL layer and can not be obtained by upper-layered buffer replacement algorithms. Therefore, the CFLRU/E and DL-CFLRU/E algorithms do not fit for DBMS applications.

CFDC [22] and CASA [23] are two more improvements of the CFLRU algorithm. CFDC improves the efficiency of the buffer manager by flushing pages in a clustered fashion, based on the oberservation that flash writes with strong spatial locality can be served by flash disks more efficiently than random writes. CASA improves CFLRU by automatically adjusting the size of buffer portions allocated for clean pages and dirty pages according to the storage device's read/write cost ratio.

The LRU-WSR policy in [6] considers the cold/hot property of dirty pages, which is not tackled by the CFLRU algorithm. LRU-WSR always tries to remain hot dirty pages in the buffer and first replaces clean pages or cold-dirty pages. The main difference between CFLRU and LRU-WSR is that the latter considers the eviction of dirty cold pages. Hence, dirty cold pages will not reside in the buffer as long as in the case of CFLRU.

LRU-WSR has a high dependency on the write locality of workloads. It shows poor performance in case of low write locality which may cause dirty pages to be quickly evicted. The LRU-WSR policy also does not capture the frequency of references, which may degrade the hit ratio. Like LRU, LRU-WSR is also not scan-resistant.

The LIRS-WSR algorithm [24] is an improvement of LIRS [25] so that it can suit the requirements of flash-based DBMSs. However, LIRS-WSR has the same limitation as CFLRU and LRU-WSR, because it is not self-tuning, too, and hardly considers the reference frequency. Frequently referenced pages may be evicted before a cold dirty page, because a dirty page is always put on the top of the LIRS stack, irrespective of its reference frequency. Moreover, LIRS-WSR needs additional buffer space, because it has to maintain historical reference information for those pages that were referenced previously, but are currently not in the buffer.

	LRU	*CFLRU*	*LRU-WSR*	*CCF-LRU*	*CFLRU/C*	*AD-LRU*
Recency	√	√	√	√	√	√
Cleanness[2]		√	√	√	√	√
Frequency				partial		√

Table 1: Motivation of AD-LRU

The CCF-LRU algorithm proposed in [26] used two LRU queues, a cold clean queue and a mixed queue, to maintain buffer pages. The cold clean queue stores those cold (first-referenced) clean pages, while the mixed queue stores dirty pages or hot clean pages. It always selects a victim from the cold clean queue, and if the cold clean queue is empty, it employs the same policy as LRU-WSR to select a dirty page from the mixed queue. This algorithm focused on the reference frequency of clean pages and has little consideration of the reference frequency of dirty pages. Besides, the CCF-LRU algorithm has no techniques to control the length of the cold clean queue, which will lead to frequent evictions of recently-read pages in the cold clean queue and lower the hit ratio, because, when its size is small, a new read page will be evicted out very quickly. On the contrary, our proposed AD-LRU algorithm takes into account both the reference frequency of clean pages and dirty pages and has a new mechanism to control the length of the cold queue to avoid a drop in the hit ratio. Our experimental results also demonstrate that AD-LRU has better performance than CCF-LRU.

3. The AD-LRU Algorithm

In this section, we present the basic idea and details of AD-LRU.

3.1. Basic Idea

For buffer replacement algorithms of flash-based DBMSs, we need to consider not only the hit ratio but also the write costs. It has been proven that the idea to evict clean pages first reduces the number of write operations and, thus, the overall runtime [8]. However, traditional algorithms such as LRU will not always evict a clean page. On the other hand, the flash-aware algorithms CFLRU and LRU-WSR likely evict hot clean pages from the buffer, because they both use the clean-first strategy and do not take the page reference frequency into account. In such cases, the hit ratio may degrade as confirmed by previous experiments [8, 6].

Therefore, we explicitly enhance the traditional LRU policy by frequency considerations and first evict least-recently- and least-frequently-used clean pages to reduce the write count during the replacement. Our aim is to reduce the write

[2]Cleanness means that the algorithm distinguish between clean pages and dirty pages when evicting pages from the buffer.

costs of the buffer replacement algorithm while keeping a high hit ratio. Table 1 shows a simple description of the current buffer replacement algorithms for flash-based DBMSs. It also presents the motivation of our AD-LRU algorithm, which tries to integrate the properties of recency, frequency, and cleanness into the buffer replacement policy.

The AD-LRU concepts can be summarized as follows:

(1) We use two LRU queues to capture both the recency and frequency of page references, among which one cold LRU queue stores the pages referenced only once and the hot LRU queue maintains the pages referenced at least twice.

(2) The sizes of the double LRU queues are dynamically adjusted according to the changes in the reference patterns. We increase the size of the hot LRU queue and decrease the size of the cold one when a page in the cold queue is re-referenced. The hot queue shrinks when a page is selected as victim and moved from there to the cold queue.

(3) During the eviction procedure, we first select the least-recently-used clean page from the cold LRU queue as the victim, for which a specific pointer FC is used (see Figure 2). If clean pages do not exist in the cold LRU queue, we use a second-chance policy [6] to select a dirty page as the victim. For this reason, each page in the double LRU queues is marked by a referenced bit, which is always set to 1, when the page is referenced (see Section 2.2). Hence, the second-chance policy ensures that dirty pages in the cold LRU queue will not be kept in the buffer for an overly long period.

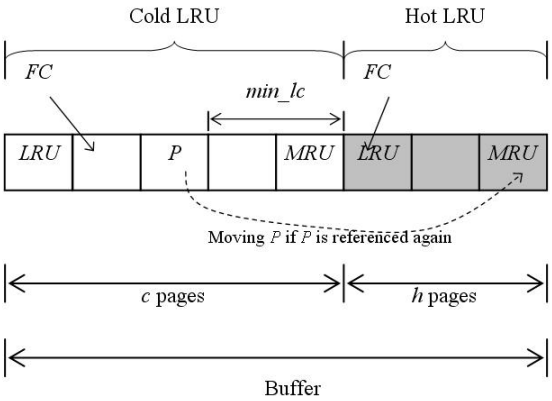


Figure 2: Double LRU queues of the AD-LRU algorithm

As illustrated by Figure 2, the double LRU queues of AD-LRU separate the cold and hot pages in the buffer, where the cold and hot LRU queues contain c and h pages, respectively. The values of c and, at the same time, h are dynamically adjusted according to the reference patterns. When a page P is first referenced, it is put in the cold queue and becomes the MRU element.

When it is referenced again, it is moved into the hot queue and put at the MRU The parameter min_lc in Figure 2 sets the lowest limit for the size of the cold queue. It means if the size of the cold queue reaches min_lc, we will evict pages from the hot queue rather than from the cold one. The reason to introduce this parameter is that a cold queue too small would likely be filled and, consequently, would result in frequent replacements in it, because pages requested from external stores always arrive at the cold queue and, in turn, we always select a victim from this queue at first.

The FC (First-Clean) position indicates the least-recently-used clean page in the LRU queue. When selecting the victim, we directly choose the FC page in the cold LRU queue if FC is valid, or we choose a dirty page from the queue using a second-chance policy. If the size of the cold queue reaches min_lc, we select the victim from the hot queue using a similar process, i.e., we first choose the FC page or, if FC is null, we select a dirty page based on the second-chance policy.

