IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 24, 2019, accepted April 9, 2019, date of current version May 3, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2912751

A Novel Adaptive Database Cache Optimization Algorithm Based on Predictive Working Sets in Cloud Environment

ATUL O. THAKARE AND PARAG S. DESHPANDE
Department of Computer Science and Engineering, Visvesvaraya National Institute of Technology, Nagpur, India
Corresponding author: Atul O. Thakare (aothakare@gmail.com)

ABSTRACT Many applications are hosted on cloud databases where several applications share the same database instance. Such database management system exhibits periodic behavior in terms of data references. For example, U.S. 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 block references. This periodicity of block references can be used to improve cache performance by improving the block replacement policy. This paper proposes a novel cache replacement policy by considering periodicity of references of database blocks. It also proposes how to estimate the probability of a block reference in a specified time interval using ''Parzen Windows,'' and determines a working set based on this probability and defines the cache management policy using this working set. The experimentation shows considerable improvement of the hit ratio as the performance measure of the buffer cache management as compared with the other state-of-art algorithms.

INDEX TERMS Cloud databases, databases and data warehouses, data mining, intelligent information systems, decision support systems.

I. INTRODUCTION

Generally, database management systems efficiently reduce the number of disk accesses by using a buffer pool. The efficiency of the buffer management policies, usually measured by hit ratio is key to the performance of database applications [1]. Poor hit ratio results in serious performance degradation even if the database queries are optimized. High buffer hit ratio can be achieved, by avoiding the replacement of cached blocks having a high probability of access in the near future. Thus for high performance, blocks having lower probability of getting referred in near future needs to be evicted. Only those blocks with relatively high access probability deserve to stay in the cache for a longer time. High probability blocks or hot blocks are often called as the working set blocks. Most of the existing buffer management policies under-perform due to their inability to predict future access patterns [1], because of lack of information about the past access patterns.

Most of the existing buffer management policies use recent references of the blocks present in the cache, which may not be sufficient to predict future access patterns. In this regard, we propose to capture the periodicity in block access patterns and use it as an additional feature in improving cache performance. Periodicity means block access patterns which are getting repeated in a particular period of time (a particular time interval). The information about the time period-wise hot blocks can be used in optimizing cache performance.

To achieve this, we propose to use recent week's block access history and obtain the working sets from it corresponding to different time intervals. These working sets should be referred along with the recent access patterns to make block replacement decisions. The 24 hours daytime is statically partitioned to obtain 24-time intervals each of 60 minutes duration, and a working set is computed for each one of them. The time interval can be set arbitrary but it will make algorithm harder than NP-hard problem because of addition of more degree of freedom and infinite search space; because time interval belongs to domain of real numbers. The best way to tackle complexity and increase accuracy is to reduce time interval but it will add additional replacement overheads because a historical working set will be loaded more frequently. To keep the overheads to the minimum we decided to keep all the intervals of fixed length.

The associate editor coordinating the review of this manuscript and approving it for publication was Arif Ur Rahman.

A Working set for a particular interval of time is prepared by examining the block reference patterns in that interval from the last seven days history. For example, a working set for time interval 10.00 am −11.00 am is defined by analyzing the block references between 10.00 am −11.00 am of the last seven days. Defining the working sets needs computing access probabilities of a block in a particular time interval. We have used the 'Parzen Window' density estimation to find historical access probabilities of different blocks within different time intervals [15]. For each time interval, we rank the blocks based on their probabilities and the top K blocks based on ranks will find the place in its working set.

In order to keep the working sets synchronized with the changes in the block access patterns, periodically they need to be recomputed. At each re-computation, last 7 days historical data will be analyzed. As this task of refreshing the working sets is computationally intensive, it needs to be performed off-line once every week. This procedure of mining historical data is totally offline procedure and can be done on some other machine. Hence it will not have any impact during execution of replacement algorithm. Refreshing the working sets (by exporting the recently computed to the online database system) will tune them with the latest historical patterns and is termed as the off-line adaptability of the buffer management scheme. Each Database Block joining the cache can be classified into two types:

1) The block which belongs to the working set of the current time interval (hot blocks).
2) The block which does not belongs to the working set of the current time interval (called as normal blocks).

