Computing and Informatics, Vol. 38, 2019, 1237–1271, doi: 10.31577/cai_2019_6_1237

PROBABILISTIC PAGE REPLACEMENT POLICY IN BUFFER CACHE MANAGEMENT FOR FLASH-BASED CLOUD DATABASES

Atul O. Thakare, Parag S. Deshpande

Department of Computer Science and Engineering
Visvesvaraya National Institute of Technology
South Ambazari Road, Nagpur, Maharashtra
India – 440010
e-mail: aothakare@gmail.com, psdeshpande@cse.vnit.ac.in

Abstract. In the fast evolution of storage systems, the newly emerged flash memory-based Solid State Drives (SSDs) are becoming an important part of the computer storage hierarchy. Amongst the several advantages of flash-based SSDs, high read performance, and low power consumption are of primary importance. Amongst its few disadvantages, its asymmetric I/O latencies for read, write and erase operations are the most crucial for overall performance. In this paper, we proposed two novel probabilistic adaptive algorithms that compute the future probability of reference based on recency, frequency, and periodicity of past page references. The page replacement is performed by considering the probability of reference of cached pages as well as asymmetric read-write-erase properties of flash devices. The experimental results show that our proposed method is successful in minimizing the performance overheads of flash-based systems as well as in maintaining the good hit ratio. The results also justify the utility of a genetic algorithm in maximizing the overall performance gains.

Keywords: Cloud databases, data mining, flash solid state drives, adaptive optimization, genetic algorithm

Mathematics Subject Classification 2010: 68Uxx

1 INTRODUCTION

1.1 Flash Based Systems

Most storage systems built on conventional hard disk drives (HDDs) suffer from technical limitations, such as low random access performance and high power consumption. The mechanical nature of HDDs prevents these problems to get addressed via technology evolution. Flash memory based Solid State Drive (SSD) is a type of electrically erasable and programmable read-only memory (EEPROM) which plays a crucial role in revolutionizing the storage system design. With the continuous decrease in price and increase in storage capacity, the usage of SSDs is growing in mobile devices, embedded computing and portable devices such as PDAs (personal digital assistants), digital audio players, digital cameras, HPCs (handheld PCs), etc. Due to its several other attractive features like high reliability (shock resistant), high I/O performance (fast random reads) and low power consumption it is also considered for replacing magnetic hard disks in enterprise database servers.

Unlike rotating hard disk drives which have mechanical movement overheads measured in terms of seek time and rotational latency, the flash SSD requires no movement of any of its mechanical part while read or write access to it. Unlike hard disks which have nearly uniform read-write cost, random writes to SSDs are much slower than random reads from it because of its erase-before-write limitation. In SSDs typically, write operations are about ten times slower than read operations, and erase operations are about ten times slower than write operations [13].

The storage space of flash memory is divided into fixed-sized blocks, each one containing fixed-sized pages. Hence block size (usually 16 KB) is multiple of page size (usually 512 bytes). Read/write operations are performed in units of pages whereas erase operation is performed in units of blocks. Flash memory has the typical property of writing a page only once, because of which page data cannot be updated in-place. Hence when some contents of a page need to be modified, the entire page must be written into a free page slot and the old page contents need to be invalidated (out-of-page updates) [25]. The invalidated page can be written only after erasing the block containing that page, i.e., a written memory must be erased before it can be written again. SSDs use an additional driver software called flash translation layer (FTL) which works in coordination with the host operating system, and whose function is to map the logical blocks to the physical pages on the flash device [27]. SSDs have limited processing power and random access memory (RAM), to manage relatively much larger sized flash memory. To achieve good performance with limited resources, FTLs are designed to hide the mismatch between the write and the erase operations, by exploiting the localities in the write requests. But in case there is a high percentage of random write requests, the performance of SSDs can drop significantly. Especially in cloud environments where the page access patterns are highly complicated due to numerous clients with diverse requirements accessing the cloud storage simultaneously, the random write performance of SSDs is extremely crucial.

The erase operation is performed by garbage collection policy, whenever a sufficient number of free slots are not available in the flash memory. Hence the increase in the number of writes will cause an increase in the number of erase operations, which are slowest among the 3 flash operations. Another unique feature of flash memory is limited erase count, i.e., a limited number of erase operations are allowed to be carried on each block. After a specified number of erase operations (in the range 10 000 to 1 000 000 depending on its physical characteristics) the block will become unreliable. Due to this, the lifetime of flash memory is shorter than that of devices like the hard disk.

Moreover, due to the electrical properties of flash cells, garbage collection overheads are higher in the case of random writes as compared to sequential writes. If the write requests are randomly distributed over the logical block address space, sooner or later all the physical blocks in flash memory will be fragmented, severely damaging the performance of garbage collection by increasing I/O latencies [15].

Based on the above facts we can conclude that, in flash SSDs, replacement of a dirty page can induce more cost than the cost of replacement of a clean page. This reduces the guarantees of optimizing I/O performance with the reduction in the miss rate. In other words the relationship between I/O cost and miss rate may not be consistent in the flash disks. Hence the effectiveness of buffer management schemes can be measured in terms of hit rate and average I/O service cost per page fault. The average I/O cost depends on flash memory characteristics as well as read-write patterns in the workload.

As flash memory is becoming a promising alternative to replace hard disks, like other software systems, the database systems also need to device techniques to cope with flash I/O properties to reduce I/O latencies. Unfortunately, most of the existing disk-oriented buffer replacement algorithms aim at better performance by considering

1. read and write operations equally, i.e., having the same latency,
2. hit ratio improvement as the only means of performance improvement,

due to which hard disk-based DBMS buffer managers are inefficient to deliver good DBMS performance on flash-based systems.

While designing the buffer management policy for flash-oriented systems, its asymmetric read-write feature needs to be taken into account in addition to the locality of pages in the main memory. Hence the buffer management schemes for flash memory need to aim at not only the better hit ratios but also to minimize the replacement costs incurring when a dirty page has to be propagated to flash memory to make room for a requested page currently not in buffer. In other words, the replacement policy should be able to minimize the number of erase operations by controlling the number of write operations on flash memory and, at the same time, avoid a significant fall in the hit ratio, because the fall in the hit ratio will lead to an additional increase in the number of read operations.

1.2 Cloud Databases

Cloud computing has emerged as an important computing paradigm which refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services. The services offer facilities for data storage, data processing, and information management [26]. The services have been referred to as Software as a Service (SaaS), IaaS (Infrastructure as a Service) and PaaS (Platform as a Service). A cloud database is a database that typically runs on a cloud computing platform (private, public or hybrid), access to it is provided as a service.

Nowadays many applications are hosted on cloud databases where several applications share the same database instance. Such a database management system exhibits periodic behavior in terms of data references. For example, US customers access data at a particular time while Japanese customers access data at some other time. The periodicity of data references is translated into periodic page references. This periodicity of page references can be used as a new parameter to improve cache performance by improving page replacement policy.

1.3 Our Contributions

To meet the above-mentioned challenges on cloud-based flash memory devices, we propose two cost-based adaptive buffer-management algorithms based on the probabilistic model of page references, namely Accurate Probabilistic Adaptive Clean First Algorithm (APR-ACF) and Approximate Probabilistic Adaptive Clean First Algorithm (PR-ACF). Our algorithms focus on achieving the following objectives:

1. Reducing the number of write/erase operations by considering the read-write cost ratio of flash disks at each replacement.
2. Maintaining the hit rate as high as possible by considering the recency, frequency, and periodicity of page references.

An additional feature of our algorithms is that it is self-tuning to respond to changes in page reference patterns, by analyzing the frequency, recency, and periodicity of page references in an online as well as offline manner.

Summarizing:

- We have introduced the idea of the periodicity of page references considering recent trends in database applications, especially in a cloud environment.
- We have designed the concept of buffer management based on the probability distribution of page reference which is more generalized and is capable to map any existing specialized algorithms.
- The probability of reference is used to calculate which pages may be referenced soon. The probability of reference gives us the information about pages in the buffer cache, which one of them is a cold/warm/hot page.

- Our flash aware buffer management scheme prefers to make replacement in the COLD REGION; a part of the buffer cache which contains the pages with a low probability of reference.
- Within the COLD region it prefers to replace the CLEAN page with the lowest probability of reference. In the absence of a CLEAN page in the COLD region, it replaces the DIRTY page with the lowest probability of reference.
- Our algorithm is designed to retain in the buffer:

  1. The dirty pages with medium to high reference probability (warm/hot dirty pages).
  2. The clean pages with high reference probability (hot clean pages).

- Our algorithm is designed to quickly remove from buffer:

  1. The dirty pages with low reference probability (cold dirty pages).
  2. The clean pages with low to medium reference probability (cold/warm clean pages).

  If a conflict arises, clean pages are preferred over dirty pages for replacement. The hotness or coldness of a page is decided based on its current probability of reference. Here, the reference probability of a page is computed based on recent references and past historical references, as described in Section 3.
- Our algorithm adapts with the changing page reference patterns by dynamically resizing the COLD and HOT regions of the buffer cache.

The remainder of this article is structured as follows. The related works in this field of cache management in hard disks based systems and flash-based systems are introduced in Section 2 before describing the proposed buffer replacement algorithms APR-ACF and PR-ACF in Section 3. Section 4 describes a page access probability model and Section 5 explains the proposed page replacement algorithm using the probabilistic model described in Section 4. Section 6 presents cost-benefit analysis, whereas Section 7 explains a practically more efficient version of the page replacement algorithm described in Section 5. The detailed analysis of the simulation experiments and the results on various traces of different characteristics are explained in Section 8. Section 9 explains the need for finding the optimal set of time intervals, its application in maximizing the overall performance benefits, and the use of genetic algorithms in defining the optimal time intervals.