3.2. AD-LRU Page Eviction

The page fetching algorithm is characterized as follows (see Algorithm *AD-LRU_fetch*). If the requested page is found in the hot LRU queue, we just move the page to the MRU position (line 1 to line 5). If the page is found in the cold LRU queue, we enlarge the hot queue, thereby automatically reduce the cold queue, and move the page to the MRU position in the hot queue (line 6 to line 12). If a page miss occurs and the buffer has free space, we increase the size of the cold queue and put the fetched page into the cold queue (line 14 to line 19). If the buffer is full, then we have to select a page for replacement. If the cold LRU queue contains more than min_lc pages, we evict the victim from the cold queue (line 22), otherwise, we evict it from the hot queue and reduce the hot queue thereby enlarging the cold queue (line 24 to line 25). When a page is referenced, its referenced bit is set to 1, which will be used in the $SelectVictim$ routine to determine the victim based on the second-chance policy.

The algorithm $SelectVictim$ first selects the FC page (least-recently-used clean page) from the LRU queue as the victim. If no clean pages exist in the queue, it selects a dirty page using the second-chance policy. The referenced bit of the buffer page under consideration is checked and, if it is 1, we move the page to the MRU position and set the referenced bit to 0; this inspection is continued until the first page with referenced bit having 0 is located, which is then returned as the result.

Fig. 3 shows an example of how the SelectVictim algorithm works. In this example, we suppose that there are six pages in the buffer and the buffer is full. When the buffer manager receives a new page reference, our AD-LRU algorithm will choose page 4 as the victim, as shown in Fig. 3. On the other side, Fig. 4 shows the victims selected by the CFLRU algorithm and the LRU-WSR algorithm. Compared with CFLRU, which selects the hot clean page 2 in the example, AD-LRU will choose a cold clean page so as to achieve a higher hit ratio. Compared with LRU-WSR, which selects the cold dirty page 2, AD-LRU has fewer write operations, because it avoids to evict dirty pages.

Algorithm 1: AD-LRU_fetch

```
   data : L_c: the cold queue containing c pages, initially c = 0; L_h: hot
          queue containing h pages, initially h = 0
   result: return a reference to the requested page p
 1 if p is in L_h then // p is in the hot queue
 2 |   move p to the MRU position in L_h;
 3 |   adjust FC in L_h to let FC point to the least-recently-used clean page
   |   in L_h ;
 4 |   Ref(p) = 1;
 5 |   return a reference to p in L_h;
 6 else if p is in L_c then // p is in the cold queue
   |   // adjust the hot/cold queues
 7 |   h++; c--;
   |   // p becomes hot
 8 |   move p to the MRU position in L_h;
 9 |   adjust FC in L_c to let FC point to the least-recently-used clean page
   |   in L_c ;
10 |   Ref(p) = 1;
11 |   adjust FC in L_h to let FC point to the least-recently-used clean page
   |   in L_h ;
12 |   return a reference to p in L_h;
13 else // p is not in the buffer
14 |   if there is free space in the buffer then
15 |   |   c++; put p in L_c;
16 |   |   adjust L_c by putting p into the MRU position;
17 |   |   adjust FC in L_c to let FC point to the least-recently-used clean
   |   |   page in L_c ;
18 |   |   Ref(p) = 1;
19 |   |   return a reference to p in L_c;
20 |   else
21 |   |   if c > min_lc then
22 |   |   |   victim = SelectVictim (L_c);
23 |   |   else// cold queue is too small, replace L_h
24 |   |   |   victim = SelectVictim (L_h) ;
   |   |   |   // adjust the cold/hot regions
25 |   |   |   h--; c++;
26 |   |   if victim is dirty then // write to flash
27 |   |   |   WriteDirty (p);
28 |   |   put p into a free frame in L_c;
29 |   |   adjust L_c by putting p into the MRU position;
30 |   |   Ref(p) = 1;
31 |   |   return a reference to p in L_c;
32
```

Algorithm 2: SelectVictim

```
   data : LRU queue L
   result: return a reference to the victim page
 1 if FC of L is not null then
       // select the first clean page
 2     remove the FC page from L;
 3     adjust the FC position in L;
 4     return a reference to the FC page;
   // select a dirty page using the second-chance policy
   // starting from the LRU position
 5 victim = L.first;
 6 while Ref(victim) = 1 do
 7     move victim to the MRU position in L;
 8     Ref(victim) = 0;
       // continue to check the LRU position
 9     victim = L.first;
10 remove victim from L;
11 return a reference to the victim;
```

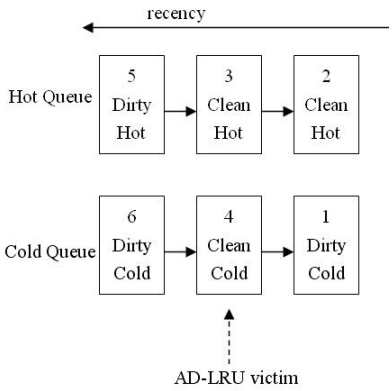


Figure 3: Example of AD-LRU victim selection

The AD-LRU algorithm has the same overall goal as CFLRU and LRU-WSR; however, it is designed in a new way. The differences between AD-LRU and those approaches are listed as follows:

(1) AD-LRU considers reference frequency, an important property of reference patterns, which is more or less ignored by CFLRU and LRU-WSR. Therefore, we anticipate a superior performance for AD-LRU, especially when high reference locality is present, as confirmed by our experiments shown in Section 4.

(2) Cold-dirty pages may reside in the buffer under CFLRU for an overly long

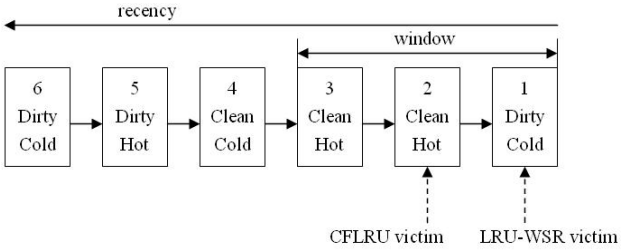


Figure 4: Victims selected by CFLRU and LRU-WSR

period, whereas AD-LRU purges the buffer from the cold pages as soon as appropriate.

(3) AD-LRU is self-tuning. The sizes of cold and hot LRU queues can be dynamically adjusted to the workloads, while the clean-first window size of CFLRU has to be statically determined.

(4) AD-LRU is scan-resistant, a property missing in CFLRU and LRU-WSR. Under AD-LRU, a scan only influences the cold queue in most cases, while the frequently accessed pages in the hot queue will not be evicted.