Working set theory adapted here, prolongs the stay of hot blocks once cached so that they should not be replaced as long as they exhibit access frequency around their historical access patterns. To give some additional stay to hot blocks, the working set principle enforces quick replacement of the normal blocks. But it needs to adopt a different policy for handling random blocks (outliers). Outliers can be described as the blocks whose current access patterns are reversed or not consistent with their historical access patterns. The buffer management scheme has to change its behavior (adapt), for handling these outliers and this is termed as on-line adaptability of the buffer management scheme. Here outliers can be of two types. They are

1) Hot blocks whose current access pattern is not frequent or less frequent as compared to most of the other cached blocks.
2) Normal blocks whose current access pattern is frequent or more frequent as compared to most of the other cached blocks.

In cloud databases, multiple clients across various regions share same database instance resulting in periodic behavior of references. The proposed methods provides replacement policy considering periodicity of references so it is more suitable for cloud databases [31]–[33].

The rest of the paper is organized as follows; section 2 describes the commonality amongst different well-known cache algorithms along with their distinctive features, followed by section 3 which describes the architecture of proposed algorithms. Section 4 describes working of proposed algorithms, its advantages over the existing algorithms and their efficiency in terms of time complexity. Section 5 describes the experimentation and results and section 6 has the conclusion and future work.

II. RELATED WORK

In conventional database systems, the buffer management policy has evolved over the years. The recency and/or frequency based policies like LRU, LFU, Clock etc. works well with the suitable workloads but all of them suffer from performance degradation when tested on non-friendly workloads. Following is the listing of merits and demerits that an existing buffer management policies have:

A. RECENCY BASED POLICIES

LRU work well with many applications but it has serious limitations when tested on non-suitable workloads. Moreover, it is expensive, as it requires that the recency sequence be re-ordered at each access. In addition, it is not scan-resistant and has high lock contention. Workloads in which locality of time accesses exhibits many different patterns for different localities, LRU performs poorly as it is static in approach of giving importance to recently accessed blocks over less recently accessed blocks [5]. Another of its limitation is it lacks the ability to differentiate between blocks that have frequent references and blocks that have infrequent references [3]. The most common complaint with LRU algorithm is that, a burst of references to infrequently used blocks, such as sequential scans through large files, may cause the replacement of frequently referenced blocks in cache.

CLOCK and DUELING CLOCK is a close approximation of LRU but unlike LRU both are scan resistant. Still, they suffer from other drawbacks of LRU [11].

Mid-point

insertion policy of touch count algorithm enables it to perform better than LRU on LRU non-friendly workloads.

B. FREQUENCY OR COUNTING BASED POLICIES

LFU (Least Frequently Used) takes a decision based on frequency parameter but, situations in which some blocks are accessed extremely frequently in a certain period and never requested again will lead to cache pollution with LFU policy. LFU has no means to discriminate recent versus past reference frequency of a block and is therefore unable to cope with evolving access patterns. Some examples of LFU like algorithms are FBR (frequency-based replacement algorithm), MQ (Multi-Queue) and 2Q. Robinson and Devarakonda proposed a frequency-based replacement algorithm (FBR) which maintains reference counts to ''factor out'' locality [2]. Zhou *et al.* proposed Multi-Queue (MQ) algorithm, which sets up multiple queues and uses access frequencies to

determine which queue a block should be in [34]. Like LIRS [7], 2Q also uses two linear data structures following a principle that only re-referenced blocks deserve to be in cache for a longer time. Similar to LIRS, 2Q identify blocks of small reuse distance and hold them in a cache [35].

C. RECENCY AND FREQUENCY BASED POLICIES

Unlike LRU policy which considers only the recency information while evicting pages, LRU-K also considers the frequency information by evicting blocks with the largest backward K-distance. In essence, the LRU-K algorithm tries to approximate Least Frequently Used (LFU) cache replacement algorithm in an efficient way [3].

Lee *et al.* proposed the LRFU policy which associates a value called the CRF (Combined Recency and Frequency) with each block. Using CRF it quantifies the likelihood that the block will be referenced in the near future. CRF is computed by using a weighing function. The weighing function essentially reflects the influence of the recency and frequency factors of a block's past references in projecting the likelihood of its rereference in the future [6]. However, LRFU is not effective on workloads with a looping pattern because the block reference frequencies in looping references are hard to distinguish [6].

CLOCK-Pro boost CLOCK by adding to it all the performance advantages of LIRS. Without any pre-determined parameters, CLOCK-Pro protects its performance against the changing access patterns to give high performance on a broad spectrum of workloads [22].