2 RELATED WORK

2.1 Literature Survey

Since the database management system is accessed by various types of users and data is stored in different types of storage structures, more complex referencing patterns

are seen. Such page referencing patterns are categorized as sequential references, random references, hierarchical references and looping hierarchical references [1]. Such referencing patterns describe a new query behavior model, the query locality set model [QLSM] and based on it, a buffer management algorithm DBMIN was proposed by Chou and DeWitt [2]. Based on the looping behavior of the operations the hot set model is proposed by Sacco and Schkolnick [3]. The domain separation algorithm proposed by authors in [3] divides the buffer pool into several domains. Each domain represents a separate data structure like B-Tree or Cluster. So when B-Tree is accessed it is ensured that non leaf parent node always resides in memory. Another problem in database cache optimization is handling infrequent long queries. If the query is infrequent and requires lot of disk page access then to bring the required disk pages in the cache, whole existing buffers will be replaced. This will lead to removal of existing and stable working set in the cache and can result in lot of cache misses afterwards. Nowadays most of the database instances are residing in cloud environment and same database instance can be shared by multiple users across the globe. Each such user may not access whole data but some specific set of data. For example users from U.S. may access data of baseball products, while the users from Brazil may be interested only in football products. So if the data of such products is stored in clustered table and in data pages $\langle B_m \dots B_{m+i} \rangle$ [pages having data of baseball products] and $\langle B_n \dots B_{n+j} \rangle$ [pages having data of football products], then the pages, $\langle B_m \dots B_{m+i} \rangle$ may be referred more frequently in 12–18 UTC while pages $\langle B_n \dots B_{n+j} \rangle$ may be referred more frequently in 17–23 UTC. So the page referencing pattern shows certain periodicity and can be modeled using some probability distribution such as Gaussian distribution. Due to advent of superior hardware and advanced tracing systems in database, the detailed historical page trace is available indicating page number and page reference time which can be processed offline to construct probability distribution of page reference.

Nowadays different storage options like index organized tables, clusters, indexes, and partitions are available for storing data. In such structures, all data related with same key are stored in the same page or in pages that are physically contiguous to each other. Same query may generate different referencing pattern if the underlying storage structure is different. The heterogeneity of users in terms of their likings, their operating time, 24×7 application environment and different underlying storage structures has generated many complex reference patterns and conventional buffer management algorithms, which are based on one or two reference pattern(s), may not provide better solution for buffer management.

The efficient buffer management system demands accurate estimation of future probability of reference. The estimation should consider heterogeneity of users and their operating time, different storage structures and current pattern of references. Such estimation requires analysis of past references which normally provided by database management systems and decision of replacement should be taken using complete past pattern rather than using single event like LRU. Modern machine learning techniques can be used to estimate probability distribution and allows us to map the problem of replacement as classification problem. The only argument

against such methodology is time required for such complex decision making which is very critical in cache management. Modern hardware can provide better implementation of such algorithms and the improvement in hit ratio, which may substantially reduce disk access, may outweigh cost incurred due to complex buffer management.

2.1.1 Traditional Buffer Replacement Algorithms

Traditional replacement algorithms, which are hard disk-oriented buffer management algorithms, primarily focus on the hit ratio for good performance. Many algorithms have been proposed so far, most of which are based on the recency and/or frequency property of page references. Among them, the best-known algorithms are LRU, CLOCK [17], LRU-2 [18], 2Q [21], LRFU [19], etc. Few more of them are algorithms like LIRS [20], ARC, CAR [15], CART, Clock-Pro [22], etc., with an additional adaptability, i.e., self-tuning feature.

2.1.2 Buffer Management Algorithms for Flash-Based DBMSs

Most of the asymmetry-aware flash buffering algorithms indicate two design points:

1. Distinguish clean pages (pages which contains the same copy of the original data in flash memory; need not be written back on flash memory at the time of replacement) and dirty pages (pages which are modified after reading the original data from flash memory into main memory; hence need to be written back on the flash memory whenever they are replaced).
2. Compare the locality of the two kinds of pages to make the replacement decision.

Most of these algorithms try to reduce the number of write operations, by delaying the process of evicting the dirty pages to enhance the I/O performance. One of the first flash-based buffering algorithm is proposed by Park et al. [7], namely CF-LRU which is based on the principle of holding the dirty pages as long as possible. For giving additional stay to dirty pages it enforces quicker replacement of clean pages. Li et al. [13] published the Cold-Clean-First LRU (CCF-LRU) algorithm which evict the cold clean pages first. It gives priority to replacing clean pages, what many times results in quick replacement of newly inserted pages, thus reducing the hit ratio. In absence of clean page, it prefers to replace cold dirty pages. Based on CCF-LRU, Jin et al. [8] designed Adaptive Double LRU (AD-LRU) algorithm to further improve the runtime efficiency. AD-LRU maintains the cold and hot pages in two separate LRU queues and dynamically adjusts their length according to the reference patterns. Jung et al. [16] propose the LRU-WSR algorithm which uses a Write Sequence Reordering (WSR) strategy. LRU-WSR prolongs the stay of hot dirty pages in the buffer and prefers replacing clean pages or cold-dirty pages. The authors On et al. [12] develop a FD-Buffer algorithm in which clean and dirty pages are separated into two pools. The size ratio of the two pools is dynamically adjusted based on the read write asymmetry property of the flash memory and the runtime workload. Prober scheme [10] is efficient to exploit the workloads which

contain large sequential write sequences, especially in the write-dominant traces. To address the problem of cache pollution, Prober identifies large sequential write request at early stage and enforces its quick replacement by labeling it as a cold page. By monitoring I/O access patterns at runtime, Hystor [11] can effectively identify pages that can result in long latencies or are semantically critical (e.g. file system metadata), and stores them in SSDs for future accesses to achieve a significant performance improvement. To further enhance write performances, Hystor also serves a write-back buffer to speed up write requests.

3 PROPOSED WORK

3.1 Estimating Probability of Reference

In the proposed work we have provided a method for estimating the probability of reference of the page in any time interval and suggested buffer management algorithm based on it. We have also shown how different types of patterns can be accommodated in this model and how algorithms like LRU and LFU can be simulated by changing different algorithm parameters. We have also discussed the cost-benefit analysis and analyzed to find out under what conditions the current strategy provides benefit to cost ratio.

Moreover the proposed page referencing model essentially captures most of the page referencing behavior [24]. The proposed method is based on the following assumptions which are valid in most of the practical conditions.

1. The 24×7 internet-based database systems show periodicity of references due to different users across the globe.
2. The queries are made efficient by using various storage structures that store the related data physically adjacent to each other.
3. The query load generally comprises periodic frequent queries and infrequent queries.
4. Based on the estimated probability of references we can determine cold, warm and hot pages in the buffer cache.
5. Performance of flash-based database system can be optimized by modifying the buffer management policy to minimize the replacement of hot or warm dirty pages as well as hot clean pages if possible.

4 MODELING PAGE ACCESS PROBABILITY

The probability model [5] of page reference is based on recent references of the page and periodic references of the page which are captured from past data in a specific time interval. Generally in managing buffer cache, the page with a lower probability of reference is replaced and page with a higher probability of reference is kept in the

buffer cache. The probability of future reference from current time τ_c to future time $\tau_c + \Theta$ is estimated using following principles:

1. If the page is referred frequently in the recent past, then its probability of reference in the future is higher. The probability estimated using the principle is denoted as R_{b_i,t_1,t_2} where b_i is page id and $t_1 - t_2$ ($t_2 > t_1$) indicates time interval.
2. If the page is referred frequently in some interval in the past, then there is higher probability of referencing it in future in the same interval. The probability estimated using this principle is denoted as H_{b_i,t_1,t_2}.
3. The total probability of reference T_{b_i,t_1,t_2} is calculated as weighted sum R_{b_i,t_1,t_2} and H_{b_i,t_1,t_2}. Thus $T_{b_i,t_1,t_2} = \omega_1 R_{b_i,t_1,t_2} + \omega_2 H_{b_i,t_1,t_2}$ where $\omega_2 + \omega_1 = 1, \omega_i \in$ Domain of real numbers between 0.0 and 1.0.

In calculating probability in all cases, probability distribution is estimated using Parzen window technique with Gaussian kernel [4]. This technique provides all the essential properties to capture behavior of page referencing pattern which is explained in the remaining part of this section.

4.1 Calculating Probability R_{b_i,t_1,t_2} Based on Recent References

Using Parzen window classifier with normal distribution [4], probability density estimation function of each page is estimated (Parzen window is used because the approximate estimation can be chosen by changing distribution parameters).

Let

- N be total number of page references in time period t.
- t be total time period of data collection specified in small time units. For example, if data is collected for 10 days and time unit is minutes then $T = 10 * 1\,440 = 14\,400$ where 1 440 is the number of minutes for one day, i.e., 24 hours.
- τ_i be time instance when page b is referred to in the specified interval.
- $S = \tau_1, \tau_2, \tau_3, \tau_4, \ldots, \tau_k$ indicate the set of time units where a page is referred. For example, if a page is referred at 11 am on day 1 and 11.05 am on day 2 then $S = \{660, (1\,440 + 665)\}$.
- P be periodicity of reference. For example, page referencing pattern is repeated for each day, then $P = 1\,440$, i.e., the number of time units in one day.
- σ be a user-defined parameter which controls the effect of past reference on probability. The higher value of σ indicates bigger effect of reference on the future pattern. For experimentation, value of σ is chosen as 1.

Hence, if page b_i is accessed at time $\tau_1, \tau_2, \ldots, \tau_k$ then probability density function of page b_i is

$$P_{bi} = \frac{1}{N} \sum_{i=1}^{k} \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-1}{2}\left(\frac{t-\tau_i}{\sigma}\right)^2}. \tag{1}$$

$P_{bi} = 0$ if b_i is not referred in the past. Probability of page reference b_i in the interval t_1 to t_2 is

$$P_{b_i,t_1,t_2} = P_{b_i}(t_1 \le t \le t_2) = \int_{t_1}^{t_2} \left(\frac{1}{N} \sum_{i=1}^{k} \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-1}{2}\left(\frac{t-\tau_i}{\sigma}\right)^2} \right) \mathrm{d}t. \tag{2}$$

The density function satisfies following essential properties of probability function:

$$P_{b_i}(t_1 \le t \le t_2) \neq 0 \quad \text{if } t_2 > t_1, \tag{3}$$

$$P_{b_i}(-\infty \le t \le \infty) = \frac{N_K}{N} \tag{4}$$

where N_K is total historical references of page b_i. If $B = b_1, b_2, \ldots, b_n$ is set of all pages then

$$P_{b_i}(-\infty \le t \le \infty) = \sum_{i=1}^{n} P_{b_i,-\infty,\infty} = \frac{1}{N} \sum_{i=1}^{n} N_k = 1. \tag{5}$$

4.2 Calculating Probability $\mathbf{H_{b_i,t_1,t_2}}$ Based on Past References of Historical Data

In this case, the time period is divided into fixed interval of equal size. For example if the time period is a day then it is divided into time intervals of one hour, so number of time intervals is 24 from 0–1 am, 1–2 am, etc. For practical implementation, they are represented using number of minutes from the start of period as 0–60, 61–120, 121–180, etc. If the page is referred many times in the past in some interval then it is also likely to be referred in the same interval in future. For each such interval working set of pages is calculated as set of top N pages in terms of probability of reference in that interval.