3.3. Considerations on Checkpointing

The concept of checkpointing requires a DBMS to periodically or incrementally force dirty pages out to non-volatile storage for fast recovery, regardless of their access frequency or access recency. To create a checkpoint at a safe place, earlier solutions flushed all modified buffer pages thereby achieving a transaction-consistent or action-consistent firewall for redo recovery on disk. Such direct checkpoints are not practical anymore, becausegiven large database buffer sizesthey would repeatedly imply quite a long time of flushing out all buffer pages. Today, the method of choice is fuzzy checkpointing [27], where only logs describing the checkpoint are written to the log. The logs can help to determine which pages containing committed data were actually in the buffer at the moment of a crash, with two or three write operations [28]. As a consequence, fuzzy checkpointing does not require that any dirty page be forced to non-volatile storage when a checkpoint is created. The dirty pages in the buffer will be flushed to non-volatile storage via asynchronous I/O actions not linked to any specific point in time. Clearly, fuzzy checkpointing will not affect the hit ratio of a buffer replacement algorithm, because hot dirty pages will remain in the buffer after being committed into non-volatile storage. On the other side, more write operations will be introduced by checkpoints. Note that this additional write overhead is almost the same for different buffer replacement algorithms. Thus, the write count of our AD-LRU algorithm as well as other buffer replacement policies will increase in DBMSs supporting checkpoints, but generally AD-LRU will still keep its performance advantages compared with other algorithms such as CFLRU and LRU-WSR.

4. Performance Evaluation

In this section, we compare AD-LRU with five competitor algorithms, i.e., LRU, CFLRU, LRU-WSR, CFLRU/C, and CCF-LRU. We do not compare the CFLRU/E and DL-CFLRU/E algorithms in our experiment, because those two algorithms need to know the erase count of blocks in flash disks, which can not be realized in a upper-layered buffer manager. We perform the experiments both in a simulation environment and in a real DBMS, where different types of workloads are applied. The simulation experiment is performed to test the hit ratio and write count of each algorithm, whereas the DBMS-based experiment aims at the comparison of the overall runtime.

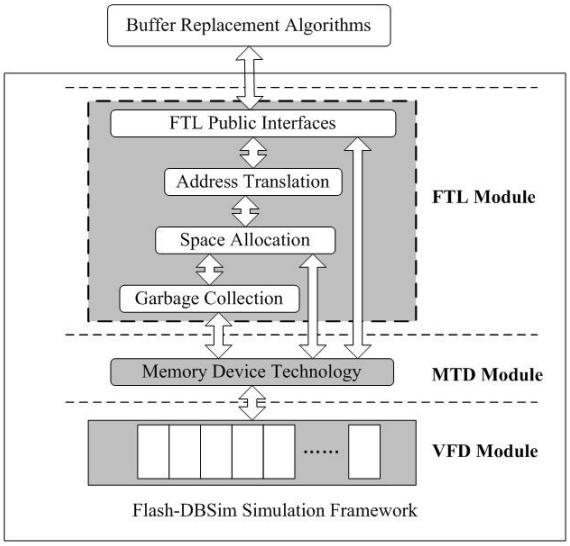


Figure 5: Flash-DBSim architecture

4.1. Simulation Experiment

Experiment setup. The simulation experiments are conducted based on a flash memory simulation platform, called Flash-DBSim [29, 30]. Flash-DBSim is a reusable and reconfigurable framework for simulation-based evaluation of algorithms on flash disks, as shown in Figure 5. The VFD module is a software layer that simulates the actual flash memory devices. Its most important function module is to provide virtual flash memory using DRAM or even magnetic disks. It also provides manipulating operations over the virtual flash memory, such as page reads, page writes, and block erases. The MTD module maintains a list of different virtual flash devices, which enables us to easily manipulate different types of flash devices, e.g., NAND, NOR, or even hybrid flash disks. The FTL module simulates the virtual flash memory as a block device, so that the

Attribute	Value
Page Size	2,048 B
Block Size	64 pages
Read Latency	25 μs/page(MAX)
Write Latency	220 μs/page
Erase Latency	1.5 ms/block
Endurance	100,000

Table 2: MICRON MT29F4G08AAA flash chip characteristics [10]

upper-layer applications can access the virtual flash memory via block-level interfaces. The FTL module employs the EE-Greedy algorithm [31] in the garbage collection part and uses the threshold for wear-levelling proposed in [32]. In one word, Flash-DBSim can be regarded as a reconfigurable SSD (solid state disk). It exhibits different SSD properties for upper layers, e.g., buffer manager and index. In the simulation experiment, we refer to the MICRON MT29F4G08AAA flash chip in the Flash-DBSim. The detailed parameters of the selected flash chips are listed in Table 2.

Workloads. We use four types of synthetic traces in the simulation experiment, i.e., random trace, read-most trace (e.g., of decision support systems), write-most trace (e.g., of OLTP systems), and Zipf trace [33]. For the Zipf trace, the probability of accessing the i^{th} page among a totality of N pages, P_i, is given by the following expression:

$$P_i = \frac{1}{H_N^{1-\theta} \cdot i^{1-\theta}} \tag{1}$$

Here, $\theta = \log a / \log b$ and H_N^s is the N^{th} harmonic number of order s, namely $1^{-s} + 2^{-s} + \ldots + N^{-s}$. For example, if a is 0.8 and b is 0.2, the distribution means that eighty percent of the references deal with the most active twenty percent of the pages. Such a referential locality is refered to as "80–20" in this article. There are total 100,000 page references in each of the first three traces, which are restricted to a set of pages whose numbers range from 0 to 49,999. The total number of page references in the Zipf trace is set to $500,000$ in order to obtain a good approximation, while the page numbers still fall in [0, 49999]. Table 3 to Table 6 show the details concerning these workloads.

Parameter setup. Parameter w of the CFLRU algorithm is set to 0.5, which means half of the buffer is used as clean-first window. As mentioned before, the hit ratio of CFLRU is affected by w. When w is close to 0, CFLRU approximates LRU. When w is close to 1, it can use the entire buffer space to store dirty pages. So a middle value 0.5 is reasonable to conduct the comparison between our algorithm and CFLRU. Parameter min_lc of AD-LRU is set to 0.1 for the Zipf trace and 0.5 for the other three traces. The impact of min_lc will be discussed late in this section. The page size is 2,048 bytes. The buffer size ranges from 512 buffer pages to 18,000 pages, i.e., from 1 MB to nearly 36 MB.

For each of the algorithms, we ran the traces shown in Table 3 to Table 6

Attribute	Value
Total Buffer Requests	100,000
Total Pages in Flash Memory	50,000
Page Size	2,048 B
Read / Write Ratio	50% / 50%
Total Different Pages Referenced	43,247
Reference Pattern	Uniform

Table 3: Simulated trace for random access

Attribute	Value
Total Buffer Requests	100,000
Total Pages in Flash Memory	50,000
Page Size	2,048 B
Read / Write Ratio	90% / 10%
Total Different Pages Referenced	43,212
Reference Pattern	Uniform

Table 4: Simulated trace for read-most access

Attribute	Value
Total Buffer Requests	100,000
Total Pages in Flash Memory	50,000
Page Size	2,048 B
Read / Write Ratio	10% / 90%
Total Different Pages Referenced	43,182
Reference Pattern	Uniform

Table 5: Simulated trace for write-most access

Attribute	Value
Total Buffer Requests	500,000
Total Pages in Flash Memory	50,000
Page Size	2,048 B
Read / Write Ratio	50% / 50%
Reference Locality	80–20
Total Different Pages Referenced	47,023

Table 6: Simulated Zipf trace (500k-50k)

and compared the hit ratios. For the Zipf trace, AD-LRU achieved the best hit ratios, as shown in Fig. 6. For the other traces, the hit ratios of all the algorithms are comparable (thus not shown), due to the (randomly generated) uniform distribution of the page references. Even for such page references, AD-LRU still outperforms the competitors in terms of write count and runtime, as shown in the following sections.

