University of Kaiserslautern
Department of Computer Science
Database and Information Systems

Caching for flash-based databases

Summer Semester 2013

Table of Contents

CACHING FOR FLASH-BASED DATABASES

Sougata Bhattacharjee
Technical University of Kaiserslautern

Abstract

Today database storage is mainly based on two technologies: HDD & DRAM. Flash memory devices are now considered to have tremendous potential as a new storage medium that can partially replace HDD & DRAM, because the energy concern of both HDD & DRAM is growing for fast expanding data. Due to its superiority such as low access latency, low energy consumption, light weight, and shock resistance, the success of flash memory as a storage alternative for mobile computing devices has been steadily expanded into personal computer and enterprise server markets with ever increasing capacity of its storage. Another advantage of flash memory is that it is cheaper than DRAM and faster than HDD. But due to some limitations and complex characteristics of flash memory, the efficient use of flash become challenging and, as a consequence of this, it may exhibit poor performance. Conventional storage systems are without prior knowledge of such limitation. Therefore, both algorithmic and architectural improvements are necessary for flash devices to fully exploit the potential of flash memory. In this paper, we will discuss the characteristics of flash memory and the advantages of flash memory over conventional storage devices. We identify the problems with cache management while we discuss various cache replacement policies: Least Recently Used (LRU) and Adaptive Replacement Cache (ARC). Then we highlight the weaknesses of conventional buffer management policies for flash-based database systems. To overcome such deficiencies, we propose the Clean-First Dirty-Clustered (CFDC) algorithm. CFDC is one of the earliest policies, which addresses the flash random write problem. We propose another improved algorithm called CASA (Cost-Aware Self-Adaptive). It is mainly designed for two-tier storage systems with homogeneous storage devices and can automatically adapt to the workload at runtime. The paper mainly contributes to buffer management issues for two-tier storage systems where flash-based SSDs are used as the primary storage, that is, what we called caching for flash-based databases.

1. Introduction

The world's information is doubling every two years. In 2011, the world had created a staggering 1.8 zettabytes of data. By 2020, the world will generate 50 times the amount of this information and 75 times the number of "information containers", while the IT staff to manage it will grow less than 1.5 times. According to IDC, the amount of data captured in digital form was 281 Exabyte. The volume of information continues to grow rapidly. The growth of data or information at this rapid speed is called information explosion. As the amount of available data grows, the problem of managing the data becomes more difficult, which can lead to data overload. So data and data management are becoming more important to the user and the organization. The concept of "data management" arose in the 1980s as technology moved from sequential processing (first cards, then tape) to random access processing and efficient data management mainly depended on efficient storing and accessing of data. These are basic functionalities of a database system. A database is a collection of information that is organized so that it can easily be accessed, managed, and updated. A database system consists of a collection of data which is referred to as database and a software system called database management system (DBMS)

[http://www.databasecompare.com/What-is-data-database-DB-DBMS-and-DBS.html]. Data can be stored on a computer storage system which mainly consists of DRAM (Dynamic Random Access Memory) and HDDs (hard disk drives or magnetic disks). DRAM is used as main memory in computers and HDD is used as secondary or external storage device. HDD is the most dominant storage device for both personal computers and server computers today. However, HDDs have a major limitation due to its mechanical structure, which is latency. Latency is the access delay caused by the mechanical arm movement (access motion) to locate the disk cylinder to be accessed (seek time) and the disk rotation to bring the disk sector addressed under the read-write mechanism. The average seek time of a high-end HDD is 2 - 4 ms. For a rotational speed of 15,000 rpm, the average rotational delay is 2 ms. Hence, the entire access time is typically 4 – 6 ms, which means that the IOPS (IOs per sec) rate of such an HDD is ~200. In contrast, DRAM is much faster than HDD but has two major limitations: cost and energy consumption.
Hence, latency of HDDs and cost and power consumption of DRAM are the major limitations of today's storage systems. To overcome these limitations we need a storage system which is much faster than HDDs and cheaper than DRAM. We can introduce a new storage technology called flash memory, which was invented In 1980 by Dr. FujioMasuoka while working for Toshiba. Since this time, flash memory achieved great improvements in both price and performance over HDDs and DRAM. Flash SSDs originate a disruptive change with respect to storage technology and are becoming a strong competitor to conventional HDDs and DRAM in the area of persistent database stores [4].

1.1 Flash Memory

Flash memory is an electronic non-volatile computer storage device that can be electrically erased and reprogrammed. Flash memory was developed from EEPROM (electrically erasable programmable read-only memory) [14]. It does not require power to store data on it and no mechanical components are required to access those data. It does not have mechanically moving parts like disk arms in a magnetic disk drive; hence, it provides uniform random access speed. Further attractive properties of this type of memory are small form factor, shock resistance, and energy efficiency. One of the key characteristics of flash memory is to quickly perform a sector read or a sector write located anywhere in the flash memory. Because of these properties, flash memory has been the dominant media in mobile devices and embedded systems, for example, in USB flash devices, smart phones, digital cameras, etc. One more recent application of flash memory is its use as a replacement for hard disks in enterprise server environments. Recently, many manufacturers have packed flash memory chips into solid-state drives (SSDs) for personal computers and servers.

Compared to DRAM, flash memory is a low-cost and non-volatile device, i.e. it retains the data stored when power is turned off. Due to its low-cost advantage over DRAM, the market segment of flash memory has have been growing fast in recent years. DRAM and flash store bits in fundamentally the same way: charge (on-bit) or a lack of charge (off-bit) is stored in a capacitor [The Daily Circuit, http://www.dailycircuitry.com/2010/11/flash-vs-dram.html]. Flash memory stores information in an array of memory cells, each cell stores only one bit of information. In recent times, the situation is improved due to introduction of better processes. Some newer types of flash memory, known as multi-level cell (MLC) devices can store more than one bit per cell by choosing between multiple levels of electrical charge to apply to the floating gates of its cells. Flash chip stacking has a traditional problem of overheating because each layer consumes power and the layers insulate themselves. But now flash devices improved their density due to additional bits per cell. From 1995-2005, for example, the density of flash memory chips has doubled per year (faster than the expectation based on Moore's Law), while cost reduction was about 50% per year [Samsung. Memory technology and solutions roadmap.http://www.samsung.com/us/aboutsamsung/ir/ireventpresentations/analystday/downloads/analyst_20051104_0800.pdf, 2005. Samsung Analyst Day, Samsung Electronics Co., Ltd., J. Gray. Tape is dead, disk is tape, flash is disk, RAM locality is king. Storage Guru Gong Show, December 2006.].

Compared to magnetic HDDs, flash devices have a much lower latency, because a flash device does not require any mechanical movement for accessing the data. Flash devices have no moving parts and they store information only in microchips, for example, used as a USB flash drive. They can instantly start storing information or reading files, without having to wait for any moving parts to get in place. So, it is possible for a single solid state drive (SSD which is based on flash memory) to achieve maximum IOPS which were not possible using a single HDD. Another advantage is that an SSD does not need to be defragmented; in contrast, a fragmented HDD will greatly reduce read/write times of the HDDs. Solid state drives also consume much less power and produce much less heat than hard disk drives do. And comparing the price basis, 'cheaper' SSDs than HDDs based on $ per GB may be soon available in the market.

Device	Model	EURO/GB	Latency(ms)
RAM1	Kingston KVR667D2D8P5/2G	19.00	~ 10 ns
RAM2	Kingston KHX1600C9D3B1K2/4GX	19.11	~ 10 ns
RAM3	Kingston KVR1333D3D4R9S/4G	24.70	~ 10 ns
SSD1	SuperTalent FSD32GC35M	N/A	0.1
SSD2	MTRON MSP-SATA-7525-032	N/A	0.083
SSD3	Intel SSDSA2MH160G1GN	2.40	0.029
SSD4	Intel SSDSA1MH160G2GN	2.44	0.029
SSD5	Crucial CTFDDAC256MAG-1G1	2.01	0.017
HDD1	WD WD800AAJS 7200 RPM	0.38	15.000
HDD2	WD WD1500HLFS 10000 RPM	0.77	4.500
HDD3	Fujitsu MBA3147RC 15000 RPM	0.76	2.000

Table 1: Price and performance of three different kind of storage devices [Sales price of Internet stores as of Nov' 2010]

In terms of power consumption, SSDs use significantly less power at peak load than hard drives and DRAM; for example, flash consumes less than 2W vs. 6W for an HDD.SSDs feature a non-mechanical design of NAND flash mounted on circuit boards and are shock resistant up to 1500g/0.5ms. Hard Drives consist of numerous moving parts making them vulnerable to shock and damage [SSD vs HDD: http://ocz.com/consumer/ssd-guide/ssd-vs-hdd]. Due to its advantage over HDD and DRAM, it is gaining the attention in new market segments. Flash disks have been already widely used for mobile devices, embedded systems, and server platforms (in the form of Solid State Drives, SSDs); they are also used as substitutes for hard drives in high-performance desktop computers and some servers with RAID and SAN architectures.

Though a flash device has several advantages, its efficient use is much more challenging due to some of its special properties and limitations. Flash memory was developed from EPROM. One limitation of flash memory is that, although it can be read or programmed a byte or a word at a time in a random access fashion, it can only be erased one block at a time. A cell can change its value from 1 to 0, but setting a value from 0 to 1 requires an expensive erase operation. So, once a bit has been set to 0, it can be changed back to 1 only by erasing the entire block. Flash memory (specifically NOR flash) offers random-access read and programming operations, but does not offer arbitrary random-access rewrite or erase operations. This limitation is referred to as erase-before-write [14]. Another limitation is that flash memory has a limited number of program/erase (P/E) cycles. Most flash devices are withstanding around 10,000 to 1,000,000 P/E cycles before the wear begins. This limitation is referred as write endurance. The method used to read NAND flash memory can cause nearby cells in the same memory block to change over time (become programmed). This is known as read disturb.

Flash memory can be classified into two main types: NAND and NOR flash. The NAND type is primarily used for general storage and transfer of data, used in main memory, USB flash drives, solid-state drives, and similar products. NOR flash allows true random access and therefore direct code execution, is used as a replacement for EPROM and NOR flash memory is used as code storage. Although NOR flash memory has lower density, it is more expensive than NAND flash memory.

Flash memory is accessed using an intermediate layer called Flash Translation Layer (FTL). FTL is used to hide the limitations of flash memory and to provide the illusion of an HDD-like block device.FTL basically hides the erase-before-write limitation of flash memory. Although SSD can work like HDD, its performance characteristics are largely different compared to HDD due to the specific characteristics and limitation of flash. For an example, random writes on flash devices are slower than sequential write and read operations. This problem is known as flash random write (FRW).