Here probability density function is similar to previous case but instead of using τ_1, τ_2 as absolute time, the time is calculated from start of the period. For example if page is referred on day 1 at 5:30 pm and on day 2 at 5:45 pm then τ_1 & τ_2 are taken as 1 050 and 1 065 minute, i.e. number of minutes from start of the period. The probability density function of reference of page b_i is indicated as

$$H_{b_i} = \frac{1}{N} \sum_{i=1}^{k} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-1}{2} \left(\frac{\mid t - \tau_i \mid}{\sigma} \right)^2 \right) \tag{6}$$

where $\mid t - \tau_i \mid$ indicates time difference between t and τ_i considering circular scale. The probability of reference between time period t_1 and t_2 is

$$H(t_1 \le t \le t_2) = \int_{t_1}^{t_2} H_{b_i} \mathrm{d}t. \tag{7}$$

4.3 Finding Out LRU (Least Recently Used), MRU (Most Recently Used), MFU (Most Frequently Used) Buffers from Probability Model

The above probabilistic model of page reference is able to capture various types of buffers which may be used to implement existing conventional algorithms. Many of the replacement policies consider least recently used buffers (LRU), most recently used buffers (MRU), frequently used buffers (FUB) as criteria for replacement or stay in buffer cache. These buffers are determined using following formulas. The least recently used buffer (LRU) can be found out using Equation (8):

$$B_{LRU} = {}_{i}^{argmin} P_{b_i,t_c,\infty} \tag{8}$$

where $P_{b_i,t_c,\infty}$ is calculated based only on most recent reference of b_i. The most recently referred buffer (MRU) can be found out using Equation (9):

$$B_i = {}_{i}^{argmax} P_{b_i,t_c-\theta,t_c} \tag{9}$$

where $\sigma \gg \theta$.

The most frequently used buffer (MFU) can be found out using Equation (9) with the following constraints on the value of σ:

$$0 \ll \sigma < \theta.$$

5 PROPOSED ALGORITHMS

Based on the above probabilistic model of page reference the proposed algorithm is designed as follows:

- Divide the period into number of intervals T_0, T_1, T_2, ...
- For each interval calculate the historical working set based on SQL traces as given in Section 4.2 and Section 5.1.
- For each page in working set, calculate probability according to Equation (9). For fast access the information of page and its probability is kept in a hash table.

5.1 Selecting Pages for Working Set

For each time interval, pages are sorted in descending order based on their computed probability value in that interval and first k pages are chosen where k is the size of buffer cache. The pages are chosen only if their probability is higher than predefined threshold.

5.2 Assigning Ranks to Working Set Pages

Within each working set, pages are given ranks based on their probability values.

5.3 Buffer Cache Organization

- Maintain list of pages which are referred in past along with their time of reference. This list is used to calculate probability R_{b_i,t_1,t_2} according to equations given in Section 4.1. Since it is not possible to store all past references, the size of list is m times the size of buffer cache where $m > 1$.
- Maintain list of pages in cache with their total probability value. The list is sorted in descending order of their probability value. The page having lowest probability is at the one end of the list which is called as COLD END while page having highest probability is at another end which is called as HOT END. The pages at the COLD END and HOT END are referred as '*bc*' and '*bh*', respectively.
- The variable SPLIT divides the buffer cache into two parts which are HOT REGION and COLD REGION. Initially SPLIT is initialized to 50 % of the buffer cache size, hence it divides buffer cache into two equal parts. The first part of the buffer cache starting with HOT END is called as HOT REGION and the remaining part ending with COLD END is called as COLD REGION. The other END of the HOT REGION opposite to HOT END is called as WARM END.
- In order to keep adaptability with changing reference patterns, we maintain a LRU list (LIST1) of recently replaced pages from the buffer cache. The length of the list is same as the length of the buffer cache.

5.4 Accurate Probabilistic Adaptive Clean First Algorithm (APR-ACF)

When page b_i is referred at time T_i then following procedure is called (assume buffer full condition). We call this algorithm as Accurate Probabilistic Adaptive Clean First Algorithm (APR-ACF).

Replacement (b_i,T_i)
Begin
If (b_i is not in buffer cache)

1. scan the COLD REGION from the COLD END towards the WARM END for the first CLEAN page.
   - if found then replace first CLEAN page with b_i. (a)
   - if not found then replace the COLD END page with b_i.
2. Add replaced page to MRU end of LIST1. If LIST1 is already FULL, delete the LRU END page from it, and add replaced page to MRU end. (a1)
3. Update probabilities of each page with new time reference T_i. (b)
4. Sort the list. (c)

5. resize() (d)
6. **End**

Following procedure resize() adapts with the changes in the reference patterns of the working set pages and normal pages. This is achieved by resizing the COLD and HOT regions at each page reference as follows:

resize ()

- If referred page b_i is a working set page and is present in the LIST1 (list of recently replaced pages), then update variable SPLIT to decrease length of COLD REGION by 5 % of the buffer cache length. This will increase the length of HOT REGION by the same 5 %.
- If referred page b_i is a normal page (does not belong to working set of current time interval) and is present in the LIST1, then update variable SPLIT to increase length of COLD REGION by 5 % of the buffer cache length. This will decrease the length of HOT REGION by the same 5 %.
- The range of values that SPLIT variable can take are 10 %–90 % of the buffer cache length (boundary condition). In case both the above two conditions fails or increase/decrease of lengths of buffer cache parts causes violation of boundary condition, the COLD and HOT regions will not be resized.

Step (a) executes in $O(1)$ time. Step (b) is executed in $O(k)$ time while Step (c) is executed in $O(k \log k)$ time where k is size of buffer cache. Step (d) is an adaptive step which responds quickly to changes in page reference patterns (can take place in constant time). Adaptation occurs whenever the page fault corresponds to a page which is recently replaced. In case it is a working set page, the length of HOT region will be increased. In other case when it is a normal page, length of COLD region will be increased. Our algorithm assumes that in the cache most of the working set pages will be present in the HOT region and most of the normal pages will be present in the COLD region. Generally, working set pages will join the buffer cache in the HOT region, whereas normal pages will join the buffer cache in the COLD region. Afterwards, based on their relative reference patterns with respect to other cached pages, they will move towards the HOT end or COLD end of the buffer cache. Step (a1) can also happen in a constant time. The proposed algorithm is capable of providing performance for most of the common referencing patterns which is explained as follows.

Sequential References. In a sequential scan, pages are referenced and processed one after another. For non repeated scan, the probability of buffer goes on decreasing and finally buffer is edged out of the cache. In case of repeated scan the frequency of reference is increased and according to Equation (9) the probability is increased which will increase stay of page in the buffer cache. In the case of clustered sequential access (CS) like merge join, the buffer is frequently referred and according to Equation (9) its probability is increased.

Hierarchical References. This reference behavior is observed where index is repeatedly used, non leaf nodes of the index tree are referred frequently and their probability of stay is increased according to the Equation (9).

Infrequent Long Query. If the query is infrequent and requires lot of disk page access then to bring the required disk pages in the cache, whole existing buffers will be replaced. This will lead to removal of existing and stable working set in the cache and can result in lot of cache misses afterwards. The pages of such long infrequent queries will not appear in the working set due to their infrequent access pattern and they are inserted towards COLD END and subsequently they will be replaced quickly without disturbing the stable working set.

6 COSTS BENEFIT ANALYSIS

The above proposed algorithm (APR-ACF) enhances flash I/O performance by increasing the hit ratio and minimizing the write-erase operations, but the cost of replacement policy is very high because it involves updating probabilities and sorting list. The overall performance of the system can be analyzed by considering overheads in replacement strategy and improved hit ratio as explained in following section.

Suppose

- C: size of buffer cache,
- B: hit ratio,
- N: number of memory references made by replacement algorithm,
- M: time required for one memory access,
- D: time required for one disk access,
- B_0: minimum hit ratio assuming elementary replacement policy,
- B_1: maximum hit ratio gain by most complex algorithm.

Normally hit ratio is improved if more information is used to decide replacement buffer, but the improvement is not linear. Practically after reaching certain value there will be only marginal increase in hit ratio even if large information is scanned for deciding replacement buffer. Thus the gain in hit ratio gain is inversely proportional to the hit ratio hence the relationship can be approximated using following equation:

$$B = B_0 + B_1(1 - \exp(-\alpha_i nC)) \tag{10}$$

where $0 < B_0 + B_1 < 1$.

Assuming replacement decision is made by analyzing n bytes of information and it is repeated C times, i.e. for each buffer in the cache. The α_1 is proportionally constant which is approximately equal to improvement in hit ratio by scanning additional one information byte. Here exponential function is used because the hit ratio will increase less as proportion to increase in complexity of the replacement

algorithm, and after certain value it cannot be increased by the most complex replacement algorithm.

The cost of replacement policy depends on number of information bytes scanned in deciding replacement. This cost is linearly proportional to number of information bytes scanned.

Cost of replacement strategy = $(\alpha_0 + \alpha_2 nC)$ where α_0 is minimum bytes scanned for each reference and α_2 is proportionality constant.

If there are N references, then the total access time (T_0) with elementary replacement strategy is given by following equation:

$$T = N\left(M + (1 - B_0)D\right) + N\alpha_0 M = N\left(M + (1 - B_0)\gamma M\right) + N\alpha_0 M \tag{11}$$

where $D = \gamma M$.

According to current technology parameters, $\gamma \gg 10^6$.

When replacement is done using complex algorithm by scanning n bytes of information then the total access time (T) is

$$\begin{aligned} T_0 &= N\left(M + (1 - B)\gamma M\right) + NM\left(\alpha_0 + \alpha_2 nC\right) \\ &= N\left(M + (1 - B_0 - B_1 \exp(-\alpha_i nC))\gamma M\right) + MN\alpha_0 + N\alpha_2 nCM, \end{aligned} \tag{12}$$

$$T = T_0 - N\gamma B_1 \exp(-\alpha_i nC)M + N\alpha_2 nCM, \tag{13}$$

$$T = T_0 - T_- + T_+ \tag{14}$$

where

- T_- is reduction in time due to improved hit ratio due to complex replacement strategy,
- T_+ is increase in replacement time due to complex replacement strategy.