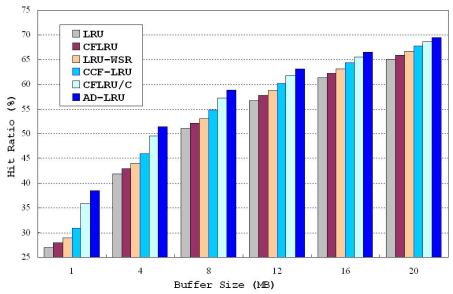


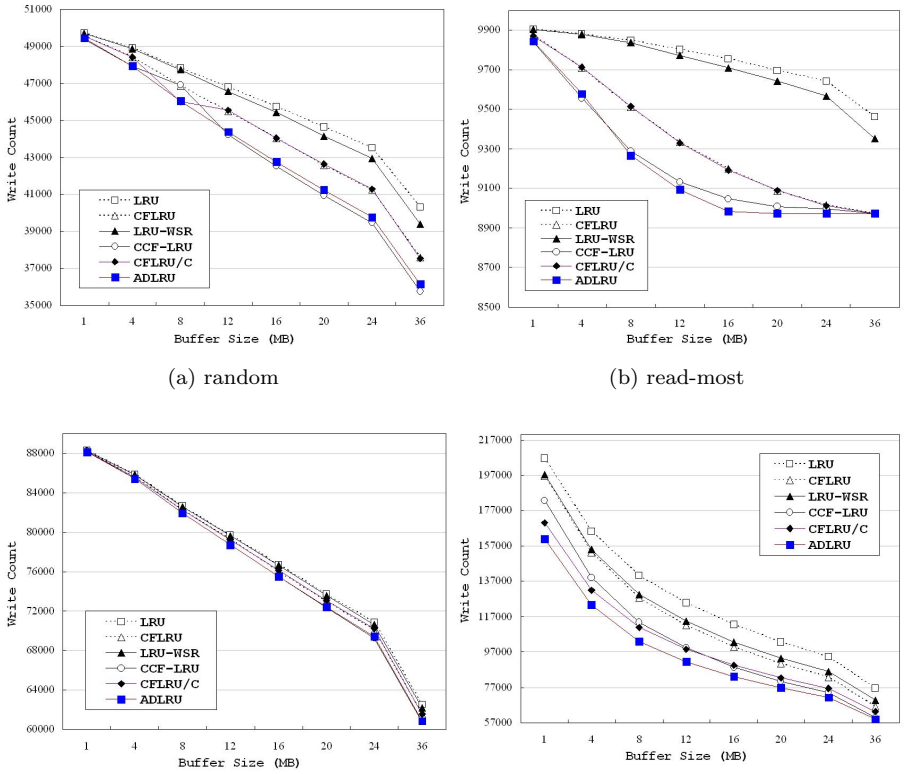
Figure 6: Hit ratios for the Zipf trace (500k-50k)

Write count. Figure 7a to Figure 7d show the number of pages propagated to flash memory. We obtained these results by counting the number of physical page writes in Flash-DBSim [30] and, at the end of each test, we flushed the dirty pages in the buffer to the flash memory to get the exact write counts. LRU generates the largest number of write operations in all cases, because it has no provisions to reduce the number of writes to flash memory. While CFLRU first replaces clean pages and keeps dirty pages for the longest time among all algorithms, it has the second smallest write count. As shown in all the four figures, AD-LRU has the smallest write count. The reason is that it divides all the buffer pages into a hot LRU queue and a cold queue, and first selects the least-recently-used clean pages from the cold queue.

In Figure 7b, AD-LRU tends to remain a stable write count when the buffer size is over 20 MB: we obtained $8,973$ in our experiment. Note our read-most trace (see Table 4) contains about 10% writes among the total references, i.e., about 10,000 updates, which should be the maximal count of possible writes in this trace. Figure 7b also indicates that AD-LRU reaches the lower bound of the write count more quickly with minimal buffer requirement. According to the write-most scenario illustrated in Figure 7c, AD-LRU still has a lower write count than the other three algorithms. However, the gap between them is not as big as shown in other figures. This is because there are more dirty pages evicted in the write-most case, as it is likely that no clean pages are found in the buffer when executing the replacement algorithm.

Erase count. Figure 8 illustrates the erase counts for all algorithms considered w.r.t. the Zipf trace listed in Table 7. The erase behavior is mostly influenced by garbage collection and wearleveling. In the Flash-DBSim environment, we use the garbage collection algorithm of [31] and the wear-leveling algorithm of [32]. We do not compare the erase counts for the traces *random*,

read-most, and *write-most*, because only few erase operations are performed when running those traces, owing to the small number of pages referenced. As Figure 8 shows, the erase counts of the buffer replacement algorithms are nearly proportional to the write counts shown in Figure 7d.



(a) random (b) read-most

(c) write-most (d) Zipf

Figure 7: Write count vs. buffer size for various workload pattern

4.2. DBMS-based Experiment

In the simulation experiment, we assume that each flash read or write has the same latency, but in real SSD-based systems, we have found that SSD's I/O latencies vary in each read/write cycle [34]. Hence, it is necessary to perform the experiment in a real DBMS to demonstrate the superior performance of our algorithm. In the DBMS-based experiment, we concentrate on the comparison of the four algorithms over a larger Zipf trace and a real OLTP trace. In particular, we focus on the write count and runtime of the involved algorithms. The comparison of hit ratios has been studied in the simulation experiment and we will not make further discussions.

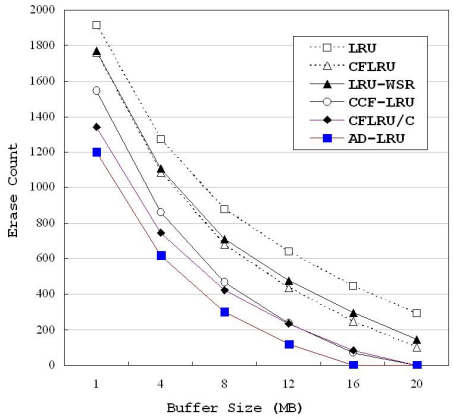


Figure 8: Erase count of the Zipf pattern for various buffer sizes

Experiment setup. The DBMS-based experiments are performed on the XTC database engine [35]. XTC is strictly designed in accordance to the well-known five-layer database architecture proven for relational DBMS implementations, so our experiments are also meaningful to the relational DBMS environment. To better explain our results, we have only used its two bottom-most layers in our experiments, i.e., the file manager supporting block-oriented access to the data files and the buffer manager serving page requests. Although designed for XML data management, the processing behavior of these two XTC layers is very close to that of a relational DBMS.

The test computer has an AMD Athlon Dual Core Processor, 512 MB of main memory, is running Ubuntu Linux with kernel version 2.6.24-19, and is equipped with a magnetic disk and a flash disk, both connected to the SATA interface used by the file system EXT2. Both OS and database engine are installed on the magnetic disk. The test data (as a database file) resides on the flash disk. The flash disk we used in the experiments is a 32 GB SLC-based Super Talent DuraDrive FSD32GC35M SSD. As discussed in Section 2.1, our algorithm does not rely on the SSD type.