2. Architecture of a Flash-based Database System

FlashDB is an optimized self-tuning database using NAND flash storage. FlashDB is self-tuning; after it is initially configured with the page read and write costs of the underlying storage device, it automatically adapts its storage structure in such a way that it optimizes energy consumption and latency for the workload it handles [8]. Generally, FlashDB instances are running on different kinds of flash devices with different workloads. As shown in Fig. 1, FlashDB consists of two major components: a database management system and a storage manager. The database management system implements the database functions including index management and query compilation, and the storage manager provides efficient storage functionalities such as data buffering and garbage collection.

Logical storage provides logical sector address abstraction on top of physical flash pages. It hides flash-specific complexities using Out-of-Place and Garbage Collection mechanisms. The Garbage Collection mechanism cleans dirty pages produced by write operation. However, a page cannot be erased independently. At first, we need to choose a flash block containing dirty pages. Then, valid pages of the block are copied to another block. Finally, the block is erased.

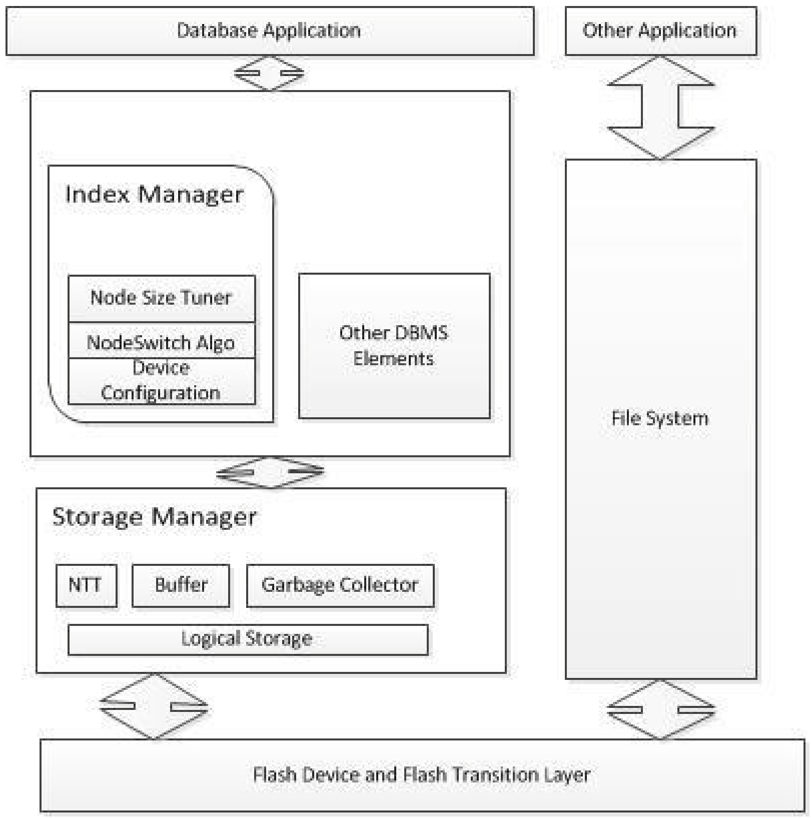


Fig. 1: FlashDB Architecture [reference 8, fig.3]

Here we will not discuss the other components in details. The interface to the flash chips is made through an abstraction layer called Flash Translation Layer (FTL)**.** We will discuss FTL in detail, when we use flash devices. The FTL provides a disk-like interface, which includes the capability to read and write a page directly without worrying about the erase-before-write limitation. But FTL internally needs to deal with the characteristics of the underlying flash device. Thus, even if a storage system uses a flash device with a built-in FTL, it is advisable to consider the flash characteristics in the storage design.

2.1 Flash Translation Layer

Flash memory supports three kind of basic operation: Read, Write, and Erase. Read and Write operations are performed in units of flash pages and Erase operations are performed for flash block; a block can contain multiple pages. The FTL provides an interface, which has the capability to read and write a logical page directly without worrying about the erase-before-write limitation. FTL implements an out-of-place update scheme. A page is addressed by read/write operations and is generally 2 KB or 4 KB in size. If we erase the block, where the target page belongs to, and write new data to the cleaned page targeted, the data of the other pages that belong to the same block will be lost. Hence, we cannot perform an in-place update scheme that overwrites the original page. That is why we need to use out-of-place update schemes. The new data are written to another clean page and the original page is invalidated. The location of valid data changes on every re-write request. Performing the out-of-place update and maintaining the mapping information between logical sector and physical location are essential functions of FTL.

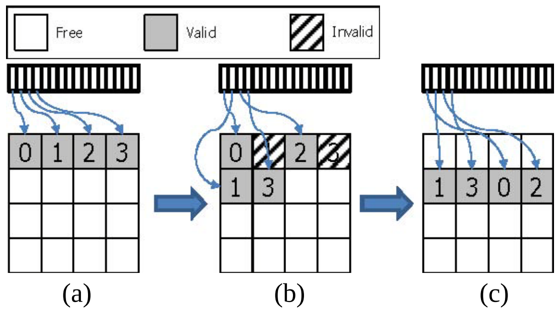


Fig. 2: Remapping and Block Cleaning (Flash Transition Layer) [Reference 9, fig.1]

As shown in Figure 2.a, the content of the flash memory is illustrated after the first write requests to the four logical sectors, which are stored in pages numbered from 0 to 3. After initial write requests, sectors 1 and 3 are requested to be updated, as shown in Figure 2.b. Because of the overwrite limitation, the in-place update scheme cannot be performed. So, FTL allocates new pages 4 and 5 (second row, as shown in figure 2.b) and then writes the updated data onto those new pages. After updating, FTL invalidates the original pages 1 and 3 and finally modifies the mapping table to emulate these changes. Figure 2.c shows the block cleaning mechanism. At first, it chooses a block to be reclaimed and copies valid pages of the chosen block into another block that has been erased. Finally, it erases the reclaimed block [9].

FTL supports a garbage collection procedure. Multiple writes into a logical page can produce multiple page versions in flash memory. A valid page means the latest version of that page and invalid page means an older page version. So, a valid flash page is the physical page containing the latest version of the page. When free flash pages are short in supply, garbage collection reduces the number of invalid pages and increases the number of free flash pages. An address mapping table is required to keep track of the valid flash page to logical page mapping. According to the mapping unit, FTLs can be divided into three categories: page mapping, block mapping, and hybrid mapping.

FTL supports another important mechanism called wear leveling. It replaces blocks which are more frequently erased with the less frequent erased blocks. This policy helps to increase the flash device lifespan.

So we can summarize that FTL supports three main mechanisms: out-of-place update scheme, garbage collection and wear leveling. All the three mapping techniques are associated with an out-of-place update scheme. Now we will briefly describe the three mapping techniques.

2.1.1 Page-Level Mapping

This mapping technique maps each logical page to some physical page in flash memory. A page mapping table is used for storing and managing the mapping information between Logical Page Number and Physical Page Number. Page-level mapping efficiently deals with erase-before-write constraints. If a block contains M pages then a logical write requires one program operation and also one read operation and 1/M erase operation for the Garbage Collection scheme in average. Among all three mapping techniques, the page-level mapping technique has the highest performance potential. For performance reason, the mapping table is usually stored in DRAM. For an example, assume that we have a 256GB SSD with a page size of 2KB, then there will be 2^{27} entries in its mapping table. If each entry consumes 4B space, the mapping table will occupy 512MB DRAM space, which is too expensive and energy-demanding for a typical SSD product to be cost-effective and energy-efficient. This mapping technique requires more resources than the other mapping techniques due to the size of mapping table [1].

2.1.2 Block-Level Mapping

Block-level uses a mapping table to store and manage the mapping information between Logical Block Number and Physical Block Number. The address information for this mapping is much smaller. For example, if a block contains M pages, the size of the block-mapping table is M times smaller than that of page-level mapping. Using block-level mapping, the offset of a physical block's page must be identical to its offset of logical block. Due to this offset constraint, if we want to update a page, the new content must be written using the same offset in a free flash block and the remaining pages of the old block must be copied to the new block. It requires M-1 read operations and one erase operation. So, if we compare it with the page-level mapping, the block-level mapping requires extra operations to serve a request, hence it critically affects the performance [1].

2.1.3 Hybrid Mapping

The shortcomings of page-level mapping and block-level mapping can be overcome by using a hybrid mapping technique. In such a scheme, physical blocks are logically partitioned into two groups: data blocks and log blocks. Log blocks handle the write requests. The page addresses are mapped at the page level in a log block; in this way, frequent block erasures can be prevented. Data blocks cover a much larger flash area, hence the size of the mapping table is not an issue for hybrid mapping. A log block performs a write operation only for the associated data block. If it is full then each page in which it was written at least once will be merged with the associated data block. When no log blocks are available, their data are flushed into the data blocks immediately and they are then erased to generate new free log blocks. More precisely, the valid data in data blocks and the valid data in the corresponding log blocks must be merged and written to a new data block. This process is called a merge operation. However, the merge operations can only be reduced to a certain degree, therefore, a hybrid mapping scheme is unable to achieve the desired performance of page-level mapping.
The FTL serves a key role for flash devices, because it helps to organize the physical and logical blocks. It also provides the mechanisms how garbage collection is performed and how the device cache is utilized [1].

3. Page replacement algorithm

Operating systems use paging techniques for virtual memory management, where a page replacement algorithm decides which pages should swap in or which pages are swapped out when a memory frame needs to be allocated. Page swap and page selection require I/O operations. When a page was selected for replacement and paged out and is referenced again, it has to be paged in, i.e., read from disk, and this involves waiting for I/O operation completion. For this reason, we use a page replacement algorithm. The more efficient the algorithm we use, the less time we have to wait for a paging operation. Here we will propose the Adaptive Replacement Cache (ARC) algorithm which has several advantages over the Least Recently Used (LRU) technique. The LRU technique maintains an ordered list (the cache directory) of resource entries in the cache, with an order based on the time of most recent access. New entries are added at the top of the list, after the bottom entry has been popped out from the list. Cache hits move to the top, pushing all other entries down. ARC improves the LRU technique by dividing the cache list into two lists. Basically, ARC maintains two dynamic lists of pages, one is the recency list keeping pages that have been seen only once and the other one is the frequency list, which maintains pages that have been

seen at least twice. We will describe about ARC in details later.
ARC is scan resistant. ARC allows one-time-only sequential read requests to pass through the cache without mantling pages [10]. It is also easy to implement similar to the LRU technique.
The problem of caching:
The problem of cache management is to implement a replacement algorithm, which can maximize the hit ratio. When the cache is full, the algorithm must choose which items to discard to make room for the new ones. Two very important factors for caching are hit rate and latency, which both affect the cache performance. The "hit ratio" of a cache describes how often a searched-for item is actually found in the cache. More efficient replacement algorithms keep track of more usage information in order to improve the hit rate (for a given cache size). Latency describes the time between requesting a page from the cache and returning this page to the requestor. We consider a system with main memory (cache) and secondary memory (HDD) where the cache is faster than the secondary memory but more expensive. So, obviously the size of the cache is much smaller than the main memory. We assume that the cache receives continuous requests of pages and where a page is fetched into cache from secondary memory only when the page is not present in the cache. Having a full cache, an existing page must be evicted from the cache before the new page can be fetched from secondary memory. Page eviction should be done in an optimal way by the cache replacement algorithm. We already discussed the hit rate; in another way, we can say that, related to all page requests, the hit rate is the fraction of pages found in main memory and the miss rate is the fraction of pages that have to be fetched from secondary memory. Another important factor for a cache replacement algorithm is space overhead involvement.