D. RECENCY AND FREQUENCY BASED ADAPTIVE POLICIES

There are few adaptive algorithms which refer to some of the historical information related to the recently evicted blocks, which is useful in finding the recent trends and the reversal in those trends [18], [23].

Unlike LRU, LIRS (Low Inter-reference Recency Set) uses reuse distance rather than recency for making block replacement decisions. In LIRS, a page with a large reuse distance will be replaced even if it has a small recency [7], [10]. For example, a recently accessed one time used blocks will be replaced quickly because its reuse distance is infinite, even though its recency is very small.

With self-tuning parameter p which decides the division and sizes of LRU lists T1 and T2, B1 and B2, algorithms like CAR, CART, ARC adapts themselves effectively according to changes in the block access patterns.

In response to evolving and changing access patterns, Adaptive Replacement Cache (ARC) dynamically, adaptively, and continually balances between the recency and frequency components in an online and self-tuning fashion [8].

Clock with Adaptive Replacement (CAR), maintain two clocks, say, T1 and T2, where T1 contains pages with ''recency'' or ''short-term utility'' and T2 contains pages with ''frequency'' or ''long-term utility'' [9]. New pages are first inserted in T1 and promote to T2 upon qualifying test of long-term utility. By using a certain precise history mechanism that remembers recently evicted pages from T1 and T2; CAR adaptively determines the sizes of these lists in a data-driven fashion. CAR combines the best features of CLOCK and ARC by removing all the disadvantages of LRU [17], [18].

Panda *et al.* [24] presents a survey of classification of replacement strategies in associative mapping schemes with detailed discussion on their advantages and disadvantages. Gupta *et al.* [29] attempts to improve the effectiveness of previous cache replacement policies by using sequential pattern mining and clustering to predict accurately next location of the mobile users.

Cherkasova [4] proposed a Greedy-Dual-Size-Frequency caching policy which incorporates the attributes of the file and its access such as file size, file access frequency, access recency of the last access etc to maximize byte hit ratios for WWW proxies. Ma *et al.* [30] proposes an improved Greedy Dual Size Frequency (GDSF) algorithm which adds weighted frequency-based time and weighted document type to GDSF algorithm [4] in order to improve hit ratio. He *et al.* [12] presents clustering algorithm which places the objects accessed near to each other in time into the same page. Chiang *et al.* [13] proposes a periodic cache replacement policy for dynamic web content at application server.

E. FLASH SSD BASED BUFFER REPLACEMENT POLICIES

Flash Memory based Solid State Drive (SSD) is an emerging storage technology based on semiconductor chips & plays a crucial role in revolutionizing the storage system design. Currently, SSDs has been widely used for mobile devices, embedded computing systems and portable devices such as PDAs (personal digital assistants), HPCs (handheld PCs), PMPs (portable multimedia players) etc. Recently due to continuous decrease in price and increase in capacity (which gets double every year), SSDs are also considered for replacing magnetic hard disks in enterprise database servers. Few of the buffer management algorithms for flash-oriented systems which are largely influenced by the different I/O characteristics of the flash disks are as follows:

Yang *et al.* [25] present a new buffering scheme for tree indexes on SSDs which is based on the observation that access patterns on index pages are much different from those on data pages in tree index, e.g., B+ trees. It assigns priorities to index pages and uses priority and recency to detect the hotness of index pages. Authors in [19] proposed a novel buffer replacement algorithm named FOR, which stands for Flash-based Operation-aware buffer Replacement. It propose operation-aware page weight determination for the buffer replacement. The weight metric not only measures the locality of read/write operations on a page, but also takes the cost difference of read/write operations into account. Park *et al.* [16] presents a new page replacement algorithm for NAND flash memory based embedded systems that considers asymmetric operation cost of each page.

Adaptive Double LRU (AD-LRU) separates the buffer pool into a cold LRU queue and a hot LRU queue, based on reference frequencies [14], [20]. The sizes of the two queues are adjusted according to the access pattern. PR-LRU (Probability of Reference LRU) uses a reference probability to predict the possibility of a page of getting referenced in the near future. A page's reference probability is calculated using three variables, namely the reference times, the number of reference pages from the last to the penultimate references, and the number of reference pages from the first to the last reference [28].

In this paper, we are aiming at the improvement in the disk-based database buffer management. In general, in the existing algorithms we found one thing in common that they do not take into consideration the patterns that the blocks are exhibiting or the workloads are repeating over a particular period of the daytime. Instead, they work purely based on the recent trends derived from recent past reference history mostly available in the cache.