In our experiments, we deactivated the file-system prefetching and the I/O scheduling for the flash disk and emptied the Linux page caches. To ensure a stable initial state, we prepared all the involved data pages in a file, which was copied to the flash disk at each execution. Furthermore, we sequentially read and wrote a 512 MB file (of irrelevant data) from and onto the flash disk before executing each algorithm.

We run our DBMS-based experiment over two types of traces. The first trace is a Zipf trace with a larger number of references (1,000,000) as well as a larger page space (100,000) than that used in the simulation experiment. The second one is a one-hour OLTP trace of a real bank system, which has also been used in LRU-2 [15] and ARC [9]. This trace contains 607,391 page references

Attribute	Value
Total Buffer Requests	1,000,000
Total Pages in Flash Memory	100,000
Page Size	2,048 B
Read / Write Ratio	51% / 49%
Reference Locality	80-20
Total Different Pages Accessed	93,870

Table 7: Zipf trace (1000k-100k)

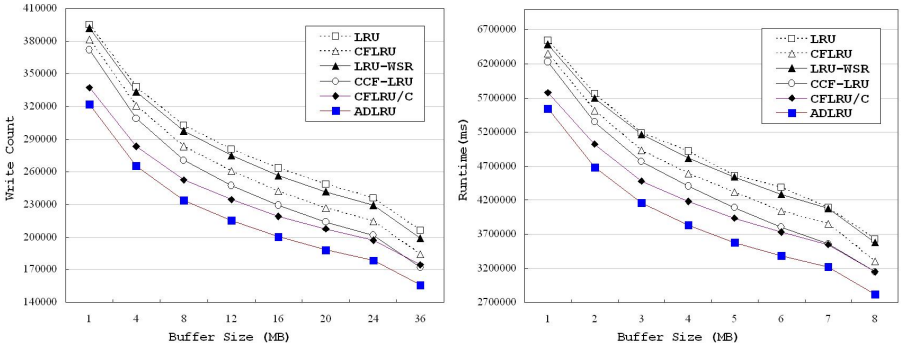
to a CODASYL database with a total size of 20 Gigabytes. The parameter w of CFLRU is set to 0.5, and the parameter min_lc of AD-LRU is set to 0.1 for all traces. The buffer page size is 2,048 bytes.

In the DBMS-based experiment, we will measure the write count and runtime of each buffer replacement algorithm mentioned in the simulation experiment. Here, the runtime of a buffer replacement algorithm is defined as follows:
Runtime = CPU time for manipulating various data structures + flash read time for missed pages + flash write time (assuming an underlying FTL) for dirty victims.
Here, flash write time contributes most to the total runtime, because flash write operations need more time than flash read operations and CPU operations. Meanwhile, flash write time has a high dependence on the underlying FTL algorithm. Although different FTL algorithms may be used in different SSDs, our goal is to compare the performance of buffer replacement algorithms under the same underlying FTL algorithm. Note that current SSD manufacturers do not report many details about the internal design of their SSDs, such as what FTL algorithm is implemented inside the SSDs.

Zipf trace experiment. Table 7 shown the details about the Zipf trace used in the DBMS-based experiment. The write count and runtime of the four algorithms over the trace are shown in Figure 9a and Figure 9b, respectively. Among all the measured algorithms, LRU has constant CPU runtime complexity. But due to its largest write count among all the algorithms, it exhibits the worst overall runtime. The CPU runtime of AD-LRU is comparable to CFLRU, LRU-WSR, CFLRU/C, and CCF-LRU, because they all need search time to locate the victim for replacement. However, since the AD-LRU algorithm has the lowest write count, it has the best overall runtime.

As a result, AD-LRU reduced the number of writes under the Zipf trace compared to LRU, CFLRU, and LRU-WSR by about 23%, 17%, and 21%, respectively. In addition, our algorithm decreased the runtime by about 21%, 16%, and 20% over LRU, CFLRU, and LRU-WSR.

OLTP trace experiment. Table 8 describes the real OLTP trace used in the DBMS-based experiment. Figure 10a and Figure 10b show the write count and runtime of the experiment. AD-LRU is comparable with CCF-LRU for write count, but superior to all the other four competitor algorithms throughout the spectrum of buffer sizes, both for write count and runtime. Note that the



(a) write count (b) runtime

Figure 9: Performance of the Zipf trace in the DBMS-based environment

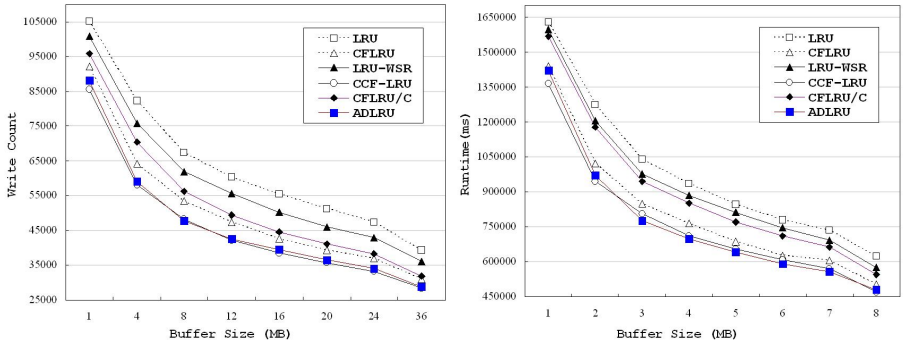
Attribute	Value
Total Buffer Requests	607,391
Database Size	20 GB
Page Size	2,048 B
Duration	One hour
Total Different Pages Accessed	51,870
Read / Write Ratio	77% / 23%

Table 8: The real OLTP trace

runtime of AD-LRU is still superior to the CCF-LRU algorithm, due to the higher hit ratio of AD-LRU. The performance of CFLRU is surprisingly better than that of LRU-WSR. We also noted this in the previous experiment over the 1,000k-100k Zipf trace. This result is somewhat different from what was reported in the LRU-WSR paper. The reason is that the parameter w of CFLRU is set to 0.5 in our experiment, while it was 0.1 in [6]. In the original paper proposing CFLRU, the parameter w varied from 1/6 to 1 in the experiment [8]. A larger window size in CFLRU means a growing probability to evict a clean page from the buffer, which will potentially reduce the number of write operations while accompanied with an increasing read count.

4.3. Impact of the parameter min_lc

The only parameter in the AD-LRU algorithm is *min_lc*, which refers to the minimal size of the cold LRU queue. The *min_lc* parameter is used to avoid the frequent replacement of recently-referenced pages in the cold LRU queue. This will occur if the cold queue is very small. On the other hand, it is very likely to get a cold queue containing only one page if we do not take any control, because AD-LRU will always choose the pages in the cold queue as victims. By setting up a minimal size of the cold queue, we have a better chance to



(a) write count (b) runtime

Figure 10: Performance of the OLTP trace in the DBMS-based environment

avoid this situation. For example, suppose that the current cold queue contains c pages, a new page is read, and the buffer is full. If min_lc is set to c, we will go to replace pages from the hot LRU queue and then increase the cold LRU queue. However, what is the optimal value for min_lc? To answer this question, we explore the performance of AD-LRU by varying the min_lc values. Here, we still use the four types of traces listed in Table 3 to Table 6, which refer to the four reference patterns: random, read-most, write-most, and Zipf. The min_lc value is changed from 0 to 0.9 in the experiment. Figure 11 shows the final results. Surprisingly, we find the min_lc parameter has little impact on the random, read-most, and write-most traces. The reason is that the page references are uniformly distributed in these three traces and there is no clear distinction between hot and cold pages. However, when the reference pattern is skewed, the best case appears when $min\_lc = 0.1$. If min_lc is much larger than 0.1, there are probably more replacements in the hot LRU queue. On the other hand, if min_lc is less than 0.1, most replacements will occur in the cold queue. Moreover, more dirty pages will be evicted from the cold queue, since its size is so small that we have little chance to find a clean page in it.