ARC Concept

ARC contains two LRU lists L1 and L2. L1 is the recency list, i.e., it contains recently seen pages, and L2 is the frequency list, i.e., L2 contains pages that have been seen at least twice "recently". Both L1 and L2 are of same size, namely the cache size is c. Together the two lists contain exactly twice as many pages that fit in the cache. Both L1 and L2 are associated with another two lists B1 and B2 known as ghost lists, which are attached to the bottom of both lists. These ghost entries keep track of recently evicted pages from L1 or L2, but the ghost lists contain only the reference or key of the pages not the data itself. The whole cache directory can be visualized in a single line:

```
. . . [   B1  <-[     T1    <-!->      T2   ]->  B2   ] . .
      [ . . . . [ . . . . . . ! . .^. . . . ] . . . . ]
                [   fixed cache size (c)    ]
```

[reference 11]

L1 and B1 together are referred to as T1, a combined history of recent single references. Similarly, T2 is the combination of L2 and B2.The inner [ ] brackets denote actual cache, however the cache is fixed in size and can be moved freely across the B1 and B2 history. ^ indicates the target size for L1. New entries enter L1, to the left of !, and are gradually pushed to the left, and eventually being pushed out from L1 into B1. Any entry in T1 that gets referenced once more, gets another chance, and enters T2, just to the right of the central ! marker. From there, it is again pushed outward, from L2 into B2. Entries in L2 that get another hit can repeat this indefinitely, until they finally drop out on the far right of B2. The pages re-entering the cache (L1, L2) will cause ! to move towards the target marker ^. If no free space exists in the L1 or L2, this marker also determines whether either L1 or L2 will evict an entry [11].

If we compare ARC with non-adaptive approaches like a fixed replacement policy (FRC_p) which attempts to keep p most recent pages in L1 and (c-p) most recent pages from L2 where c is the total number of pages in the cache. ARC works like FRC_p, but it can change the parameter p adaptively according to the variable workload. The main idea is to adaptively manage the workload and decide how many top pages

from each of the lists to be maintained in cache at a given point of time. And we can achieve such online adaptation by implementing the ARC algorithm. ARC will detect the change if one of the two lists (recency or frequency) become more important than the other during workload processing.
Like LRU, ARC is easy to implement as we need only two LRU lists; and its running time per request is essentially independent of the cache size. A real-life implementation revealed that ARC has a low space overhead—0.75 percent of the cache size.

Prior Cache Replacement algorithms

Bélády's Algorithm is an optimal and offline policy for replacing the page in the cache that has the greatest distance to its next reference [3]. The efficient caching algorithms always try to discard the page that will not be needed for the longest time in the future. This optimal result is referred to as Bélády's optimal algorithm. However, we cannot predict how far in the future a particular page will be needed, it is generally not implementable in practice. The practical minimum can be calculated only after experimentation, and one can compare the effectiveness of the actually chosen cache algorithm [13].
The Least Recently Used (LRU) algorithm always discards the least recently used items first. LRU keeps track on those pages that have been most heavily used in the past. General implementations of LRU algorithm require keeping the track of "Least Recently Used" pages in cache. The LRU algorithm has gone through several approximations and improvements. The LRU algorithm has several advantages, it is easy to implement and it is amenable to full statistical analysis. It has been proven, for example, that LRU can never result in more than N-times more page faults than OPT algorithm, where N is proportional to the number of pages in the buffer.
The independent reference model (IRM) captures the page reference frequencies. Under IRM, the Least Frequently Used (LFU) algorithm replaces the least-frequently-used pages. But LFU has several drawbacks, for example, it does not keep track of recently used pages and it does not adapt to changing access patterns, i.e. it is accumulating stale pages with past high-frequency counts, which may no longer be useful.
Compared to LRU, Most Recently Used (MRU) discards most-recently-used items first. When a file is being repeatedly scanned in a [Looping Sequential] reference pattern, MRU is the best replacement algorithm. MRU cache algorithms have more hits than LRU due to their tendency to retain older data. MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed [13].

3.1 Class of Replacement Policies

3.1.1 Double Cache Policy

Let 2c denote the total number of pages, a cache can hold. DBL maintains two variable LRU lists L1 and L2. L1 contains pages that have been accessed recently at least once and L2 contains pages that have been seen recently at least twice. Specifically, a page in L1 has been requested exactly once since the last time it was removed from $L1 \cup L2$. And similarly a page resides in L2 if it has been requested more than once since the last time it was removed from $L1 \cup L2$. The replacement policy is: Replace the LRU page in L1, if L1 contains exactly c pages; otherwise, replace the LRU page in L2, where the size of both lists can fluctuate.

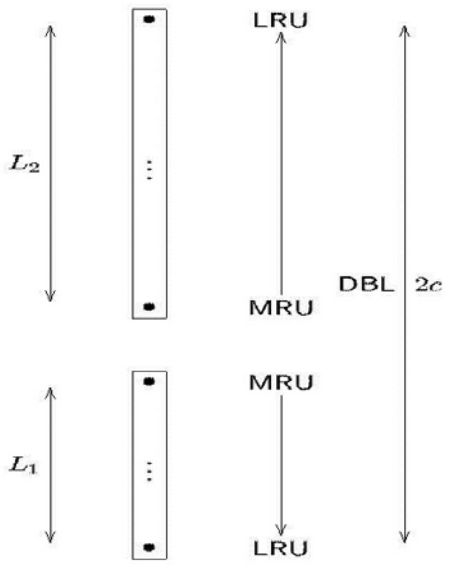
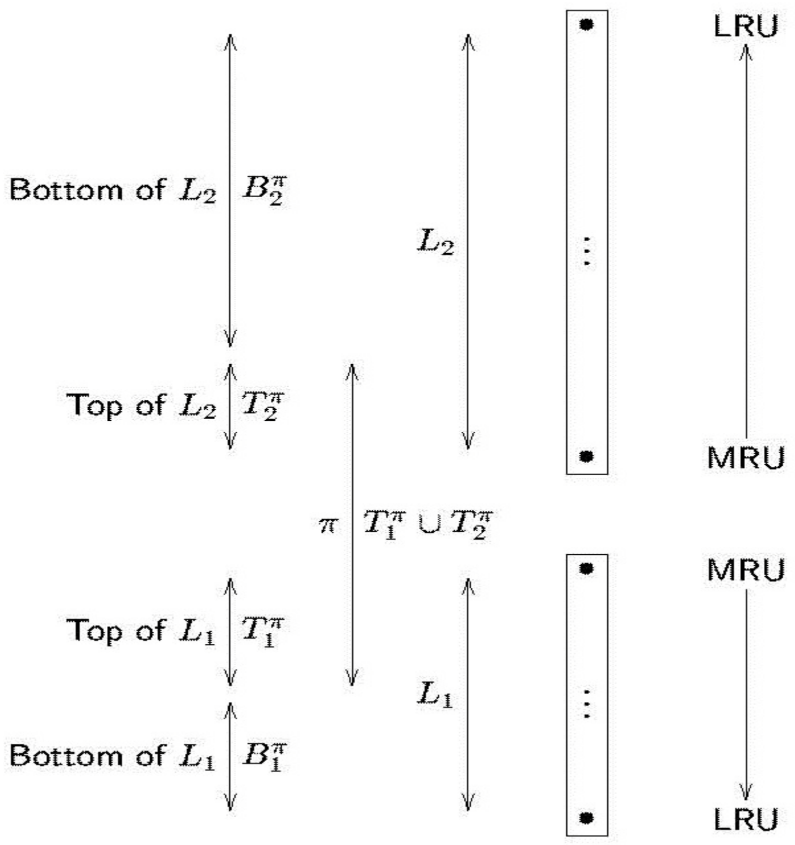


Fig.3 : General structure of the cache replacement policy DBL [reference 2, fig.2]

3.1.2 New Class of Policy

Here we introduce a new algorithm called class of policies II(c) This policy keeps track of all 2c pages in the cache which is managed by DBL(2c) but at a time keeps only c pages in cache. Fig 3 denotes the generic replacement policy II(c), where L1 and L2 are the two lists associated with DBL(2c). Suppose II(c) proposes a class of replacement policies in such a way that every policy π(c) belongs to II(c) and there exists a top portion T_1^π and bottom portion B_1^π for a dynamic list L1 and the same for a dynamic list L2. The partitions are created in such a way that T_1^π contains the most recent pages in L1 and B_1^π contains the least recent pages in L1 and similarly for L2 in T_2^π and B_2^π. However, these partitions follow some specific conditions [2]:

1. If L1 and L2 both have less than c pages, then B_1^π & B_2^π are both empty.
2. If L1 and L2 both have more than c pages or an equal number of c pages, then T_1^π &T_2^π both exactly have c pages.
3. Either T_1^π or B_1^π is empty or the LRU page in T_1^π is more recent than the MRU page in B_1^π
4. $T_1^\pi \cup T_2^\pi$ contains those pages, which have been cached by this policy throughout the process.