The overall reduction in time requires, $T_- \gg T_+$

$$\gamma B_1 \exp(-\alpha_1 nC) \gg \alpha_2 nC. \tag{15}$$

If γ is more than n, then the proposed algorithm always gives good performance, however, if buffer cache size is very large then n is also higher because information of more buffers is to be kept and cost of replacement policy tends to be very high making proposed algorithm less practically feasible. The cost can be reduced by avoiding updating and sorting list for each reference.

By using efficient data structures n can be reduced and inequality given in Equation (15) can be satisfied. To reduce cost of replacement we are proposing approximate algorithm which is having reduced replacement cost without much decreasing hit ratio.

7 APPROXIMATE ALGORITHM

To avoid modification of probabilities in step (b) and sorting of the list in step (c) of the above mentioned proposed algorithm, only historical count (HC) in the interval time of the current reference and frequency count (FC), which is the number of references in current time in that interval, are maintained. The total count (TC) is calculated as sum of historical count and frequency count which is not changed unless page is referred so the list will remain sorted and page which is referred is always inserted to maintain it in sorted order. The modified algorithm is:

Replacement (b_i, T_i)
If (b_i is not in buffer cache)
$b_i.FC = 0$;
$b_i.HC =$ (count from historical list if it exists in historical list, otherwise 0);

- scan the COLD REGION from the COLD END towards the WARM END for the first CLEAN page.
  - if found then replace first CLEAN page with b_i. (a2)
  - if not found then replace the COLD END page with b_i.
- Add replaced page to MRU end of LIST1. If LIST1 is already FULL, then delete the LRU END page from it, and add replaced page to MRU end. (a22)

$b_i.FC = b_i.FC + 1$; (b21)
$b_i.TC = b_i.TC + 1$; (b22)
$cnt++$;
Insert b_i in page_replacement_list in sort order; (c2)
If ($cnt >$ threshold) decrease FC and HC of each page; (c3)
resize() (d)
End

Steps a2, b21, b22 are executed in $O(1)$ times and step (c2) is executed in $O(\log k)$ times where k is size of buffer cache. If there is page in page_reference_list having the same total count then b_i is always inserted towards hot end in the sorted order.

If the reference pattern shows periodicity of references then historical count is increased and page is moved towards hot end. If the page is referred frequently then its current count is increased and it is moved towards hot end. If the page is referred by infrequent query then it is inserted towards cold end and it is finally moved out of buffer cache. If the page is referred frequently initially but its recent references are very less then it is moved towards cold end because of step (c3). In the COLD REGION likelihood of replacement will be high. The cost of replacement policy depends on step (c3) and can be reduced by increasing threshold. The algorithm finally guarantees longer stay to non-cold DIRTY pages and HOT CLEAN pages and a shortest possible stay to COLD CLEAN pages, followed by warm CLEAN pages and COLD dirty pages.

8 EXPERIMENTATIONS

For performance evaluation, we compare the best of our two algorithms, i.e., PR-ACF with the performance of five best competitors, namely CF-LRU, LRU-WSR, CCF-LRU, AD-LRU and PR-LRU. The performance measures used are buffer hit ratio, number of write operations and runtime.

8.1 Experimental Environment

The simulation experiments are conducted based on flash memory simulation framework, called FlashDBSim. FlashDBSim is a reusable and reconfigurable framework for the simulation-based evaluation of algorithms on flash disks [23]. FlashDBSim uses a modular design approach, which includes Virtual Flash Device Module (VFD), Memory Technology Device Module (MTD), and Flash Translation Layer Module (FTL). 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 upper-layer applications can access the virtual flash memory via block-level interfaces. The FTL module employs the EE-Greedy algorithm citeref14 in the garbage collection part and uses the threshold for wear-leveling proposed in [6]. In our experiment, we simulate a 128 MB NAND flash device with 64 pages per block and 2 KB per page. The I/O characteristics of the flash device are shown in Table 1 and the erasure limitation of blocks is 100 000 cycles.

Operation	**Access Time**	**Access Granularity**
Read	20 μs/page	Page (2 KB)
Write	200 μs/page	Page (2 KB)
Erase	1.5 ms/block	Block (128 KB = 64 pages)

Table 1. The characteristics of NAND flash memory

8.2 Dataset Characteristics

We have performed a trace-based simulation to evaluate the performance of the proposed PR-ACF algorithm in comparison with the competitor algorithms. We have done the experimentation on four different traces of 24×7 days which contain a mixture of random, sequential and repetitive patterns along-with different read and write localities. The first six days traces are used as a training dataset to define working sets for different time intervals and seventh days trace is used as a test

dataset for performance evaluation. The details of the traces are given in Tables 2 and 3.

Trace-ID	No. of Refs	No. of Pages	Read/Write Ratio	Locality
T1	250 000	12 000	80 %/20 %	60 %/40 %
T2	250 000	12 000	20 %/80 %	40 %/60 %
T3	250 000	12 000	50 %/50 %	80 %/20 %
T4	250 000	12 000	60 %/40 %	80 %/20 %

Table 2. Characteristics of four traces Set-A

A read/write ratio "$X\,\%/Y\,\%$" in Table 2 means that the read and write operations in the traces are of X and Y percentages, respectively. The locality expression in Table 2, e.g. $X\,\%/Y\,\%$, means that $X\,\%$ of total number of accesses call $Y\,\%$ of total number of data pages. Hence the likelihood of our working sets having members as the subset of these $Y\,\%$ data pages is very high.

Another workload that we have used for experimentation is OLTP one hour test trace in a real bank system containing 607 391-page references to a CODASYL database with a total size of 20 Gigabytes. The number of different pages accessed is 51 870 with each page having the size of 2 048 bytes. Ratio of read/write operations is 77 %/23 %. Table 3 gives the distribution of various types of references in each workload trace.

8.3 Performance Metrics

Three performance metrics, write count, hit ratio, and runtime were used in our simulation experiments to evaluate the results. The erase operations are not considered because the erase counts are nearly proportional to the write counts, as they are triggered due to call to write operations.

The read operations are not considered because firstly reads are covered in the hit ratio parameter and secondly they are less significant to overall performance due to its low latency compared to write operations.

Runtime parameter is highly influenced by hit ratio and the number of writes to the flash memory.

Trace-ID	Periodic Refs	Sequential Refs	Repetitive Refs	Random Refs
T1	7.22 %	14.43 %	60.26 %	18.09 %
T2	10.47 %	18.33 %	24.51 %	46.69 %
T3	19.32 %	21.62 %	29.58 %	29.48 %
T4	3.47 %	57.81 %	21.42 %	17.30 %

Table 3. Characteristics of four traces Set-B

8.4 Parameter Settings

For all the datasets, parameter w of the CFLRU algorithm is set to 0.5, which means half of the buffer is used as clean-first window. Parameter min_lc of AD-LRU is set to 0.2.

8.5 Results and Results Analysis

Figures 1–4 illustrate the comparison of the hit ratios on traces T1 to T4 for various buffer sizes.

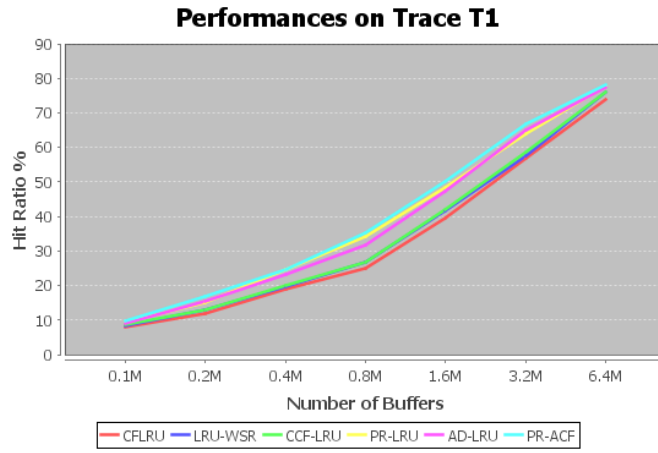

Figure 1. Hit ratio comparison on trace T1

On all the four traces, PR-ACF has a better hit ratio than the other algorithms, as shown in Figures 1–4. PR-ACF considers the locality of pages from recent as well as historical references. At each replacement, it selects the page from the COLD region, and within the COLD region, it prefers the COLDEST clean page for a replacement. In absence of CLEAN page in the COLD region, it replaces the COLDEST dirty page. Here, the COLD page corresponds to a page having a low probability of reference in the near future. Hence using this COLD first policy PR-ACF has achieved the best hit ratio in read-most as well as write-most scenarios, as the replaced page has a high probability of not getting referred shortly. On traces with a high percentage of random page references, PR-ACF manages to outperform all the competitor algorithms, because of its online adaptivity in which it continually controls the growth in the miss rate by resizing HOT and COLD regions, according to changes in the page access patterns. One of the important advantages of probabilistic cache is that pages in sequential reads and sequential writes are directly inserted in the COLD region hence quickly replaced, eliminating the possibility of cache pollution. Pages in random

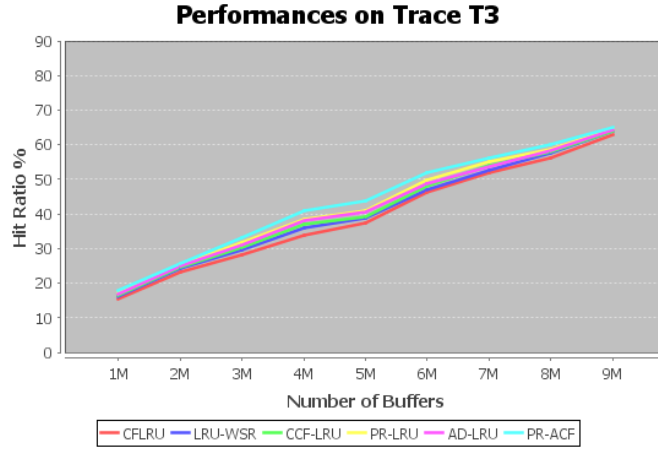

Figure 2. Hit ratio comparison on trace T2

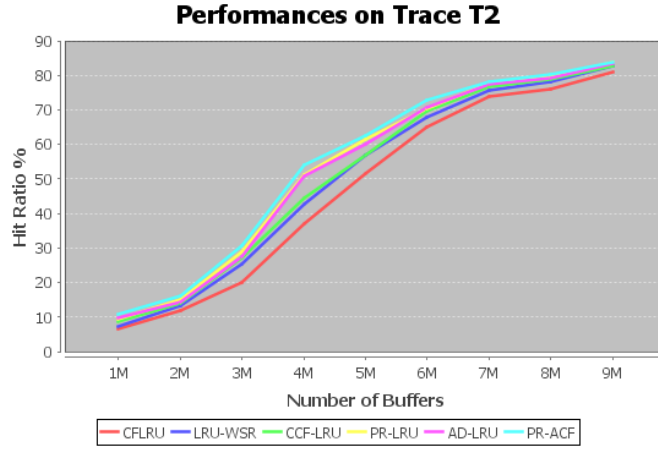

Figure 3. Hit ratio comparison on trace T3

reads and random writes also will have a short stay in the buffer cache, whereas pages in periodic and repetitive reads-writes will have longer stay in the buffer cache.