It is somehow difficult to determine the optimal value of min_lc for all kinds of workloads. Currently, we have not developed a theoretical method to determine the optimal min_lc, either online or offline. However, according to our experimental results for the four kinds of traces, it shows that to set $min\_lc = 0.1$ is an acceptable choice.

4.4. Scan Resistance

To examine the scan resistance of AD-LRU, we performed a custom-tailored experiment, where the baseline workload consists of a Zipf trace and fifty scans. The Zipf trace contains 100,000 references to 40,000 pages whose page numbers range from 0 to 39,999, and the fifty scans are restricted to 10,000 pages with page numbers ranging from 40,000 to 49,999. Each scan contains a fixed number

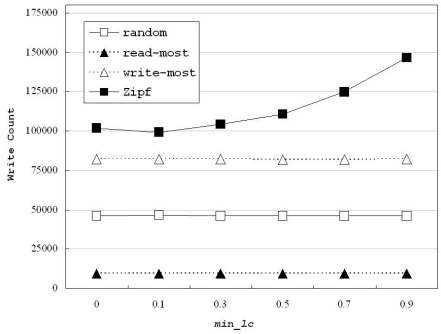


Figure 11: Impact of *min_lc* on the performance of AD-LRU

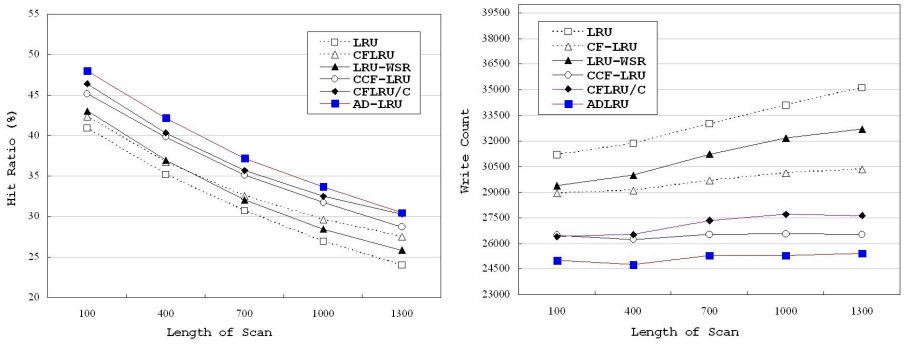
of read operations and all the scans are distributed uniformly among the Zipf trace. We change the length of scan, which refers to the number of continuous operations in a scan, to examine the performance of LRU, CFLRU, LRU-WSR, and AD-LRU. The buffer size in the experiment is 4,096 MB, the parameter *min_lc* of AD-LRU is 0.1, and the parameter w of CFLRU is 0.5.

The hit ratios and write counts are shown in Figure 12a and Figure 12b, respectively. While AD-LRU always maintains the highest hit ratio when using different scan lengths, it keeps a relatively stable write count. In contrast, LRU, CFLRU, CFLRU/C, and LRU-WSR, all imply a considerable increase of the write count when the scan length is increased. The CCF-LRU algorithm also has a stable write count, because of its two-LRU-queue mechanism. In summary, AD-LRU is also superior to LRU, CFLRU, CFLRU/C, and LRU-WSR in terms of scan resilience. Although CCF-LRU has similar scan resistance, it creates worse write counts than AD-LRU. When lots of page references from scans are present, the AD-LRU algorithm clearly adheres to its performance objectives much better than its competitors.

4.5. Device Sensitivity Study

SSDs are usually regarded as black-boxes. It has been experimentally demonstrated that the I/O performance of SSDs differs from that of flash chips [36], due to varying internal mechanisms such as address mapping and wear leveling. As a consequence, various SSD types have differing I/O performance. For example, the Intel-X25-M SSD has a sustained read speed of up to 250 MB/s, while the maximum read speed of the Mtron MSP SATA7525 SSD is only 130 MB/s.

It is not feasible to conduct experiments covering all SSD types, because the SSD design is still evolving. In this section, we use various SSD types to test the runtime performance of the AD-LRU algorithm and its competitors to validate the AD-LRU applicability for typical SSDs. Besides the previously used Super Talent SSD designed a couple of years ago and now considered as a low-end SSD, we include two additional SSDs: The SLC-based Mtron MSP SATA7525

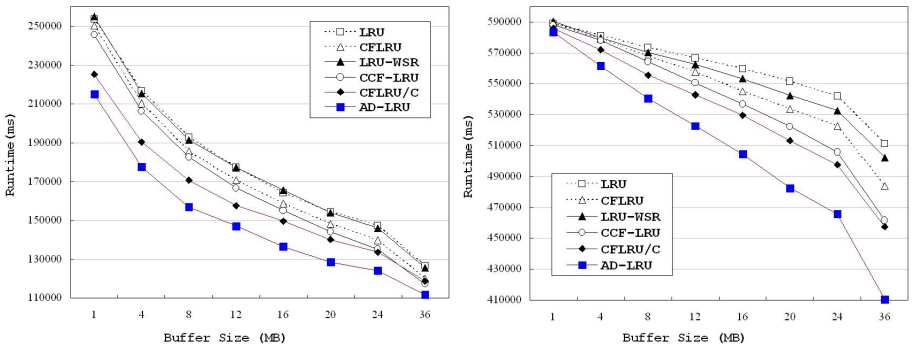


(a) hit ratio (b) write count

Figure 12: Impact of the scan length

SSD is selected as a middle-class SSD. In addition, the MLC-based Intel-X25-M (SSDSA2MH160G1) has high I/O bandwidth and stands for typical high-end SSDs. Therefore, those three SSDs are representative for a large spectrum of SSD types.

We re-executed the DBMS-based experiment over the 1000k-100k Zipf trace (as shown in Table 7) to measure the runtime cost. The results are shown in Figure 13. The runtime values substantially differ when using different SSDs, owing to their varying internal design. Nevertheless, AD-LRU has the lowest runtime cost in all cases. Therefore, we conclude that the performance benefits of AD-LRU are not device dependent.