Bottom of L_2 B_2^π
Top of L_2 T_2^π
π $T_1^\pi \cup T_2^\pi$
Top of L_1 T_1^π
Bottom of L_1 B_1^π
L_2
L_1
LRU
MRU
MRU
LRU

Fig. 4 : General structure of a generic cache replacement policy [reference 2, fig.3]

3.1.3 Adaptive Replacement Policy

The behavior of ARC at any given time can be described by the adaption parameter p. At any given time, ARC can behave like the fixed replacement cache FRCp depending on the value of P where o<P<c. The difference with FRCp is that ARC does not contain the fixed parameter P unlike FRCp. According to the workload, ARC can adaptively change the value of P. Let T_1, B_1, T_2 and B_2 be dynamic partitions of the L1 and L2 lists. ARC can adaptively decide whether to replace a page in T_1 or in T_2 according to the workload. When P approaches towards zero, the algorithm emphasizes the pages in L2. When P approaches towards the cache size, i.e. c, it emphasizes the pages in L1. ARC continuously changes the value of parameter P. ARC will increase the size of T_1 and the value of P, if there is a cache hit in B_1, similarly if there is a cache hit in B_2, then it will increase the size of T_2. However, if there is a hit in B_2, the value of P is decreased. The target size of list L1 is P and c-P is the size of L2. The magnitude of these two factors P and c-P is called learning rates. Learning rates depend upon the size of B_1 and B_2. If there is a cache hit in B_1, we increment the value of P by 1, otherwise, we increment it by $|B_2|/|B_1|$. Unlike B_1, if there is a cache hit in B_2, we decrease the value of P by 1, otherwise, we decrease it by $|B_1|/|B_2|$.
A new page, which is not present in L1 and L2, is put in the MRU position of L1 and from MRU it is gradually going towards the LRU position as described in Fig 4. Unless the page is being popped out from L1, it never goes to L2, so the pages are not affected in L2. In this way, ARC is scan-resistant. ARC will always affect T_1 and not T_2 and when a scan happens, fewer hits occur in B_1 than B_2; hence the list T_2 will grow [2].

Experiments

Workload	Cache (pages)	Cache (Mbytes)	LRU	ARC	FRCp (Offline)
P1	32,768	16	16.55	28.26	29.39
P2	32,768	16	18.47	27.38	27.61
P3	32,768	16	3.57	17.12	17.60
P4	32,768	16	5.24	11.24	9.11
P5	32,768	16	6.73	14.27	14.29
P6	32,768	16	4.24	23.84	22.62
P7	32,768	16	3.45	13.77	14.01
P8	32,768	16	17.18	27.51	28.92
P9	32,768	16	8.28	19.73	20.28
P10	32,768	16	2.48	9.46	9.63
P11	32,768	16	20.92	26.48	26.57
P12	32,768	16	8.93	15.94	15.97
P13	32,768	16	7.83	16.60	16.81
P14	32,768	16	15.73	20.52	20.55
ConCat	32,768	16	14.38	21.67	21.63
Merge(P)	262,144	128	38.05	39.91	39.40
DS1	2,097,152	1,024	11.65	22.52	18.72
SPC1-like	1,048,576	4,096	9.19	20.00	20.11
S1	524,288	2,048	23.71	33.43	34.00
S2	524,288	2,048	25.91	40.68	40.57
S3	524,288	2,048	25.26	40.44	40.29
Merge(S)	1,048,576	4,096	27.62	40.44	40.18

Table 2 : Comparison of hit ratios of LRU and ARC for various workloads [reference 2, Table VIII]

Table 2 compares the hit ratio of ARC with LRU for all traces explored [2] based on the relevant cache size. It can be easily seen that ARC outperforms LRU quite dramatically. From Table 2, we can also see that ARC performs online very close to FRC_P which works offline.

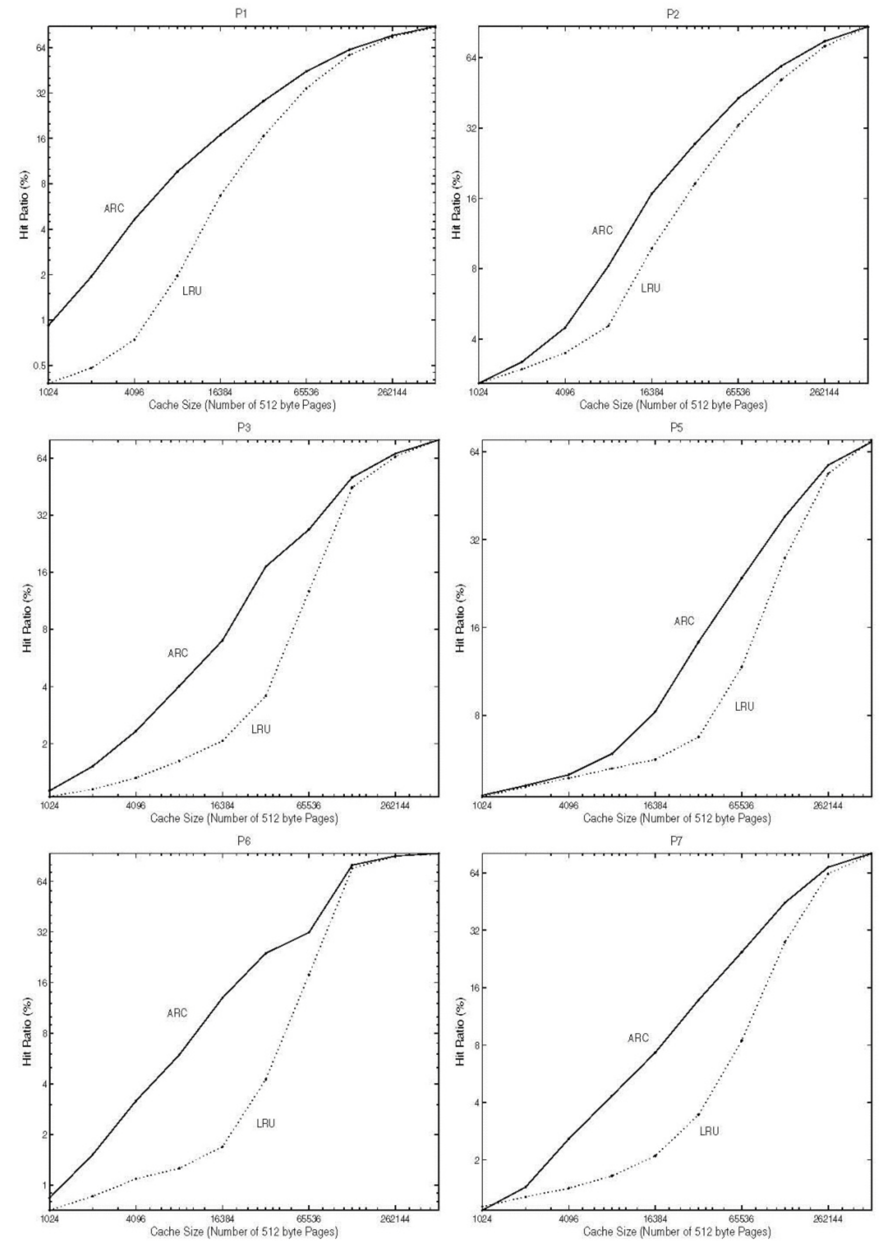


Fig.5 : A plot of hit ratios (in percentages) achieved by ARC and LRU [reference 2, fig.5]

We now plot the hit ratio of ARC and LRU in the Fig.5, where both the x and y axes use logarithmic scale. We plot the hit ratio for P1, P2, P3, P5, P6 and P7. And from Fig.5, it is very clear that ARC performs far better than LRU for all the traces.
ARC is self-tuning and empirically universal. Having a look at Table 2, we can see that ARC tunes itself and performs as well as FRC_P. This holds for all the traces (used in the experiments), so we can say that ARC is empirically universal. Sometimes, ARC works better than FRC_P, for example, in workload P4. However in some cases, ARC is slightly worse, for example, for workload P8. But throughout all the workloads considered, ARC maintains a small value for parameter P. ARC never fixed the value of P, all the time it changes the value of P. Changing the value of parameter P can slightly costs ARC over fixed offline policy in terms of hit ratio. But FRC_P can be never used in real-time applications. Furthermore, ARC provides a reasonable online approximation rather than FRC_P. Thus, using an adaptation replacement policy can produce appreciable performance improvements in modern caches.

4. Buffer Management with a Flash-Based DB

One of the most active research areas is buffer management in the area of database and operating systems. To achieve maximum performance on flash-based devices, we need to discuss about performance characteristics of flash-based devices. The buffer manager decides how and when to access the flash devices, hence we will focus on the buffer management issues for flash devices. Here we will discuss about buffer management algorithms suited to flash-based devices. Flash memory has different read and write cost with respect to time and energy, hence the replacement algorithms for flash memory have to consider both the hit count and the replacement cost caused by selecting dirty victims. The replacement cost of a dirty page is higher than that of a clean page with respect to both access time and energy consumption. In this paper, we consider the Clean-First LRU (CFLRU) replacement algorithm and the Adaptive-double LRU (AD-LRU) algorithm to exploit the characteristics of flash memory. Furthermore, we will outline the Clean-First-Dirty-Clustered (CFDC) algorithm. CFLRU splits the LRU list into the working region and the clean-first region and adopts a policy that expels clean pages preferentially in the clean-first region until the number of page hits in the working region is preserved at a suitable level. CFDC delays the eviction of dirty pages to decrease the flash writes and, hence, improves the page flushing efficiency. Finally, we will compare the performance of the considered algorithms.

4.1 Clean-First LRU Algorithm

The main goal for a buffer replacement algorithm is to minimize the number of physical writes as physical reads are cheaper than writes, considering the physical read/write asymmetry of flash-based devices. For flash devices, the page write cost can be two orders of magnitude higher than that of a page read. For an example, we have p clean pages and q dirty pages and we can assume that p pages are read n times and q pages are modified m times, where both n and m are comparable. Hence, we can replace p in favor of q as the cost of n flash reads is lower than m flash writes. This concept can be referred to as clean-first strategy, where we can evict clean pages early and delay the eviction of dirty pages. The overall cost can be minimized by selecting a clean page for eviction. A clean page contains the same copy of the original data in flash memory, hence the clean page can be just overwritten in the cache when it is evicted by the replacement policy. But it can affect other cost factors. For example, keeping dirty pages in cache as much as possible can decrease the amount of space available in the cache; hence the cache can run out of space. As a consequence, the cache miss rate will increase; hence it will increase the replacement costs of page read requests. For this reason, we need to design an algorithm, which can balance both the costs.