The other way around, many of the competitor algorithms evicts clean pages without considering their access frequencies, to protect dirty pages. Few of them protect HOT clean pages by selecting COLD dirty pages for replacement, but they consider only recent access patterns to predict the HOT and COLD pages. This adversely affects their hit ratio, what is the reason why our proposed algorithm outperforms them considerably. The increase in the hit ratio by PR-ACF when

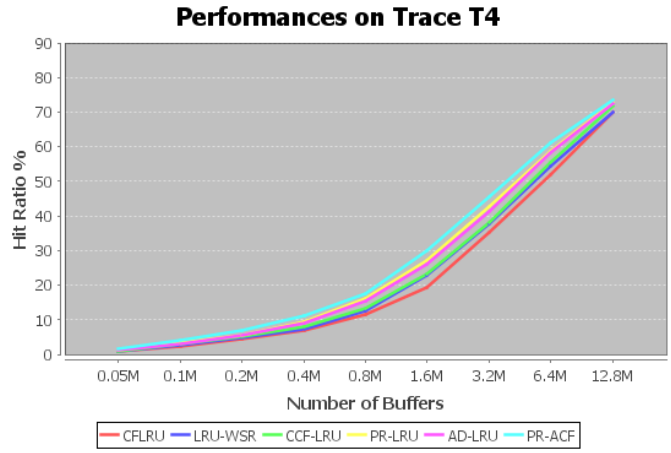

Figure 4. Hit ratio comparison on trace T4

compared with other best-performing algorithms CCF-LRU, AD-LRU and PR-LRU is 6.3 %, 4.5 %, and 2.7 %, respectively, on trace T3.

Figures 5–8 show the number of write operations for each algorithm on traces T1 to T4 for different buffer sizes. As shown in Figures 5–8, the write count of PR-ACF is less than all the competitor algorithms on all the traces. The primary reason for this is that all the HOT dirty pages (having close to largest write and erase cost) are saved from getting replaced, as all the HOT (having high probability of access at current instance of time) pages are in the HOT region, a region which is forbidden for replacement in PR-ACF. In the COLD region, PR-ACF favors evicting clean pages first from the buffer so that the number of writes incurring from replacements of COLD dirty pages can be reduced.

In absence of clean page in the COLD region, PR-ACF replaces the COLDEST dirty page, thereby protecting the HOT clean pages from replacement, and avoiding unnecessary degradation of the hit ratio. As COLDEST dirty page will be having the lowest probability of reference in the cache, there is less chance of it getting referred and causing an increase in the number of write counts in the current time interval. As the time interval changes so as the working set and the decisions of PR-ACF about which pages to protect from replacement also changes. Hence the above working principle benefits PR-ACF in all the three read-most, write-most and random-most scenarios in maintaining the low write count and high hit ratio simultaneously. The reduction in the number of writes by PR-ACF when compared with other best-performing algorithms CCF-LRU, AD-LRU and PR-LRU is 42.7 %, 29.6 %, and 24.2 %, respectively, on trace T2.

Figures 9–12 show the overall runtime of various replacement algorithms. The runtime of an algorithm is the sum of time required for read, write and erase operations plus the memory time. The access time for each type of operation is given

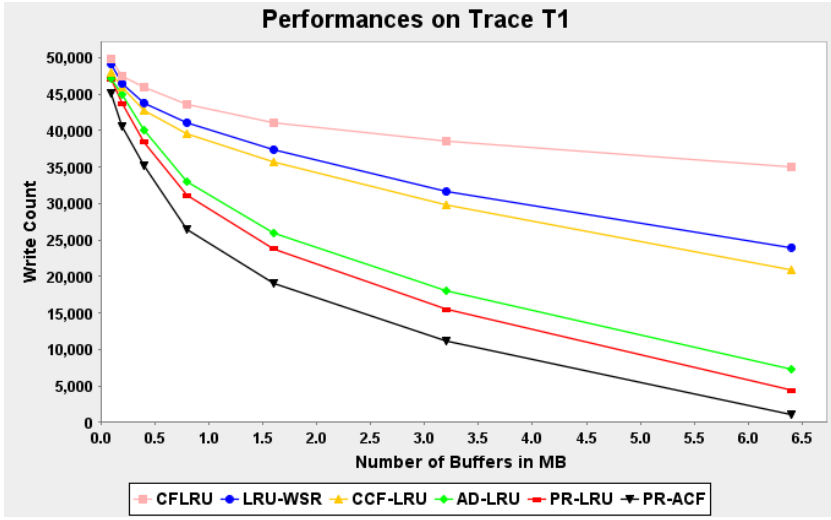

Figure 5. Write count comparison on trace T1

in Table 1. The total runtime of an algorithm can also be calculated as a number of read operations ∗ read access time + a number of write operation ∗ write access time + a number of erase operation ∗ erase access time. The total runtime of an algorithm is highly influenced by its hit ratio and the number of write operations (write count) involved.

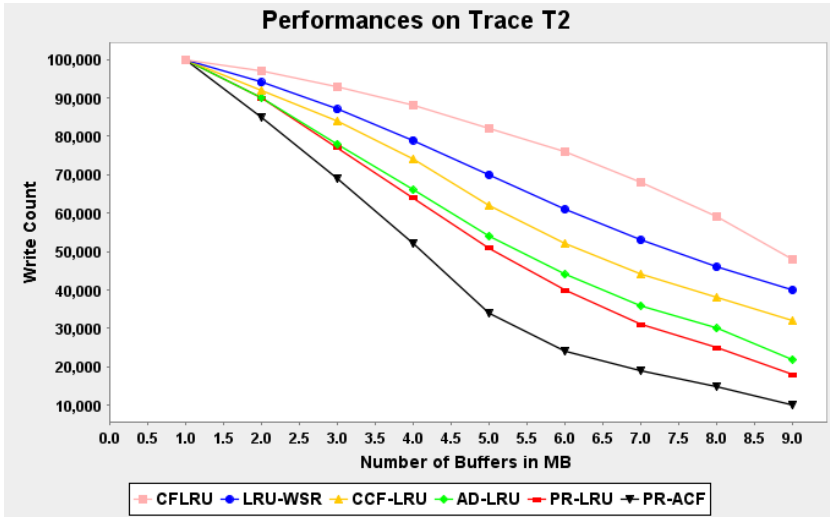

Figure 6. Write count comparison on trace T2

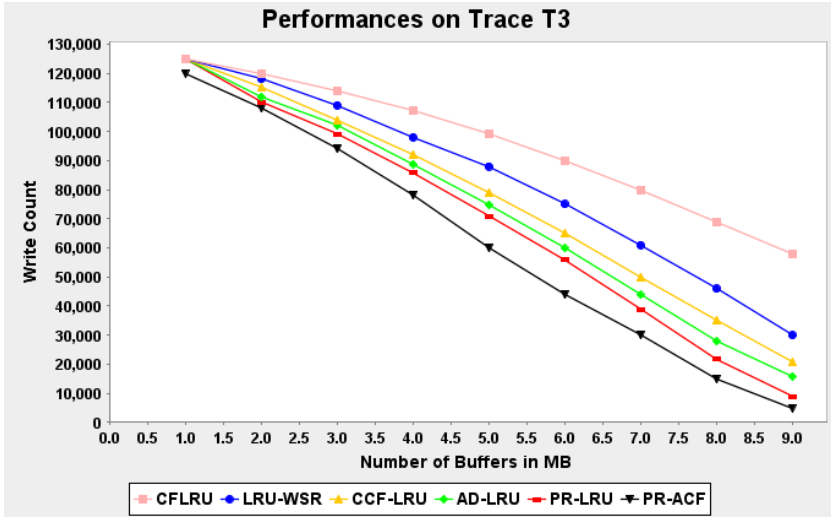

Figure 7. Write count comparison on trace T3

Specifically, total runtime is directly proportional to the write count and inversely proportional to the hit ratio. Figures 9–12 show that PR-ACF has the lowest runtime. This is because PR-ACF maintains the highest hit ratio and lowest write count amongst all the algorithms. The reduction in the runtime by PR-ACF when compared with other best-performing algorithms CCF-LRU, AD-LRU and PR-LRU is 30.4 %, 7 %, and 5.1 %, respectively, on trace T4.

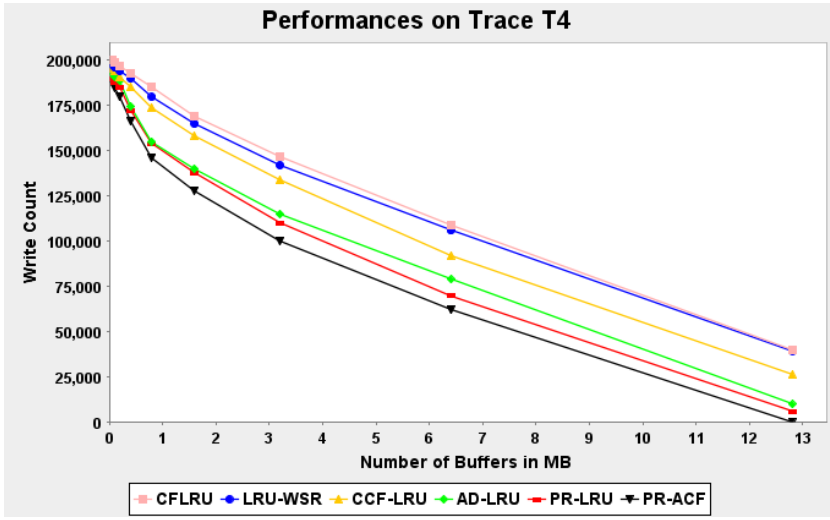

Figure 8. Write count comparison on trace T4

In most of the cases, total runtime is less influenced by running time of the algorithm (also called memory time) as compared to the I/O operational time (time taken by read, write and erase operations). However, with the weak locality, memory time has a greater impact on the runtime. With the increasing buffer cache size, the ratio of increase in memory time and decrease in total runtime keeps on increasing.