(a) Intel SSD (b) MTRON SSD

Figure 13: Device sensitivity study

	1000k-100k Zipf, buffer: 10 MB		10-mill. Zipf, buffer: 100 MB		100-mill. self-sim., buffer: 1 GB	
Algorithm	Total (ms)	CPU	Total (ms)	CPU	Total (ms)	CPU
LRU	315940	5.8%	2963854	5.1%	17197826	8.0%
CF-LRU	308169	5.8%	2839573	5.3%	15949510	8.5%
LRU-WSR	319029	5.9%	2944521	5.1%	16791406	7.4%
CCF-LRU	307681	6.1%	2734531	5.8%	15068425	8.4%
CF-LRU/c	282908	7.1%	2484362	6.6%	16173421	10 %
AD-LRU	276188	6.4%	2301532	5.9%	13638785	8.6%

Table 9: The CPU time overhead, in percentage (%), compared with the overall execution time, in milliseconds (ms), for various traces and buffer sizes

4.6. Buffer-Size Impact on CPU Usage

AD-LRU maintains two LRU queues to reduce the writes to SSD. However, the CPU time to manipulate queues may increase with growing buffer sizes. To examine the CPU time overhead of AD-LRU as well as its competitors, we generated two new traces. One of them follows the Zipf distribution and the other follows the self-similar distribution. Both of them contain much more page requests than the previously introduced traces. The reference locality of both traces follows the 80-20 rule. The Zipf trace has 10 million random page requests, addressing 1 million pages (simulating a database of 2 GB) with 50% of the requests being read-only. The self-similar trace consists of 100 million page requests, 20% of them are read-only, and randomly addresses 10 million database pages, which correspond to a database of 20 GB.

We ran the traces for all the considered algorithms in the real DBMS environment, where we measured the overall execution time and the time spent doing IO. Then we derived the CPU portion of the execution time, as an indication of the CPU usage and the time complexity of the algorithms. For comparison, we also included the 1000k-100k Zipf trace introduced in the previous sections. The buffer size was always set to 5% of the database size, namely, 10 MB for the 1000k-100k trace, 100 MB for the 10-million Zipf trace, and 1 GB for the self-similar trace.

Table 9 shows the total runtime together with the CPU portion for the three traces. As indicates by these performance figures, AD-LRU always had the best performance. At the same time, the CPU usage remained relatively stable for all tested algorithms. Database applications are typically IO-bound, and this is also the case in our experiments.

5. Conclusions

Flash memory has become an alternative to magnetic disks, which brings new challenges to traditional DBMSs. To efficiently support the characteristics of flash storage devices, traditional buffering approaches need to be revised to take into account the imbalanced I/O property of flash memory. In the recent

three years, people have tried to present new buffer replacement policies for flash-based DBMSs. However, as the experimental results in our study show, the overall performance of those algorithms are not as optimal as we expect.

In this article, we proposed AD-LRU, a new efficient buffer replacement algorithm for flash-based DBMSs. The new algorithm captures both the frequency and recency of page references by using double LRU queues to classify all the buffer pages into a hot set and a cold set. It also uses an adaptive mechanism to make the sizes of the two LRU queues suitable for different reference patterns. We use different traces, including a real trace and some simulated traces, to evaluate the performance of the AD-LRU algorithm and also to compare it to the three competitor algorithms: LRU, CFLRU, and LRU-WSR. The experimental results show that in most cases the AD-LRU algorithm outperforms all competitors w.r.t. hit ratio, write count, and overall runtime.

Based on our experimental study, we draw the following conclusions:

(1) The LRU algorithm has the worst performance in each case. It shows that traditional algorithms will not work well in flash-based DBMSs.
(2) The performance of buffer management in flash-based DBMSs is dominated by the number of write operations, given the read/write latency of a typical flash memory.
(3) While the runtime in a simulation environment is much different from that in real SSD-based systems, it is still reasonable to use a simulation method to evaluate the hit ratio and write count of a buffer algorithm.
(4) It should be a good choice in flash-based DBMSs to first evict clean pages from the buffer. However, an additional effort has to be spent to avoid a significant degradation of the hit ratio; otherwise, the overall runtime will not be as good as expected.
(5) Our AD-LRU algorithm exhibits superior performance behavior than other methods proposed so far. It has a lower number of writes and less runtime compared to LRU, CFLRU, and LRU-WSR, both in the simulation environment and in the DBMS-based experiment.

Next we will implement our algorithm in a relational database engine, e.g., PostgreSQL or BerkeyDB, and perform further performance evaluations using standard benchmarks [37]. Another future work will be focused on using more than two queues to organize buffer pages. Basically, more queues will introduce more manipulation operations on data structures as well as more additional overhead to adjust queues and control their lengths. However, it may be helpful to improve hit ratio and reduce write operations for the buffer management in flash-based DBMSs, because different types of frequency can be supported by using more queues. Finally, Flash-DBSim used in our current experiments is only able to simulate the behavior of flash chips. As reported in [36], SSDs behave much different from flash chips. Thus, it will be one of our future works to make the outcome of Flash-DBSim more similar to real SSDs and to test the robustness of the AD-LRU algorithm under different SSD types.

In this article, flash SSDs are considered as a direct replacement of magnetic disks in a classical database architecture. It is an interesting future research

direction to study the use of flash in a wider spectrum of architectures, e. g., in a three-layer storage architecture with flash as a caching layer between the RAM-based main memory buffer and the storage layer based on magnetic disks [38], or in a key-value storage system consisting of an array of flash-based storage nodes [39].

6. Acknowledgment

We are grateful to anonymous referees for valuable comments that greatly improved this article, and to Gerhard Weikum for providing the real OLTP trace. This research is partially supported by the National Science Foundation of China (No. 60833005 and No. 61073039), the German Research Foundation, and the Carl Zeiss Foundation.

References

[1] L. P. Chang, T. W. Kuo, Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation, ACM Trans. on Storage (TOS) 1 (4) (2005) 381–418.

[2] C. H. Wu, T. W. Kuo, An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems, in: IEEE/ACM ICCAD'06, 2006, pp. 601–606.

[3] X. Xiang, L. Yue, Z. Liu, P. Wei, A Reliable B-Tree Implementation over Flash Memory, in: ACM SAC'08, ACM New York, NY, USA, 2008, pp. 1487–1491.

[4] J. Kim, D. Lee, C.-G. Lee, K. Kim, RT-PLRU: A New Paging Scheme for Real-Time Execution of Program Codes on NAND Flash Memory for Portable Media Players, IEEE Trans. on Computers 60 (8).

[5] Samsung Electronics, K9XXG08UXA 1G x 8 Bit / 2G x 8 Bit / 4G x 8 Bit NAND Flash Memory Data Sheet (2006).

[6] H. Jung, H. Shim, S. Park, S. Kang, J. Cha, LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory, IEEE Trans. on Consumer Electronics 54 (3) (2008) 1215–1223.

[7] P. Wei, L. Yue, Z. Liu, X. Xiang, Flash Memory Management Based on Predicted Data Expiry-Time in Embedded Real-time Systems, in: ACM SAC'08, ACM New York, NY, USA, 2008, pp. 1477–1481.

[8] S. Park, D. Jung, J. Kang, J. Kim, J. Lee, CFLRU: a Replacement Algorithm for Flash Memory, in: CASES'06, 2006, pp. 234–241.

[9] N. Megiddo, D. S. Modha, ARC: A Self-Tuning, Low Overhead Replacement Cache, in: FAST'03, 2003.

[10] MICRON Ltd., NAND Flash Memory MT29F4G08AAA, MT29F8G08BAA, MT29F16G08FAA, MT29F16G08FAA, `http://download.micron.com/pdf/datasheets/flash/nand/4gb_nand_m40a.pdf`, retrieved May 2009.