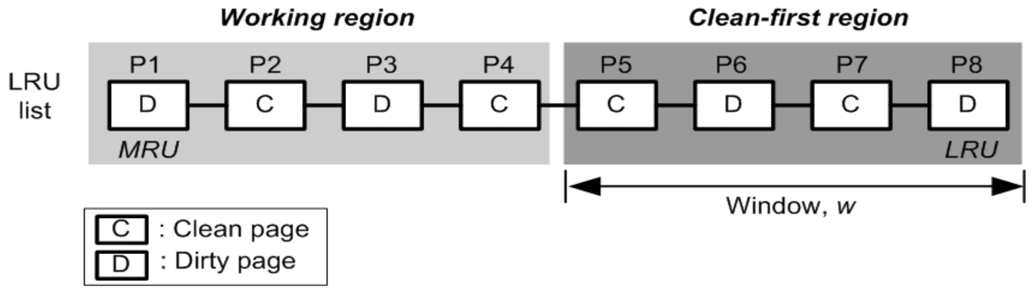


Fig. 6: Example for the CFLRU algorithm [reference 5 , fig.1]

The Clean-First LRU (CFLRU) algorithm is based on the LRU replacement algorithm; it maintains a buffer of pages ordered by access recency. It modifies the LRU policy by dividing the LRU list into two regions for finding a minimal cost point. The two regions are called working region and clean-first region as shown in Fig. 6. The working region is at the MRU end and consists of recently used pages; hence it generates more cache hits. The clean-first region is at the LRU end and consists of the pages which will be evicted. CFLRU always selects clean pages to evict over dirty pages. CFLRU selects a clean page to evict in the clean-first region first to minimize flash write cost. If no clean page is found in this region, a dirty page will be evicted from the end of the LRU list. For example, if we compare CFLRU with the LRU policy, LRU will evict pages in the order of P8, P7, P6, and P5 as shown in Fig. 6. But CFLRU will evict pages in order of P7, P5, P8, and P6.The parameter w is the window size of the clean-first region. It is important to find the right window size of the clean-first region to minimize the total replacement cost. For example, a small window size will increase the number of evicted dirty pages, that is, the number of flash write operations and a large window size will increase the cache miss rate.
CFLRU algorithm is one of the earliest buffer replacement algorithms and it has several disadvantages. First, in case of a buffer fault, the CFLRU algorithm has to work on a long buffer list as it searches for clean pages from the LRU end; which is not always the case. Clean pages are always staying close towards the working region as clean pages are selected in the clean first region over working region. Second, keeping the dirty pages in the clean-first region can shorten the memory resources, because clean pages are more frequently accessed than dirty pages. Third, a main disadvantage is to statically determine the size of w, the window size of the clean-first region. However, we can configure the window size of the clean-first region statically and dynamically. The dynamic CFLRU algorithm has a benefit that we do not have to change the window size each time the workload changes, while we can achieve the similar performance results with the static CFLRU algorithm, by configuring the best performing window size.

Other buffer replacement algorithms proposed in recent years are LRUWSR, CCFLRU, and AD-LRU, which share the same concept with CFLRU. Algorithms like FAB and REF address the FRW (Flash Random Write) problem.
The idea behind LRUWSR (Write Sequence Reordering) is to evict clean and cold-dirty pages and keep the hot-dirty pages in the buffer as long as possible. Searching for a victim page will start from the LRU end of the list. If a dirty page is found, then it will be returned, otherwise the search continues. CCFLRU (Cold-Clean-First LRU) improves LRUWSR by differentiating between cold-clean and hot-clean pages

using the second chance policy. AD-LRU has some more advantages over CFLRU, LRUWSR and CCFLRU, hence we will discuss AD-LRU algorithm in details.

4.2 AD-LRU Algorithm

For buffer replacement algorithms of flash-based DBMSs, we need to consider both the hit ratio and the write cost. The LRU algorithm does not always guarantee to evict clean pages. And the CFLRU algorithm does not deal with the page access frequency which may lead to a decreased hit ratio. Hence to overcome this limitation, the AD-LRU algorithm was developed, which considers recency, frequency, and cleanness (status of clean or dirty page) by the buffer replacement policy.

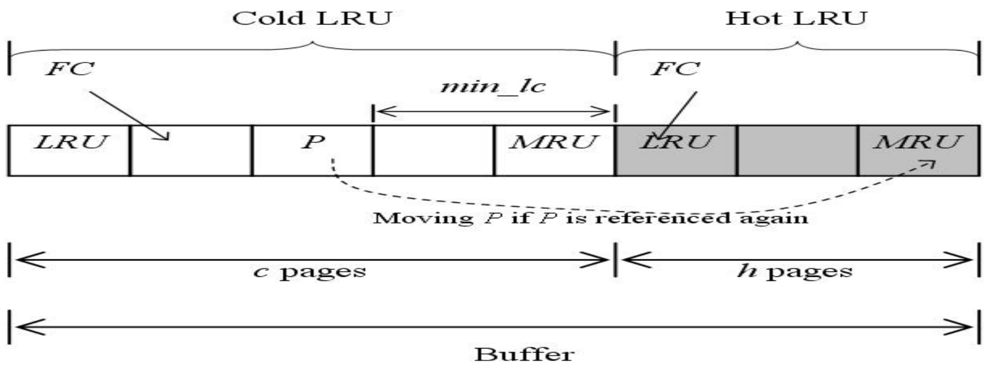


Fig.7: Double LRU lists of the AD-LRU algorithm [reference 7, fig.2]

AD-LRU uses two LRU lists to differentiate between frequency and recency lists. Fig. 7 shows those two lists; one cold LRU list, which stores the pages referenced only once, and the hot LRU list storing the pages referenced at least twice. The sizes of the double LRU queues are dynamically adjusted according to the change in the reference patterns. When a page in the cold LRU list is re-accessed, we increase the size of the hot LRU list and decrease the size of the cold LRU list. Similarly, the size of the hot LRU list is decreased when a page is selected as victim and evicted from there to the cold LRU list. AD-LRU will first evict least-recently-used clean pages from the cold LRU list as shown in Fig. 7, the pointer FC (First-clean) points to the victim page to be evicted in the cold LRU list. We will use a second-chance policy [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] to select a dirty page as the victim; if a clean page does not exist in the cold LRU list. The second-chance policy makes sure that dirty pages in the cold LRU list will not be kept in the buffer for a long period, thus it overcomes the memory space limitation of CFLRU. As shown in Fig. 7, AD-LRU distinguishes the cold and hot pages in the buffer, where the cold LRU list contains c pages and the hot LRU list contains h pages [7]. P denotes an initially referenced page, thus it is put in the cold LRU list and becomes the MRU element. If it is referenced again, then it has to be put in the hot list at the MRU end as shown in Fig. 7. The value of parameter min_lc sets the lowest limit for the size of the cold list. If the size of the cold list reaches the value of min_lc, we have to evict pages from the hot list, not from the cold list anymore. The reason behind this parameter is to set a small list size for cold one as the requested pages will come first to the cold list and are always evicted from this list first. The FC (First-Clean) position indicates the least-recently-used clean page, hence we directly choose the FC page as victim page in the cold LRU list if FC is valid, or we choose a dirty page from the list using a second-chance policy. If the size of the cold list reaches the minimum size (i.e.

min_lc), we evict the victim pages from the hot LRU list using a similar process, i.e. we choose the FC page first or, if FC is null, we select a dirty page based on the second-chance policy [7].

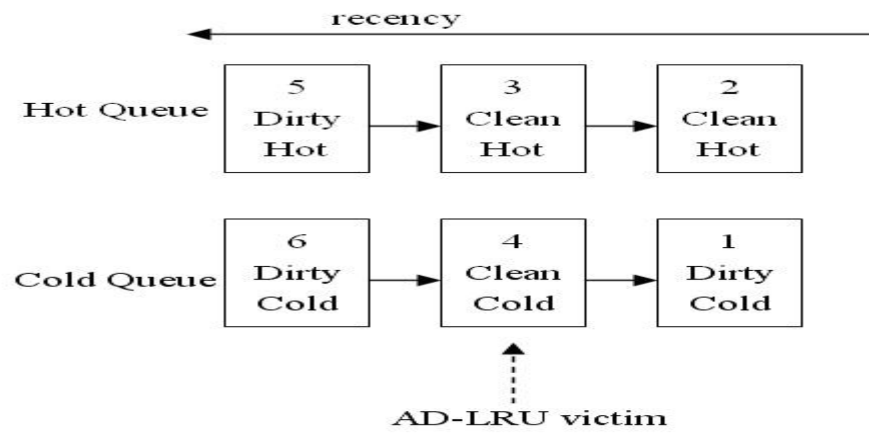


Fig. 8: An Example of Eviction procedure in AD-LRU policy [reference 7, fig.3]

For example, in Fig. 8, we assume that there are six pages in the buffer and the buffer can contain only 6 pages, which means it is full. When the buffer manager receives a new page request, our AD-LRU algorithm will choose page 4 (clean cold) as the victim, as shown Fig. 8. If no clean pages exist in the buffer, then it has to choose a dirty page using the second-chance policy [7].
If we compare AD-LRU with the CFLRU algorithm and the LRUWSR algorithm, CFLRU will select the hot clean page 2 and LRUWSR, which selects the cold dirty page 1 as shown in Fig. 9, hence AD-LRU has a lower number of write operations, because it avoids to evict dirty pages.

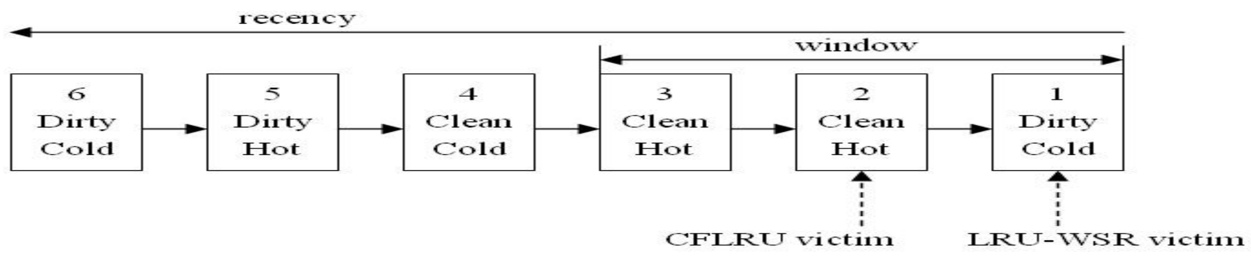


Fig. 9: Eviction procedure in CFLRU and LRUWSR policy [reference 7, fig.4]

The differences between AD-LRU and CFLRU and LRUWSR can be summarized as follows :
First, AD-LRU takes the reference frequency into account, an important property of reference patterns, while this is more or less ignored by CFLRU and LRUWSR algorithms. Thus we can expect a better performance from AD-LRU.
Second, CFLRU keeps dirty pages for a long time which may lead to unnecessary memory resource consumption; this problem is solved by AD-LRU as it cleans up the buffer from cold pages.
Third, AD-LRU is self-tuning. We can set the sizes of the hot and cold lists dynamically according to the workload, but for CFLRU, the window size has to be set statically.
Fourth, Unlike CFLRU and LRUWSR, AD-LRU is scan resistant.
So, we can say AD-LRU keeps its performance advantages compared with CFLRU and LRUWSR.