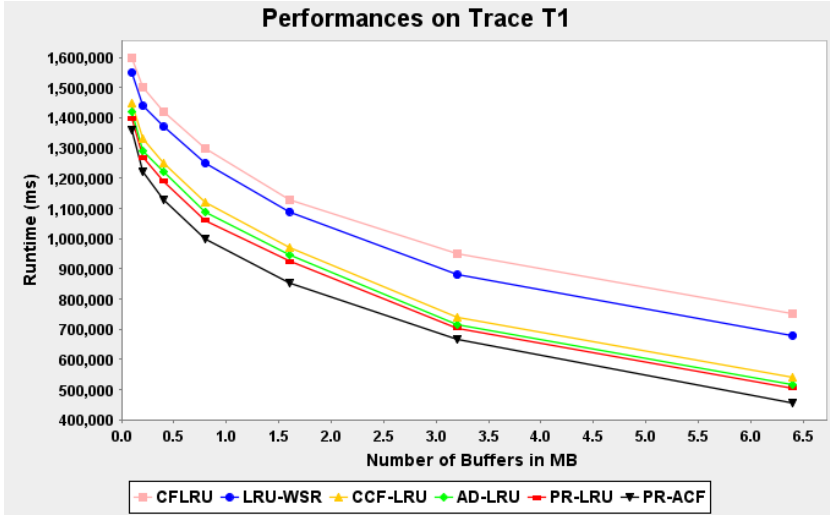

Figure 9. Runtime comparison on trace T1

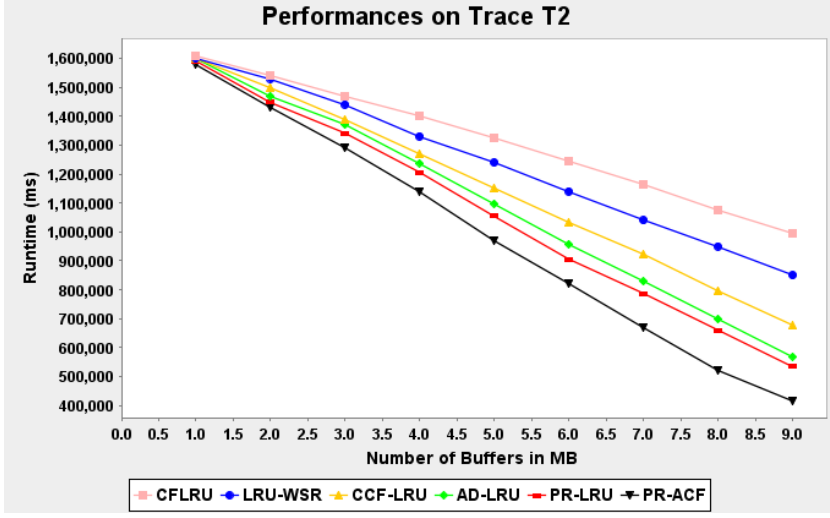

Figure 10. Runtime comparison on trace T2

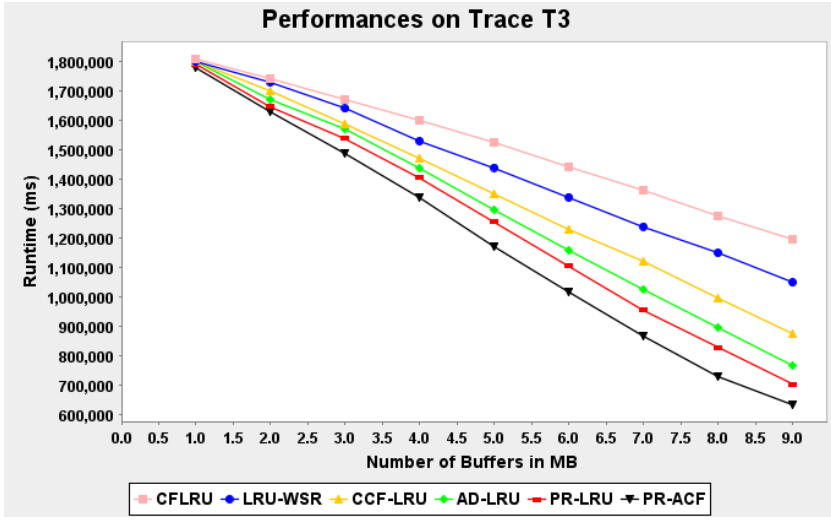

Figure 11. Runtime comparison on trace T3

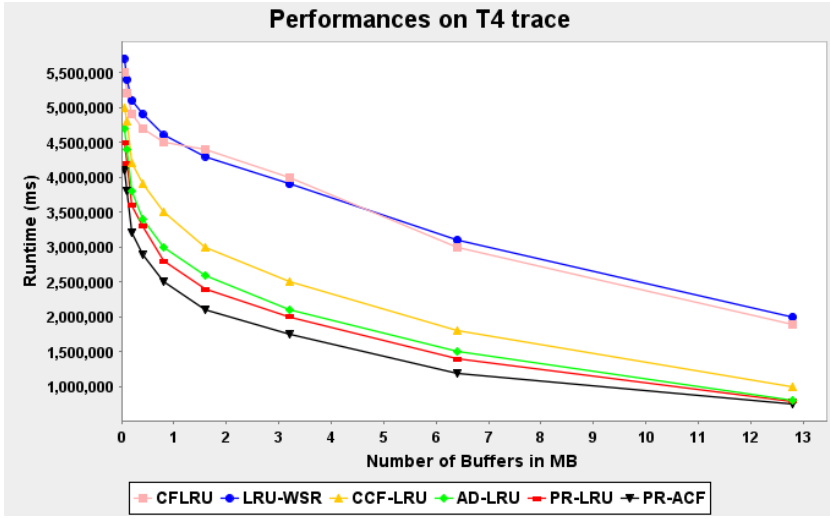

Figure 12. Runtime comparison on trace T4

Figures 13, 14, 15 show the hit ratio, write count and runtime comparison, respectively, of various replacement algorithms on real OTLP trace. PR-ACF has a clear advantage over other algorithms in terms of hit ratio, write count and runtime for most of the buffer cache sizes.

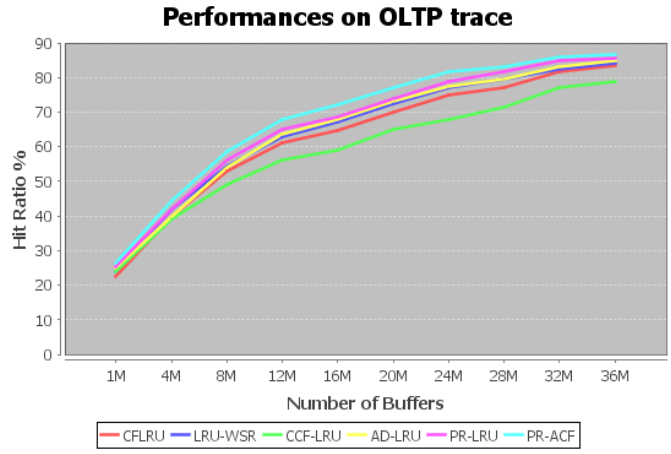

Figure 13. Hit ratio comparison on OLTP trace

9 OPTIMAL SET OF TIME INTERVALS

The above-proposed method uses static intervals like 9–10 am, 10–11 am and so on for calculating the predictive working set. This will give a simple and less computational algorithm for replacement but it may not give optimal predictive working set. Calculating correct intervals to get an optimized predictive working set and to get

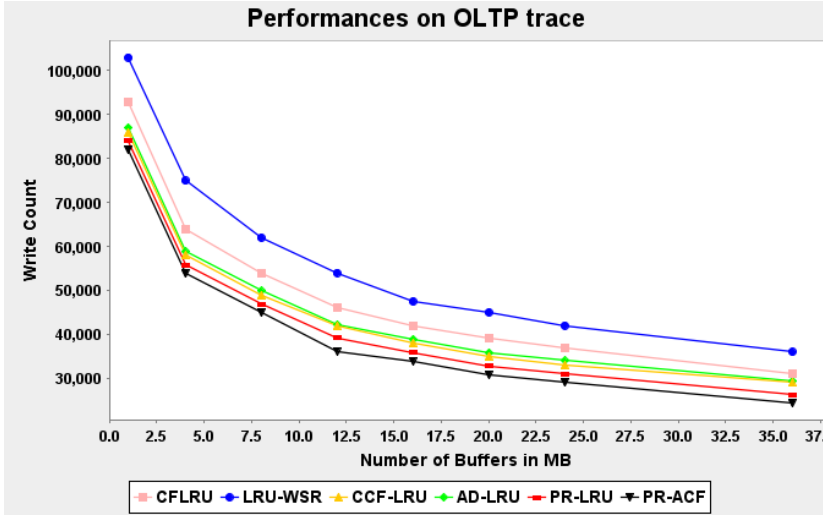

Figure 14. Write count comparison on OLTP trace

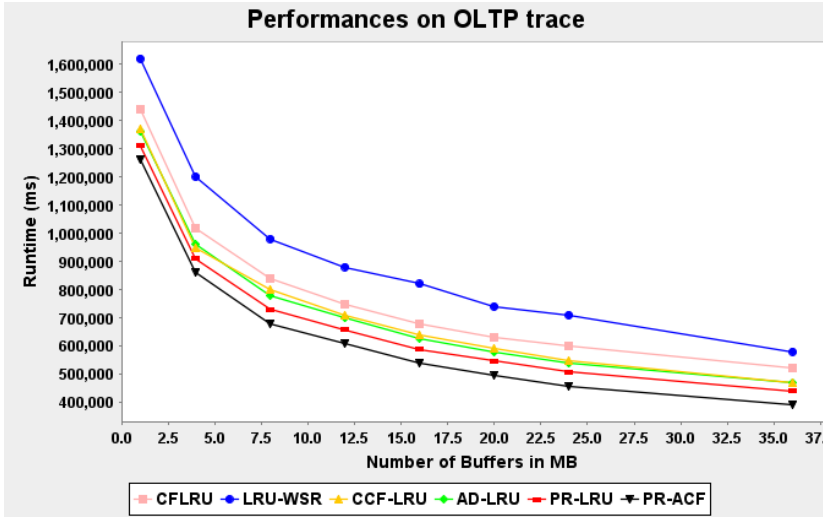

Figure 15. Runtime comparison on OLTP trace

minimum cache misses is a challenging task. Since it is a combinatorial optimization problem, techniques like genetic algorithms can be used.