Here novelty of our approach can be stated as follows:

By extracting frequent access patterns for different time intervals from block access history of the recent week, database system learns the periodic patterns in block references and defines the working sets. Each working set corresponds to a particular time interval having the start time and end time. Each working set will be used between the start time and end time of its time interval, to optimize the database performance in query processing. Using the working sets, our buffer management policy aims to distinguish between the working set blocks and normal blocks. This classification is purely based on historical patterns. In addition, it integrates the consideration of recent access patterns of the cached blocks in making replacement decisions. It also has off-line adaptability to cope up with the changing patterns and on-line adaptability to handle the reversals in memorized patterns.

To overcome most of the drawbacks of existing policies, and in order to satisfy most of the aims of the above-mentioned approach, we define the following solution characteristics of the buffer management scheme.

1) The buffer management scheme should be designed based on a history of references and current references.
2) It should not give a degraded performance for long-running infrequent queries.
3) It should take into account the periodic behavior of block references generated by various types of clients in cloud environment.
4) It should be simple and least computationally intensive.
5) The computationally intensive part should be calculated off-line by extracting frequent block referencing patterns from historical data.
6) The designed solutions should be able to estimate the size of the working set for different time intervals.
7) The buffer management scheme should be adaptive to changing access patterns.

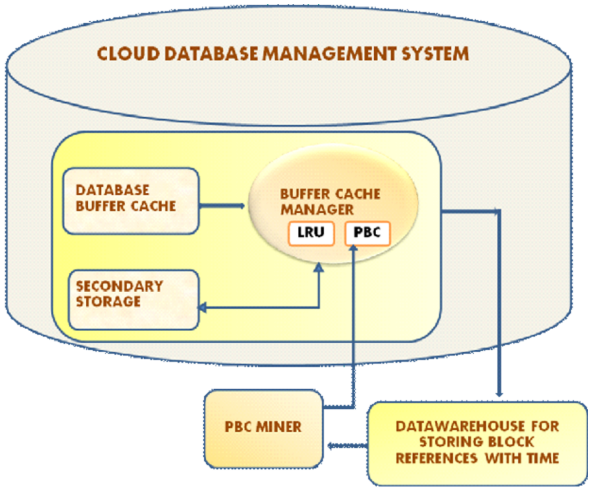

FIGURE 1. Preparation of predictive buffer cache (PBC).

Based on the above solution characteristics we proposed two novel algorithms for block replacement in the buffer management.

F. CONTRIBUTION

1) We proposed a model to estimate future probability of reference using 'Parzen Windows' estimation based on historical periodic references and recent references.
2) We proposed a replacement policy based on future probability of reference.
3) We have done exhaustive experimentation on standard and synthetic datasets and shown that the proposed replacement policy gives better results in terms of hit ratio which is considered as performance measure.

III. PROPOSED WORK

A. ARCHITECTURE OF THE PROPOSED MODEL

The system architecture is described using the diagrams Fig. 1 to Fig. 3.

The components used in the architecture are described as follows:

1) **Database Buffer Cache:** It holds the memory image of disk blocks which are having high probability of reference.
2) **Buffer Cache Manager:** It provides replacement policy for managing a buffer cache.
3) **Data-warehouse:** It holds the reference trace of the blocks in the form of a list of $< blockid, time\_of\_reference >$.
4) **Predictive Buffer Cache (PBC) Miner:** It mines the list of blocks having high probability of reference in the specified time interval.
5) **Least Recently Used (LRU) list:** It holds the list of block ids according to time of reference sorted in descending order.

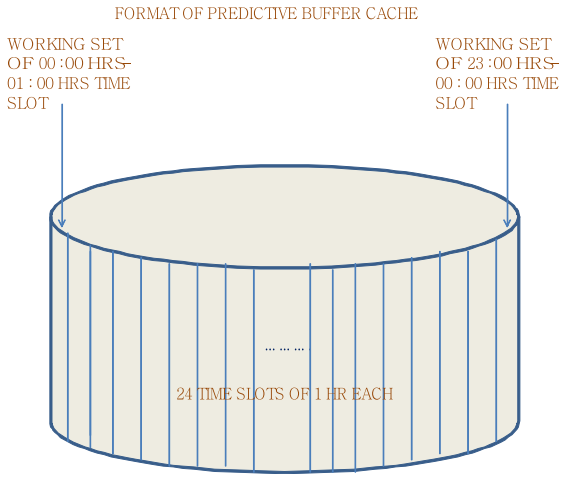

FIGURE 2. Format of Predictive Buffer Cache.