4.2.1 Performance Evaluation

Now we will compare AD-LRU with other buffer replacement policies. We assume that the page size is 2,048 bytes and the buffer size is between 512 pages to 18,000 pages, i.e. from 1 MB to approximately 36 MB. Parameter min_lc of AD-LRU is set to 0.1 for the Zipf trace and 0.5 for the other traces. The window size w is set to 0.5 for the CFLRU algorithm. We set w=0.5 because, if w=0, CFLRU works like LRU and if w approximates 1, then it can use the entire buffer space to store dirty pages, hence we take the middle value which is 0.5 [7].

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)

Using the parameters in the Tables 3 to 6 [reference 7, Table 3-6], we run the traces for all algorithms. If we compare the hit ratio, then AD-LRU has achieved the best hit ratio.

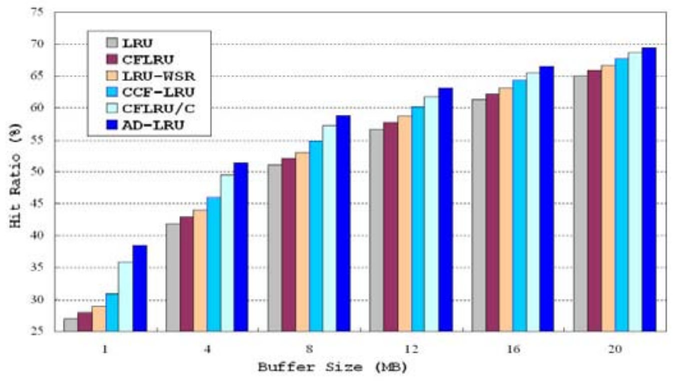
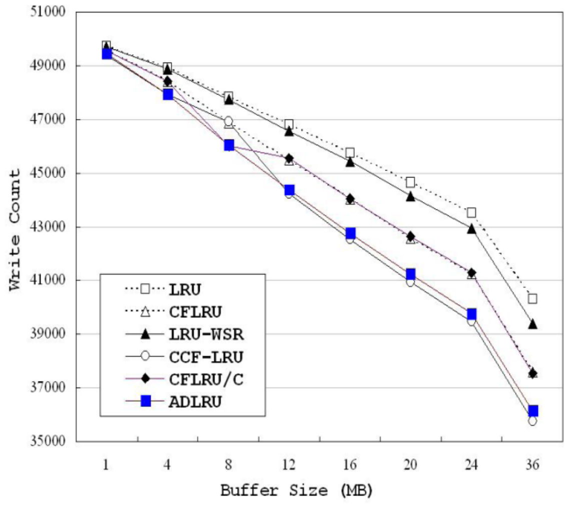
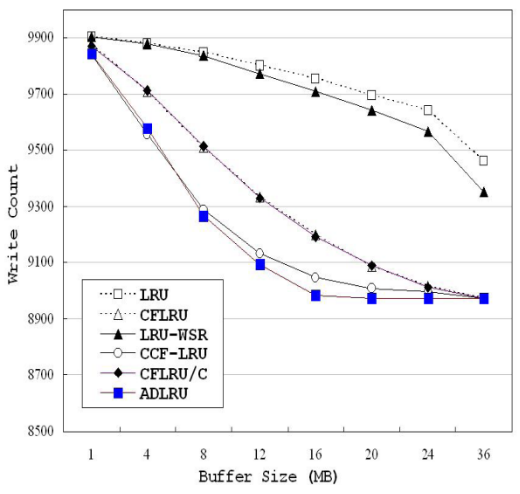


Fig. 10: Hit ratios for the Zipf trace [reference 7, fig.6]

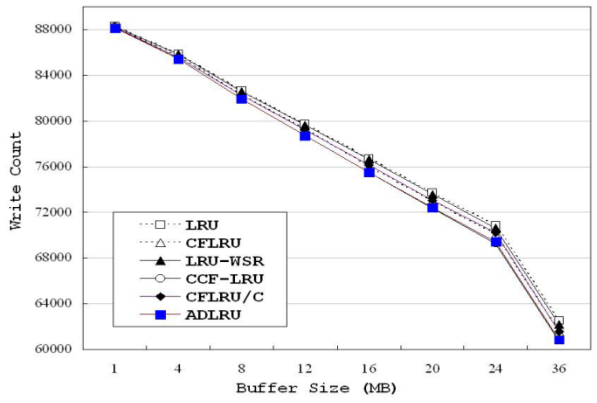
Fig. 10 shows the hit ratio for all buffer replacement policies explored, where AD-LRU has the highest hit ratio among all of them.



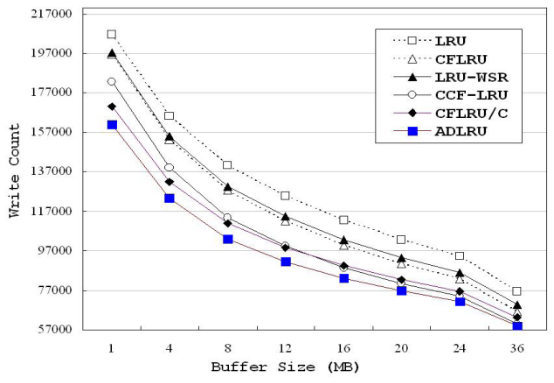
(a) Random



(b) Read-most



(c) Write-most



(d) Zipf

Fig. 11: Write count vs. buffer size for various workload pattern [reference 7, fig.7]

Using four figures (11.a, b, c, d), we want to show the write count vs. buffer size for different workload patterns. As shown in all the four graphs, AD-LRU has the smallest write count and LRU has the highest write count.

Based on such experiments, we can conclude that the LRU algorithm has the worst performance in each case; hence it will not work well in flash-based DBMSs. In contrast, the AD-LRU algorithm achieved superior performance behavior compared other methods proposed so far. It has a lower number of writes counts compared to LRU, CFLRU, and LRU-WSR, both in the simulation environment and in the DBMS-based experiment. Based on these observations, it should be a good choice in flash-based DBMSs to first evict clean pages from the buffer.

4.3 Clean-First Dirty Clustered (CFDC) Algorithm

The existing buffer replacement policy CFLRU has some limitations. It becomes inefficient when the workload is mixed with long and sequential access patterns, because hot pages cached so far are pushed away by sequentially accessed pages. To overcome the problems of existing approaches, a new algorithm called CFDC was developed. The main goals of this algorithm are as follows:
G1: Minimize the number of physical writes [1].
G2:Improve the efficiency of page flushing [1].
G3:Keep a relatively high hit ratio [1].
To improve G1, we can use the basic idea of the CFLRU algorithm. We can overcome the problem by addressing two queues for the clean-first region: one for clean pages and one for dirty pages [1]. A victim page will go to the clean queue if it is clean; otherwise, it will go to the dirty queue. When a buffer fault occurs, then a page of the tail part of the clean queue is returned as a victim, if it is not empty; otherwise, the victim page can be chosen from the dirty queue. This improved CFLRU algorithm eliminates the search costs for clean pages while working in the same way as the original CFLRU algorithm.
To improve G2, we introduce a clustered write policy, which improves the ability of writing to flash. A cluster is a collection of pages located in the same physical neighborhood and having the same cluster number. We can calculate the cluster number by dividing their page number by the constant cluster size [1]. However, page numbers are logical addresses because of the space allocation in DBMS and file systems, nevertheless, the pages in the same cluster have a higher probability of being stored physically adjacent, too. The REF algorithm has a higher Cluster-Switch count (CSC- a metric for spatial locality), compared to the clustered write policy, because REF selects the victim pages from a collection of victim blocks where these blocks can be addressed in any order. Therefore, we denote the approach used by REF as semi-clustered writes [1].
For keeping a high hit ratio (G3), we introduce the generalized two-region scheme of the improved CFLRU method. As shown in Fig. 12, the buffer pool is divided into two regions: working region and priority region. The working region is similar as that of the CFLRU policy, it keeps the hot pages that are frequently accessed. The priority region is responsible for optimizing the replacement costs by assigning different priorities to the pages. The priority window determines the size ratio of the priority region to the total buffer. By dividing the buffer space into two regions, we can maintain a high hit ratio. Both regions in our generalized scheme do not have to be bound to a specific replacement policy.

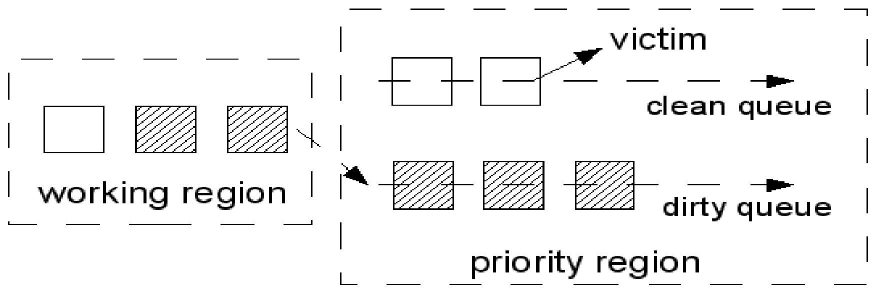


Fig. 12: The generalized two-region scheme [reference 6, fig.2]

Fig. 12 shows the generalized two-region scheme of the improved CFLRU method. Here, the working region uses LRU and the priority region assigns higher priorities to dirty pages. When a buffer fault occurs, a victim page is selected from the priority region to make space for a page currently in the working region. After this page displacement, the requested page can enter the working region [1].

4.3.A1 Page Flow:

We refer to the working region and priority region as W and P, respectively. We consider a parameter λ, called priority window, which determines the size of P with respect to total buffer. So, if the entire buffer has B pages and P has λ. B pages, then the remaining (1 - λ).B pages are managed by W. Fig. 13 describes the page flow in two-region scheme.

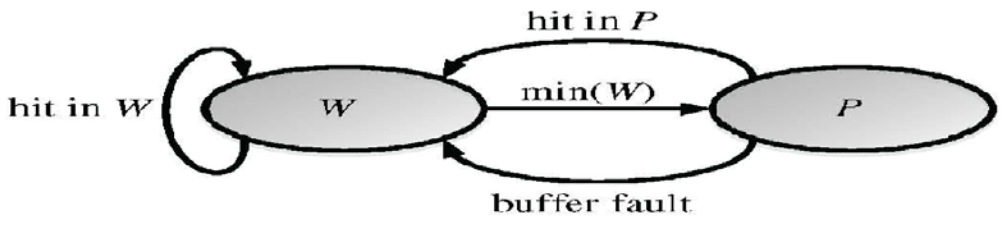


Fig. 13: Page flow in the two-region scheme [reference 1, fig.3.3]

According to the CFDC algorithm, if a page in W is hit, the base algorithm should adjust its data structures accordingly. As an example, if LRU is the base algorithm, it should move the victim page to the MRU end of its list. If a page in P is hit, a page min(W) will be chosen by W's victim selection policy and moved to P and the hit page is moved to W. If buffer fault occurs, the victim page will be always chosen from P [1].