9.1 Genetic Algorithms

Genetic Algorithms (GAs) are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. As such, they represent an intelligent exploitation of a random search used to solve optimization problems. After an initial population is randomly generated, the algorithm evolves through three operators:

- Selection, which equates to the survival of the fittest.
- Crossover, which represents reproduction by the crossover between solutions.
- Mutation, which introduces random modifications.

9.2 Proposed Genetic Algorithm for Finding Optimal Set of Time Intervals

For finding an optimal set of time intervals we have defined a system that can generate a solution under specified constraints. Initially, it reads values for the following three input parameters.

- Total number of time intervals in a solution (N).
- The minimum length of the time interval (minlength).

Algorithm 1 Generic Genetic Algorithm

```
randomly initialize population (p)
determine fitness of population (p)
while best individual is not good enough or number of evolutions does not reach its limiting value do
    select parents from population (p)
    perform crossover on parents creating population (p + 1)
    perform mutation of population (p + 1)
    determine fitness of population (p + 1)
end while
```

- The maximum length of the time interval (maxlength).

After receiving these 3 inputs, the system randomly creates an initial population of M solutions (P) in which each solution has exactly N time intervals and the length of each time interval is between minlength and maxlength. Each solution needs to cover a complete daytime of 24 hours represented by 1 to 1 440 (1 min to 1 440 min $(24 * 60)$). Where 1 represents 00:01 and 1 440 represents 00:00.

Example 1. Each solution of the sample solution set has the following characteristics: $N = 12$, minlength = 80, maxlength = 160. In the initial population of 100 solutions ($M = 100$) one of the solution is given in Table 4.

Interval-ID	Start-Time	End-Time	Fitness
1	1 065	1 148	0.56
2	1 148	1 283	0.64
3	1 283	1 419	0.64
4	1 419	66	0.58
5	66	178	0.66
6	178	305	0.6
7	305	415	0.72
8	415	538	0.68
9	538	672	0.63
10	672	826	0.7
11	826	959	0.66
12	959	1065	0.71
Solution: 21	Total Fitness		7.78

Table 4. Sample solution

The proposed algorithm works as follows: Within the minimum and maximum length constraints on the duration of the time interval, our algorithm randomly generates a population of M solutions each having N time intervals. M and N are predefined parameters that remain constant throughout the evolutionary process. Fitness of each solution is computed by taking the summation of the fitness of its

all the N time intervals. Fitness of each time interval is computed using the computeFitness function which takes the starttime and endtime of the time interval as input parameters and returns the summation of access probabilities of the top 15 % pages (working set pages) in that time interval. The information about pagewise reference probabilities for a given time interval is derived from the historical data by applying the probabilistic model discussed in Section 4.

An initial population P of M solutions is generated through the 3 steps of Algorithm 2.

9.2.1 Generating Initial Population of 100 Solutions

9.2.2 Finding Fairly Optimal Solution

The more the fitness of the solution, it can be expected that the higher is the overall daywise benefits to the cache performance from the N working sets of the solution. Here the N working sets of a solution are associated with N different intervals of time (of unequal duration) having different fitness values (For example refer to Table 4). To maximize the overall daywise benefits to the cache performance, i.e., to maximize the cache optimization, we search heuristically for the close to an optimal solution. For that the initial population of M solutions evolves through T iterations, using the selection, mutation and crossover operators (described in Section 9.3). After T iterations, in the final population of solutions, the solution with the highest fitness value is to be selected as close to an optimal solution (fairly optimal solution).

9.3 Genetic Operators

9.3.1 Selection

This operator picks the top 25 % solutions (in terms of fitness) from the initial/previous population for reproduction. The top 25 % solutions (in terms of fitness) from initial or previous population P are copied into the new population P'. The rest of the solutions in the new population P' are generated by crossover (50 % solutions) and mutation (25 % solutions) operators.

9.3.2 Crossover

This operator picks randomly any 2 solutions parent1 and parent2 from P and creates a new child solution having combined features of parent1 and parent2. The child or a new solution is included in the new population P'. We compute Avg1 and Avg2 as the average fitness value of parent1 and parent2 respectively.

$$\text{Avg1} = \text{Total Fitness of parent1}/N,$$

$$\text{Avg2} = \text{Total Fitness of parent2}/N.$$

The new – child solution will also have total N time intervals. The combined features (best of each of them) of parent1 and parent2 are copied into a new solution in the following way: The endtime of i^{th} time interval of parent1 is copied into the endtime of i^{th} time interval of a new solution if parent1 has the best solution for the interval I, i.e., interval I of parent1 satisfies the following conditions:

- Its fitness value > than Avg1 as well as Avg2.
- Its fitness value > fitness value of i^{th} time interval of parent2.

The starttime of i^{th} time interval of new solution is set to endtime of the previous – $(i-1)^{\text{th}}$ interval of the new solution or to the starttime of i^{th} time interval of parent1 if $i = 1$ (first interval).

A similar policy is employed for copying best fitness intervals of parent2 into a new solution. For the time interval I if neither parent1 nor parent2 has the best solution, we follow the following strategy: The starttime of i^{th} time interval of new solution is set to endtime of the previous – $(i-1)^{\text{th}}$ interval of the new solution or to the average of starttime of i^{th} time interval of parent1 and parent2 (for the first interval of a new solution).

The endtime of i^{th} time interval of new solution is set to the average of endtime of i^{th} time interval of parent1 and parent2.

In case of violation of interval length constraint in computing endtime in the above way, the endtime is recomputed using formula (16).

$$\text{endtime} = (\text{starttime} + \text{random}(\text{maxlength} - \text{minlength}) + \text{minlength})\,\%\text{high}. \quad (16)$$

9.3.3 Mutation

This operator selects the top 25 % solutions in terms of fitness from population P and adds them to empty set S. Each solution from set S is mutated randomly before adding it to a new population P'. The mutation is carried out in the following way.

From the solution find the top $N/4$ time intervals with the lowest fitness value. Mutate these intervals, i.e., regenerate the endtime of each of these intervals using formula (16). Recompute their fitness and add them to the mutated solution. If mutated intervals cause violation of interval length constraints for the subsequent intervals regenerate the endtimes of subsequent intervals using formula (16), up to the interval for which violation stops.

9.3.4 Experimental Results

We have taken a seven days OLTP traces from the commercial MIS database server. These traces are timestamped and have the page size of 2 KB. After analyzing, the set of intervals reported by the proposed genetic algorithm (Number of Iterations = 100, Population Size = 100, Number of Intervals = 11) is as follows:

31–158–303–425–520–695–829–937–1 099–1 254–1 382–31.

Algorithm 2 Generating initial population of 100 solutions

1: **Step 1:** ▷ Parameters which are constant for all the solutions in the population are initialized.
2: $M = 100$ ▷ No of solutions in the initial population
3: Low = 1 ▷ represents 00:01 hrs
4: High = 1 440 ▷ represents 00:00 hrs
5: Read N ▷ number of time intervals in each solution
6: Read minlength ▷ minimum length of each time interval
7: Read maxlength ▷ maximum length of each time interval
8: **[end of Step 1]**
9: **Step 2:** ▷ create initial population P
10: **for** $I = 1$ to M repeat Step 3.
11: **[end of Step 2]**
12: **Step 3:** ▷ create I^{th} solution in P
13: Solution (I).Interval (1).Starttime = Low + random(High − Low); ▷ Set starttime of first interval randomly by value between the range Low to High.
14: Solution (I).Interval (1).Endtime = (Solution (I).Interval (1).Starttime + random (maxlength − minlength) + minlength) % High; ▷ Set the endtime of first interval by adding a random value between minlength and maxlength to the starttime of first interval.
15: Solution (I).Interval (1).Fitness = computeFitness (Solution (I).Interval (1).Starttime, Solution (I).Interval (1).Endtime); ▷ Compute Fitness of first time interval of Solution I
16: **for** $J = 2$ to N repeat Step 4. ▷ Repeat the process for remaining $N - 1$ time intervals of Solution I, all the time intervals will have starttime equal to endtime of the previous time interval.
17: **[end of Step 3]**
18: **Step 4:**
19: Solution (I).Interval (J).Starttime = Solution (I).Interval $(J - 1)$.Endtime;
20: Solution (I).Interval (J).Endtime = (Solution (I).Interval (J).Starttime + random (maxlength − minlength) + minlength) % High;
21: Solution (I).Interval (J).Fitness = computeFitness (Solution (I).Interval (J).Starttime, Solution (I).Interval (J).Endtime); ▷ Compute Fitness of J^{th} time interval of Solution I
22: **[end of Step 4]**
23: **Step :** ▷ Compute Fitness of a solution Solution(I).Fitness = $\sum_{j=1}^{N}$ [Solution(I).Interval(J).Fitness]
24: **[end of Step 5]**

We defined working sets for the above optimal set of time intervals as well as working sets for the static time intervals of one-hour fixed length on the above dataset.

Here a new working set (corresponding to new time interval) is loaded automatically at the expiration of each time interval (which coincides with the start of the next time interval).

Experimental results prove that the use of optimal time intervals in PR-ACF maximizes the overall day-wise performance gains considerably in comparison to the results achieved with static time intervals (one hour each), as shown in Figure 16.