[11] T. Cho, Y. Lee, E. Kim, J. Lee, S. Choi, S. Lee, D. Kim, W. Han, Y. Lim, J. Lee, A Dual-Mode NAND Flash Memory: 1-GB Multilevel and High-Performance 512-Mb Single-Level Modes, IEEE Journal of Solida-State Circuits 46.

[12] Intel Corporation, Understanding the Flash Translation Layer (FTL) Specification, Tech. Rep. AP-684 (1998).

[13] Z. Liu, L. Yue, P. Wei, P. Jin, X. Xiang, An Adaptive Block-Set Based Management for Large-Scale Flash Memory, in: ACM SAC'09, ACM, 2009, pp. 1621–1625.

[14] F. J. Corbato, A Paging Experiment with the Multics System, in: In Honor of Philip M. Morse, MIT Press, Cambridge, Mass, 1969, p. 217.

[15] E. O'neil, P. O'neil, G. Weikum, The LRU-K Page Replacement Algorithm for Database Disk Buffering, in: SIGMOD, 1993, pp. 297–306.

[16] W. Effelsberg, T. Härder, Principles of Database Buffer Management, ACM Trans. on Database Systems (TODS) 9 (4) (1984) 560–595.

[17] V. Nicola, A. Dan, D. Dias, Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing, in: ACM SIGMETRICS Performance Evaluation Review, ACM, 1992, pp. 35–46.

[18] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, C. Kim, LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies, IEEE Trans. on Computers (2001) 1352–1361.

[19] T. Johnson, S. D., 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, in: VLDB, 1994, pp. 439–450.

[20] J. Robinson, M. Devarakonda, Data Cache Management Using Frequency-Based Replacement, ACM SIGMETRICS Performance Evaluation Review 18 (1) (1990) 142.

[21] Y. Yoo, H. Lee, Y. Ryu, H. Bahn, Page replacement algorithms for NAND flash memory storages, in: Computational Science and Its Applications (ICCSA 07), Springer, 2007, pp. 201–212.

[22] Y. Ou, T. Härder, P. Jin, CFDC: a Flash-Aware Replacement Policy for Database Buffer Management, in: Proc. of the 5th International Workshop on Data Management on New Hardware, ACM, 2009, pp. 15–20.

[23] Y. Ou, T. Härder, Clean First or Dirty First?: a Cost-Aware Self-Adaptive Buffer Replacement Policy, in: Proc. of the 14th International Database Engineering & Applications Symposium, ACM, 2010, pp. 7–14.

[24] H. Jung, K. Yoon, H. Shim, S. Park, S. Kang, J. Cha, LIRS-WSR: Integration of LIRS and Writes Sequence Reordering for Flash Memory, in: ICCSA'07, Vol. 4705 of LNCS, 2007, pp. 224–237.

[25] S. Jiang, X. Zhang, LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance, in: SIGMETRICS, ACM New York, NY, USA, 2002, pp. 31–42.

[26] Z. Li, P. Jin, X. Su, K. Cui, L. Yue, CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory, Trans. on Cons. Electr. 55 (2009) 1351–1359.

[27] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz, ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, ACM Trans. Database Syst. 17 (1) (1992) 94–162.

[28] T. Härder, A. Reuter, Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys 15 (4) (1983) 287–317.

[29] X. Su, P. Jin, X. Xiang, K. Cui, L. Yue, Flash-DBSim: A Simulation Tool for Evaluating Flash-based Database Algorithms, in: IEEE ICCSIT'09, Beijing, China, 2009.

[30] Flash-DBSim, `http://kdelab.ustc.edu.cn/flash-dbsim/index_en.html`, retrieved May 2009.

[31] O. Kwon, J. Lee, K. Koh, EF-Greedy: A Novel Garbage Collection Policy for Flash Memory Based Embedded Systems, in: ICCS'07, Springer-Verlag, 2007, pp. 913–920.

[32] K. Lofgren, R. Norman, G. Thelin, A. Gupta, Wear Leveling Techniques for Flash EEPROM Systems, United States Patent 6850443 (2005).

[33] D. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, 1973.

[34] T. Härder, K. Schmidt, Y. Ou, S. Bächle, Towards Flash Disk Use in Databases - Keeping Performance While Saving Energy?, in: BTW'09, 2009, pp. 167–186.

[35] M. P. Haustein, T. Härder, An Efficient Infrastructure for Native Transactional XML Processing, Data & Knowledge Engineering 61 (3) (2007) 500–523.

[36] L. Bouganim, B. Jónsson, P. Bonnet, uFLIP: Understanding Flash IO Patterns, in: CIDR, 2009.
URL `http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_102.pdf`

[37] J. Gray (Ed.), The Benchmark Handbook for Database and Transaction Systems (2nd Edition), Morgan Kaufmann, San Francisco, 1993.

[38] Y. Ou, T. Härder, Trading Memory for Performance and Energy, in: Database Systems for Adanced Applications, Springer, 2011.

[39] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, V. Vasudevan, FAWN: a Fast Array of Wimpy Nodes, Communications of the ACM 54 (2011) 101–109.

Peiquan Jin received his Ph.D. degree in computer science from University of Science and Technology of China, in 2003. Before that, he received his master and bachelor degree in management science both from Tianjin University of Finance and Economics in 2000 and 1997, respectively. He is currently an associate professor in School of Computer Science and Technology, University of Science and Technology of China. His research interests include flash-based databases, spatiotemporal databases, and Web information extraction. He is a member of ACM, and is an editor of International Journal of Information Processing and Management, and International Journal of Knowledge Society Research. He serves as a PC member of many international conferences, including DEXA'09-11, WAIM'11, ICCIT'08-11, NISS'09, and NDBC'09-11.

Yi Ou obtained his Diplom degree in Computer Science from the TU Kaiserslautern in 2008. Since then, he works as a Ph.D. student in the research group DBIS lead by Prof. Theo Härder. His research interests include performance and energy efficiency of database storage systems, especially caching algorithms.

Theo Härder obtained his Ph. D. degree in Computer Science from the TU Darmstadt in 1975. In 1976, he spent a post-doctoral year at the IBM Research

Lab in San Jose and joined the project System R. In 1978, he was associate professor for Computer Science at the TU Darmstadt. As a full professor, he is leading the research group DBIS at the TU Kaiserslautern since 1980. He is the recipient of the Konrad Zuse Medal (2001) and the Alwin Walther Medal (2004) and obtained the Honorary Doctoral Degree from the Computer Science Dept. of the University of Oldenburg in 2002. Theo Härder's research interests are in all areas of database and information systems – in particular, DBMS architecture, transaction systems, information integration, and XML database systems. He is author/coauthor of 7 textbooks and of more than 280 scientific contributions with $>$ 160 peer-reviewed conference papers and $>$ 70 journal publications. His professional services include numerous positions as chairman of the GI-Fachbereich *Databases and Information Systems*, conference/program chairs and program committee member, editor-in-chief of Computer Science Research and Development (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluver), and Transactions on Database Systems (ACM).

Zhi Li obtained his master degree in computer science from University of Science and Technology of China in 2010. His research fields focus on flash-based databases, especially on buffer management issues and query processing algorithms.