4.3.A2 Priority Region

Three structures are maintained by Priority Region P: one LRU list L_C of clean pages, priority queue Q of clusters contains dirty pages, and a hash table H with cluster numbers used as keys [1]. For a cluster c with n pages (n> 1) in Q, with page numbers P_0,P_1,....P_n,P_{n-1} ordered by their time of entering Q, we define a metric, IPD (inter-page distance) [1], to represent its "randomness":

$$ipd(c) = \sum_{i=1}^{n-1} |p_i - p_{i-1}|$$

IPD differentiates between randomly accessed clusters and sequentially accessed clusters. We prefer to keep a randomly accessed cluster in the buffer for a longer time than a sequentially accessed cluster. For example, a cluster with pages {0, 1, 2,3} has an IPD of 3, while a cluster with pages {7, 5, 4, 6} has an IPD of 5.

For a cluster c with n pages, its priority pr(c) is computed as follows:

$$pr(c) = \frac{ipd(c)}{n^2 \times (globaltime - timestamp(c))}$$ Formula 1 [reference 1, formula 3.2]

The algorithm tends to assign large clusters a lower priority for two reasons:
1. Flash disks are efficient in writing such clustered pages due to their spatial locality [6].
2. The pages in a large cluster have a higher probability to being sequentially accessed [6].
The purpose of the time component in Formula 1 is to prevent randomly and rarely accessed small clusters from staying in the buffer forever. The cluster timestamp(c) is the value of globaltime at the time of it is created. Each time a dirty page is evicted from the working region, the value of globaltime is incremented by 1 [6]. We can calculate its cluster number and perform a hash lookup by using this derived cluster number. If the cluster does not exist, a new cluster is created with the current globaltime and inserted to the priority queue; this cluster contains that dirty page. Otherwise, the page is added to the existing cluster and the priority queue is maintained, if necessary. If a page min(W) is clean, it becomes the new MRU node in the clean list [1].

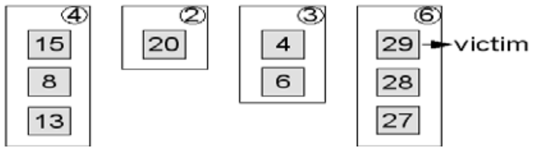


Fig. 14: Prioritized clusters [reference 6, fig.3]

Fig. 14 illustrates a priority queue with four clusters, where globaltime is set to 10, timestamp(c) is at the top right corner of each cluster, and the clustered pages are marked with their page numbers. From left to right, the cluster priorities are obtained using Formula 1: 2/9, 1/8, 1/14, 1/18. The cluster priority will be set to 0, if a victim page is chosen. Therefore if page fault occurs, this cluster will be emptied, such that it is removed from the priority queue. According to the cluster property, the removed dirty pages are logically close to each other and, because of the space allocation in most DBMSs and file systems, also have a high probability of being physically neighbored [6]. So, the write requests received by the flash disk are targeting at a limited number of flash blocks, which can be served effectively. The time complexity of CFDC algorithm is higher than that of the LRU algorithm because of maintenance of the priority queue. However, the queue is maintained in units of clusters and the maintenance is only triggered by a buffer fault and in case the page evicted from the working region is a dirty page.

4.3.1 Experiments using a Synthetic Workload

The synthetic trace simulates typical DB buffer workloads with mixed random and sequential page requests. Four types of page references are contained in the trace: 100,000 single page reads, 100,000 single page updates, 100 scan reads, and 100 scan updates [1]. For our experiments, the DB size is 764 MB. Parameter k was set to 2 for both CFDC-k and LRU-k. Parameter λ is set to 0.5. In our measurements, we varied the buffer size from 500 to 16,000 pages. According to Fig. 15.a, CFDC-k performed better than the other approaches. Furthermore, both CFDC-K and CFDC-1 variants clearly outperform all other algorithms compared. For example, with a buffer of 4,000 page frames, the performance gain of CFDC-k over REF is 26%. Figures 15.b, 15.c and 15.d illustrate detailed performance metrics such as Page-flush count, CSC, and hit ratio.

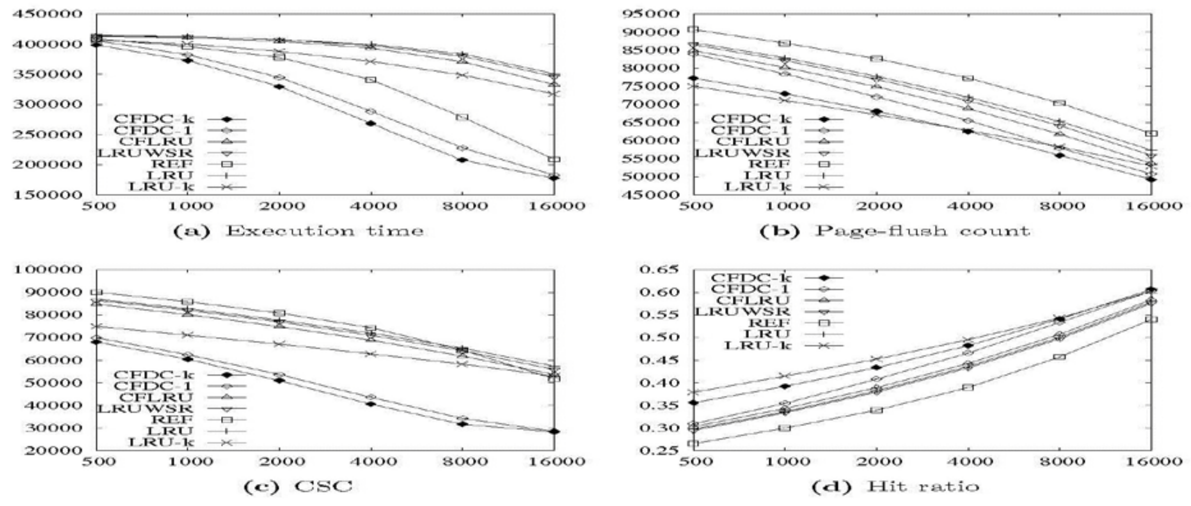


Fig. 15: Synthetic Trace Performance [reference 1, fig.3.5]

4.3.2 Experiments Real-Life Workload

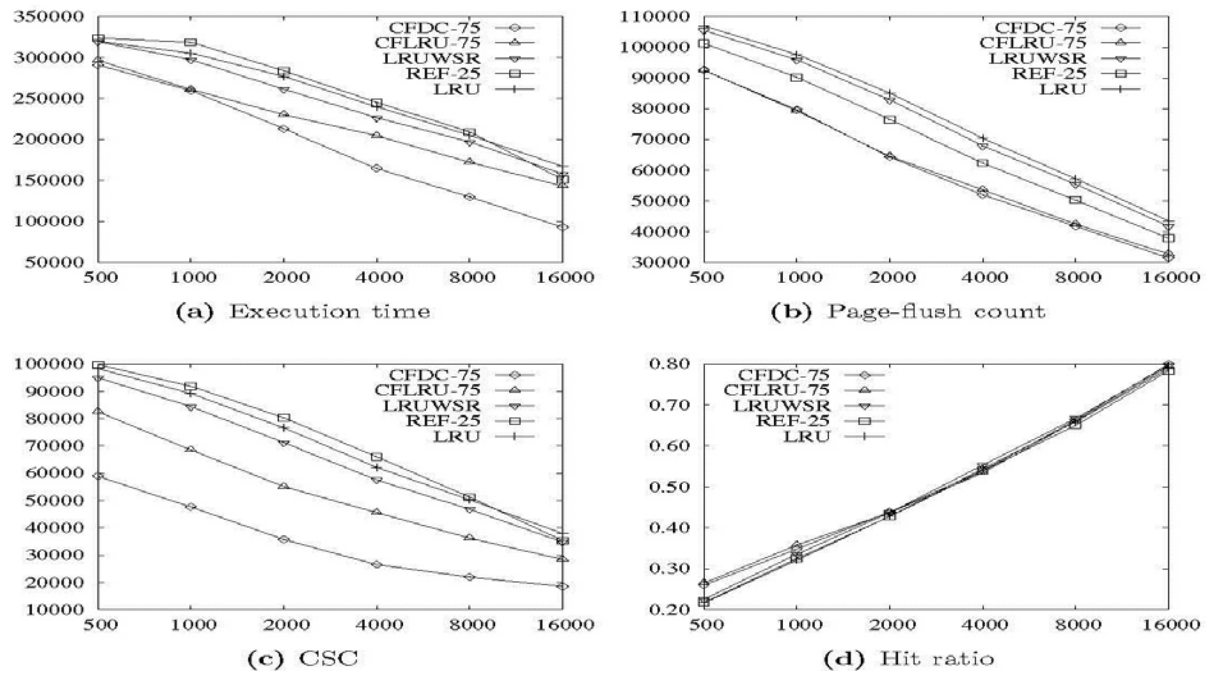


Fig. 16: Performance under real-time workload [reference 1, fig.3.5]

We also performed experiments with the related algorithms using the real-life workload. For each of the algorithms CFDC, CFLRU, and REF, we ran all experiments three times with the window size parameter set to 0.25, 0.50, and 0.75, respectively, denoted as REF-25, REF-50, REF-75, etc., and chose the best performance setting. As expected, Fig. 16 illustrates that CFDC performs best compared to the other competitor algorithms. For example, the performance gain of CFDC over CFLRU is 53% for the 16,000-page settingand 33% for the 8,000-page setting.

Our experiments have shown that the proposed algorithm CFDC significantly outperforms CFLRU and the other competitor algorithms. The reason behind this improvement can be stated as follows: for a flash disk, the number of write operations should be minimized, but, more important, locality of access patterns, especially spatial locality, should be exploited to the extent possible by the buffer management. CFDC maintains a high ratio of buffer hits by implementing a generalized two-region scheme. Another important aspect concerning the performance is the impact of the priority window, provided for both CFLRU and CFDC. If various workload changes are present, it would be necessary to provide a dynamic window size adjustment; however we will not discuss this topic here.