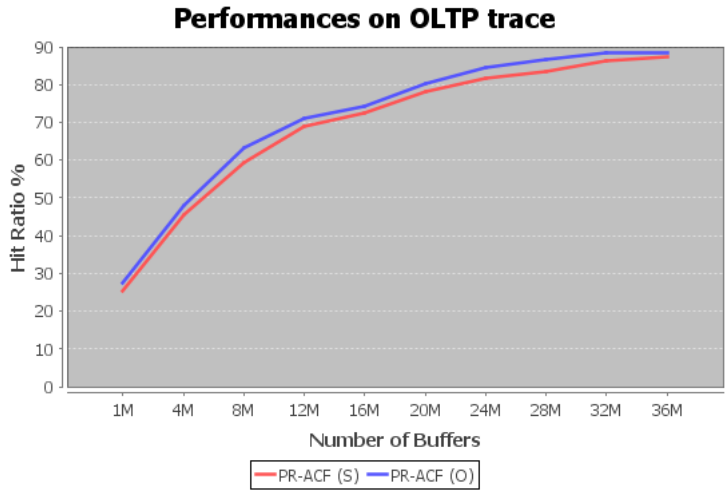

Figure 16. Performance comparison of PR-ACF algorithm with static (S) and optimal (O) time intervals

10 CONCLUSION

In this paper, we focus on a cache replacement policy for database systems equipped with flash memory as a secondary storage. We propose a new replacement policy, called PR-ACF, which considers the imbalance of read and write costs of flash memory while replacing pages. The basic idea behind PR-ACF is to avoid the replacement of dirty pages present in the buffer cache to minimize the number of write operations and at the same time preventing the significant degradation in the hit ratio to achieve the fairly close to overall optimal performance. To determine the coldness and hotness of the cached pages we propose a probabilistic model that calculates the probability of future reference of each cached page and organizes the cache based on computed probability. To improve the accuracy of prediction, the probability is calculated based on the study of reference patterns from a history of references along with recent reference patterns. The buffer cache is divided into

HOT and COLD regions which are dynamically resized according to the changing access patterns. The page replacement always happens in the COLD region, and within the COLD pages, the COLDEST clean page is targeted for replacement, thus deliberately keeping the COLD dirty pages in the cache to avoid performance degradation due to costly write and erase operations on flash memory. The proposed PR-ACF algorithm was tested on the flash simulation platform Flash-DBSim, by comparing its performance with best-known flash-based replacement algorithms. The experimental results show that our proposed algorithm performs better than the top-performing algorithms like CCF-LRU, AD-LRU, and PR-LRU with respect to write count, runtime as well as hit ratio. Experimental results also prove that the use of optimal time intervals maximizes the overall day-wise performance gains considerably in comparison to the results achieved with static time intervals.

REFERENCES

[1] Robinson, J. T.—Devarakonda, M. V.: Data Cache Management Using Frequency-Based Replacement. ACM SIGMETRICS Performance Evaluation Review, Vol. 18, 1990, No. 1, pp. 134–142, doi: 10.1145/98460.98523.

[2] Chou, H.-T.—DeWitt, D. J.: An Evaluation of Buffer Management Strategies for Relational Database Systems. Algorithmica, Vol. 1, 1986, No. 1-4, pp. 311–336, doi: 10.1007/BF01840450.

[3] Sacco, G. M.—Schkolnick, M.: Buffer Management in Relational Database Systems. ACM Transactions on Database Systems, Vol. 11, 1986, No. 4, pp. 473–498, doi: 10.1145/7239.7336.

[4] Pan, Z.-W.—Xiang, D.-H.—Xiao, Q.-W.—Zhou, D.-X.: Parzen Windows for Multi-Class Classification. Journal of Complexity, Vol. 24, 2008, No. 5-6, pp. 606–618, doi: 10.1016/j.jco.2008.07.001.

[5] Shen, H.—Yan, X.-L.: Probability Density Estimation over Evolving Data Streams Using Tilted Parzen Window. 2008 IEEE Symposium on Computers and Communications, 2008, pp. 585–589, doi: 10.1109/ISCC.2008.4625751.

[6] Lofgren, K. M. J.—Norman, R. D.—Thelin, G. B.—Gupta, A.: Wear Leveling Techniques for Flash EEPROM Systems. United States Patent US6850443B2, 2005.

[7] Park, S.-Y.—Jung, D.—Kang, J.—Kim, J.—Lee, J.: CFLRU: A Replacement Algorithm for Flash Memory. Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06), ACM, 2006, pp. 234–241, doi: 10.1145/1176760.1176789.

[8] Jin, P.—Härder, T.—Li, Z.: AD-LRU: An Efficient Buffer Replacement Algorithm for Flash-Based Databases. Data and Knowledge Engineering, Vol. 72, 2012, pp. 83–102, doi: 10.1016/j.datak.2011.09.007.

[9] Yuan, Y.—Shen, Y.—Li, W.—Yu, D.—Yan, L.—Wang, Y.: PR-LRU: A Novel Buffer Replacement Algorithm Based on the Probability of Reference for

Flash Memory. IEEE Access, Vol. 5, 2017, pp. 12626–12634, doi: 10.1109/ACCESS.2017.2723758.

[10] Zhou, W.—Feng, D.—Hua, Y.—Liu, J.—Huang, F.—Chen, Y.—Zhang, S.: Prober: Exploiting Sequential Characteristics in Buffer for Improving SSDs Write Performance. Frontiers of Computer Science, Vol. 10, 2016, No. 5, pp. 951–964, doi: 10.1007/s11704-016-5286-z.

[11] Chen, F.—Koufaty, D. A.—Zhang, X.: Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. Proceedings of the International Conference on Supercomputing (ICS '11), ACM, 2011, pp. 22–32, doi: 10.1145/1995896.1995902.

[12] On, S. T.—Gao, S.—He, B.—Wu, M.—Luo, Q.—Xu, J.: FD-Buffer: A Cost-Based Adaptive Buffer Replacement Algorithm for Flash Memory Devices. IEEE Transactions on Computers, Vol. 63, 2014, No. 9, pp. 2288–2301, doi: 10.1109/TC.2013.52.

[13] Li, Z.—Jin, P.—Su, X.—Cui, K.—Yue, L.: CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory. IEEE Transactions on Consumer Electronics, Vol. 55, 2009, No. 3, pp. 1351–1359, doi: 10.1109/TCE.2009.5277999.

[14] Kwon, O.—Lee, J.—Koh, K.: EF-Greedy: A Novel Garbage Collection Policy for Flash Memory Based Embedded Systems. In: Shi, Y., van Albada, G. D., Dongarra, J., Sloot, P. M. A. (Eds.): Computational Science – ICCS '07. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4490, 2007, pp. 913–920, doi: 10.1007/978-3-540-72590-9_138.

[15] Consuegra, M. E.—Martinez, W. A.—Narasimhan, G.—Rangaswami, R.—Shao, L.—Vietri, G.: Analyzing Adaptive Cache Replacement Strategies. arXiv preprint arXiv:1503.07624, 2015, 26 pp.

[16] Jung, H.—Shim, H.—Park, S.—Kang, S.—Cha, J.: LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory. IEEE Transactions on Consumer Electronics, Vol. 54, 2008, No. 3, pp. 1215–1223, doi: 10.1109/TCE.2008.4637609.

[17] Corbató, F. J.: A Paging Experiment with the Multics System. In Honor of Philip M. Morse, MIT Press, Cambridge, Mass., 1969, pp. 217–228.

[18] O'Neil, E. J.—O'Neil, P. E.—Weikum, G.: The LRU-K Page Replacement Algorithm for Database Disk Buffering. ACM SIGMOD Record, Vol. 22, 1993, No. 2, pp. 297–306, doi: 10.1145/170036.170081.

[19] Lee, D.—Choi, J.—Kim, J.-H.—Noh, S. H.—Min, S. L.—Cho, Y.—Kim, C. S.: LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Transactions on Computers, Vol. 50, 2001, No. 12, pp. 1352–1361, doi: 10.1109/TC.2001.970573.

[20] Jiang, S.—Zhang, X.: LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. ACM SIGMETRICS Performance Evaluation Review, Vol. 30, 2002, No. 1, pp. 31–42, doi: 10.1145/511399.511340.

[21] Johnson, T.—Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94), 1994, pp. 439–450.

[22] Bansal, S.—Modha, D. S.: CAR: Clock with Adaptive Replacement. Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04), 2004, pp. 187–200.

[23] Su, X.—Jin, P.—Xiang, X.—Cui, K.—Yue, L.: Flash-DBSim: A Simulation Tool for Evaluating Flash-Based Database Algorithms. 2009 2nd IEEE International Conference on Computer Science and Information Technology (ICCSIT '09), Beijing, China, 2009, doi: 10.1109/ICCSIT.2009.5234967.

[24] Zhou, Y.—Chen, Z.—Li, K.: Second-Level Buffer Cache Management. IEEE Transactions on Parallel and Distributed Systems, Vol. 15, 2004, No. 6, pp. 505–519, doi: 10.1109/TPDS.2004.13.

[25] Jung, H.—Yoon, K.—Shim, H.—Park, S.—Kang, S.—Cha, J.: LIRS-WSR: Integration of LIRS and Writes Sequence Reordering for Flash Memory. In: Gervasi, O., Gavrilova, M. L. (Eds.): Computational Science and Its Applications (ICCSA 2007). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4705, 2007, pp. 224–237, doi: 10.1007/978-3-540-74472-6_18.

[26] Hayes, B.: Cloud Computing. Communications of the ACM, Vol. 51, 2008, No. 7, pp. 9–11, doi: 10.1145/1364782.1364786.

[27] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. White Paper, `http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf`, 1998.

Atul O. Thakare received his post-graduate degree (M.Eng. (CSE)) from SGB Amravati University, Maharashtra, India and his under-graduate degree (B.Eng. (CT)) from RTM Nagpur University, Maharashtra India. Currently, he is pursuing his Ph.D. from VNIT, Nagpur, Maharashtra, India. He has a total of 16 years of experience, six years in the IT industry and ten years in the academic profession.

Parag S. Deshpande has completed his Ph.D. from Nagpur University, Nagpur, India and his M-Tech from IIT Powai, Mumbai, India. He is currently working as Professor in the Department of Computer Science and Engineering, VNIT, Nagpur, Maharashtra, India. He has 31 years of academic experience. He is the author of several books including C and Data Structure, Data Warehousing Using Oracle, SQL/PLSQL for Oracle 11g. He is member of ISTE and SAE-India.