4.4 CASA Algorithm

4.4.1 Introduction

Page read operations from flash SSD are faster than update operations. For this reason, existing buffer management algorithms for those devices usually trade physical reads for physical writes to some extent. Butthey avoid the actual R/W (read/write) cost ratio of the storage devices and fluctuation of the workload. We propose a new algorithm called Cost-Aware Self-Adaptive (CASA) buffer management algorithm, which makes the adjustment between physical reads and physical writes in a controlled fashion, depending on the read/write cost ratio of the storage devices and automatically adapts itself according to the changing workloads. According to most of the existing flash-aware buffer algorithms, a physical write is more expensive than a physical read. But read/write asymmetry is varying from device to device. For an example, Intel X25-V SSD reaches 25K IOPS for reads and 2.5K IOPS for Writes [Intel. X25-V SSD Datasheet, 2010, Intel corp.], however Intel X25-M SSD reaches 35K IOPS for reads and 8.6K IOPS for Writes [Intel. X25-M SSD Datasheet, 2010, Intel corp.]. Ignoring the possible fluctuation of write request during various workload, while making the replacement decision, is a common problem of existing buffer algorithms. However, CFLRU has a parameter for the user to make the replacement decision, but again its selection is difficult because of performance tuning. And for other algorithms like LRUWSR and CCFLRU, it is difficult to choose the victim page (i.e., whether to choose a cold-dirty page or a hot-clean page).

4.4.2 CASA Overview

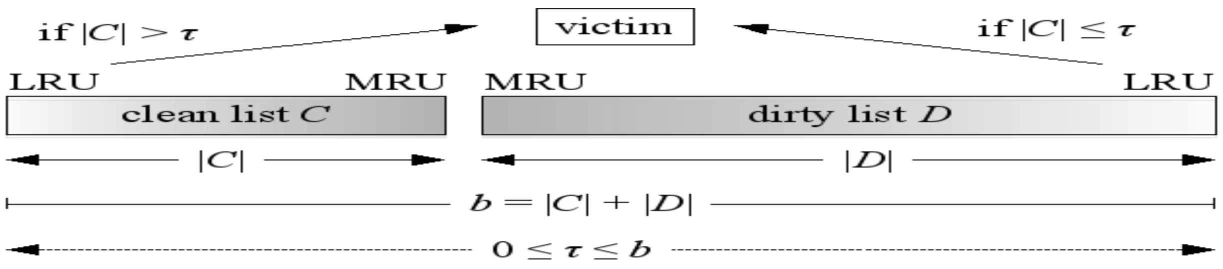


Fig. 17: CASA Algorithm [Reference 4, fig 1]

CASA manages the buffer pool B of b pages using two dynamic lists: the clean list C keeps clean pages which are not modified since being last read from secondary storage devices, and the dirty list D contains dirty pages that are modified at least once in the buffer since being accessed from storage devices. In both lists, pages are ordered by reference recency. Initially both lists are empty. According to Fig. 17, C+D=b where C and D can be a value between 0 and b. CASA adjusts automatically the parameter τ, the maximum size of τ is the size of the C list. So the dynamic size of D is b - τ. If there is a page hit in C, then the current cost effectiveness of C is D/C. And similarly, if a page hit in D, then its cost effectiveness is C/D. According to the cost effectiveness of the current workload, we change the parameter τ [4].

4.4.3 Algorithm

Along with the page requests, the algorithm [reference 4, Algorithm 1] requires as input also the normalized read and write costs, C_R and C_W of the underlying storage devices such that, $C_R+C_W=1$ and C_R/C_W = cost ratio. This value can be derived from cost ratio, which is important to the algorithm in the extent of the read/write asymmetry, but not the exact costs of physical reads and writes. The parameter τ adjusts its value according to cost ratio and relative cost effectiveness. The adjustment is performed in two cases: Case 1, A logical R-request is served in C and Case 2, A logical W-request is served in D. In Case 1, we increase the parameter τ by C_R* (|C|/|D|) where |C| is not 0. The increment combines the "saved cost" of this buffer hit C_R and the relative cost effectiveness (|D|/|C|) [reference 4]. However for Case 2, we decrease the parameter τ by C_W * (|C|/|D|). If a buffer fault occurs, the parameter τ determines the list from where the victim page should be chosen. The actual sizes of both lists are also controlled by the state (clean/dirty) of requested pages. The clean/dirty state of a requested page p is decided by its previous state in the buffer and the current request type (Read or Write). According to the CASA algorithm, if the requested page p is clean then it will be moved to the list C, otherwise to the list D. Hence, the sizes of C and D are dynamically determined by the parameter τ (i.e. CASA controlled the size of the both lists dynamically).

The CASA algorithm requires page requests where the request type must be present. However, this may not be the case in some systems, when a page is first requested without claiming the request type and after some time the page is read or updated. Furthermore, most DBMSs use the classical pin-use-unpin protocol [J. Gray, A. Reuter: Transaction Processing, 1993] for page requests. It is easy to use an update flag, this flag is cleared when the pin call occurs and in the time of actual page update operation the flag value is set. When unpin call happens, the buffer manager knows the request type by checking this flag value.

4.4.4 Dynamic Cost Ratio Detection

CASA automatically optimizes itself at runtime, it has the knowledge concerning cost ratios. We have assumed that the knowledge of cost ratio is available to the algorithm. However it can be provided by the device manufacturer or by the administrator. It would be even more better if, in the future, devices provide an interface for querying the cost ratio online. We can measure online the elapsed time required for each physical I/O request. Hence, this elapsed time can be used to calculate the cost ratio information. However, these measurements can be changed at any point of time. For example, latency of a physical read on magnetic disks depends on the position of the disk arm. On flash SSDs, a physical write operation may trigger a more expensive flash erase operation. Hence, we use an n-point moving average of the measured values for clear understanding of short-term changes, because only the long-term average cost is of interest. Hence, the average cost of the last n physical read (or write) operations is used as the basis for the normalized cost C_R (or C_W) required by the CASA algorithm. Note that no change to the algorithm is required for using the dynamically detected costs. To test the cost-ratio detection technique, we ran traces on WDWD1500HLFS HDD (magnetic disk) and an Intel SSDSA2MH160G1GN flash SSD. We chose n = 32768 for the n-point moving average, because it is large enough to smooth out the short-term fluctuations and its space overhead is small.

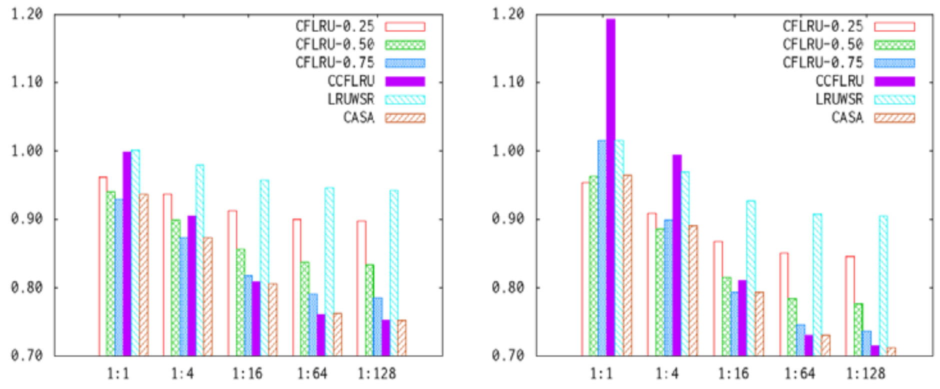


Fig.18: Virtual execution times relative to LRU for buffer sizes of 1,000 and 10,000 pages [reference 4,fig.4]

Fig. 18 shows the virtual execution times relative to LRU running the bank trace for buffer sizes of 1,000 and 10,000 pages. The R/W cost ratio was scaled from 1:1 to1:128.
We chose the buffer size from 1,000 to 10,000 pages and measured the real execution times. In Fig.18, we have shown the relative virtual performance of the related algorithms.
With emerging flash SSDs, the problem of buffer management for storage devices with asymmetric I/O costs is of great importance. To overcome the common problems of existing flash-aware buffer algorithms, we proposed to use the Read/Write cost ratio to capture the R/W asymmetry of the underlying storage devices and presented a cost-aware self-adaptive algorithm called CASA. CASA does not require manual parameter tuning and is efficient. Furthermore, it can adapt itself to various cost ratios and to changing workloads. Furthermore CASA is not only limited to flash-based storage devices, but it can be also applicable to block-oriented storage devices with asymmetric I/O costs.

5. Conclusion

Flash Memory is an advance technology that finds its way into our daily lives on an increasing scale. We have already discussed the advantage of flash memory over HDD and DRAM. We addressed the limitation of flash memory and the implementation issues of flash-based database systems. There is some uncertainty over the behavior of flash devices due to the complexity of the flash transition layer. We proposed some efficient techniques, which address buffer management issues for two-tier storage systems (where flash memory is used as main storage device). Among the proposed techniques we have shown that CASA is probably a more efficient technique, because CASA does not have to deal with parameter tuning and it adapts itself according to various cost ratios. The self-adaptive algorithm SAWC is better when workload changes more frequently. Finally we can say, flash is a relatively simple structure and due to its advantages over conventional storages, the demand of flash is increasing. Furthermore, NAND flash memory is the most advancing scaled technology among electronic devices today.

References:

1. Yi Ou: Caching for flash-based databases and flash-based caching for databases,Ph.D. Thesis, University of Kaiserslautern, Verlag Dr. Hut, Online August 2012
2. Nimrod Megiddo, Dharmendra S. Modha: ARC: A Self-Tuning, Low Overhead Replacement Cache. FAST 2003: (115-130)
3. Nimrod Megiddo, Dharmendra S. Modha: Outperforming LRU with an Adaptive Replacement Cache Algorithm. IEEE Computer 37(4): 58-65 (2004).
4. Yi Ou, Theo Härder: Clean first or dirty first?: a cost-aware self-adaptive buffer replacement policy. IDEAS 2010: 7-14
5. Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jinsoo Kim, Joonwon Lee: CFLRU: a replacement algorithm for flash memory. CASES 2006: 234-241
6. Yi Ou, Theo Härder, Peiquan Jin: CFDC: a flash-aware replacement policy for database buffer management. DaMoN 2009: 15-20
7. Peiquan Jin, Yi Ou, Theo Härder, Zhi Li: AD-LRU: An efficient buffer replacement algorithm for flash-based databases. Data Knowl. Eng. 72: 83-102 (2012)
8. Suman Nath, Aman Kansal: FlashDB: dynamic self-tuning database for NAND flash. IPSN 2007: 410-419
9. Kyoungmoon Sun, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H. Noh, Sang Lyul Min: LTFTL: lightweight time-shift flash translation layer for flash memory based embedded storage. EMSOFT 2008: 51-58
10. Nimrod Megiddo, Dharmendra S. Modha: System and method for implementing an adaptive replacement cache policy, US 6996676 B2, 2006.
11. Wikipedia: Adaptive replacement cache
12. Wikipedia: Page replacement algorithm
13. Wikipedia: Cache algorithms
14. Wikipedia: Flash memory
15. Flash Memory DBMS for Transactional Database Applications(FMDB) - http://www.cs.arizona.edu/projects/fmdb/overview.html