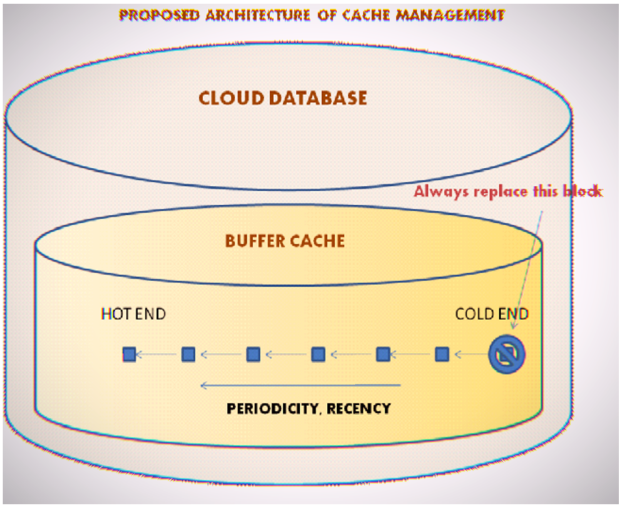

FIGURE 3. Modified Model of Cache Management.

A buffer cache is to be monitored continuously 24 X 7 and the details about block references are collected in the warehousing system. A buffer cache reader (invokes after every 15 minutes) will examine all the blocks in the buffer cache and will store these block identifiers along with the time-stamp of reference in the data-warehousing system. After every 7 days, a buffer Cache Miner (PBC) will be scheduled offline which will analyze last 7 days data from data warehouse to find the working set for each time interval. Thus it defines the predictive buffer cache which contains the working sets for all the time intervals as shown in Fig 2. Each time interval is static, i.e., of the fixed length of 60 minutes and there will be total 24-time intervals in a 24 hours date-time. This process of finding the working sets for different time intervals is done off-line. Database System while processing the queries online will make use of these working sets for making the block replacement decisions. In general, a block which belongs to the working set of the current time interval will not be replaced as shown in Fig 3.

Notation	Meaning
N	Total number of block references in time period T.
T	Total time period of data collection specified in small time units.
τ_i	Time instance when block B is referred in the specified interval.
S	indicates set of time units where a block is referred.
P-value	Periodicity of reference.
L-value	Recency of reference.
σ	indicates the effect of reference on future pattern.

B. COMPUTING BLOCK ACCESS PROBABILITY USING PARZEN WINDOW ESTIMATION

Each block in the working set is assigned the predictive value termed as its rank. This value is based on the probability of a block derived from the historical data. For finding probability of access of a block Parzen Window Density Estimation is used.

The block references are stored in following format.

$$S = \{< b1, t1, d1 >, < b2, t2, d2 >, < b3, t3, d3 >, \ldots, < bn, tn, dn >\}.$$

where b_i is the block id, t_i is the time of access and d_i is the date of access. This data is available in dynamic views in most of the database management systems. We assume that one week's data gives us sufficient information to perform our experimentation and to demonstrate the proof of concept. Block reference patterns are governed by the patterns of user access.

Using Parzen window classifier, probability density estimation function of each block is estimated. (Parzen window is used because the approximate estimation can be chosen by changing distribution parameters), For experimentation, we have used Parzen window with a normal distribution [equations (1) and (2)] [15],

$$P(B) = \frac{1}{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-1}{2}\left(\frac{T-\tau_i}{\sigma}\right)^2} \quad (1)$$

$$P(B) = \frac{1}{N}\sum_{i=1}^{k}\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-1}{2}\left(\frac{T-\tau_i}{\sigma}\right)^2} \quad (2)$$

Let

N : Total number of block references in time period T.

T : 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 * 1440 = 14400 where 1440 is number of minutes for one day, i.e., 24 hours.

τ_i – Time instance when block B is referred in the specified interval.

S = $\tau_1, \tau_2, \tau_3, \tau_4, \ldots, \tau_k$ indicates set of time units where a block is referred. For example if a block is referred at 11 am on day 1 and 11.05 am on day 2 then S = {660, (1440+665)}

P : Periodicity of reference. For example block referencing pattern is repeated for each day then P = 1440, i.e., number of time units in one day.

σ : This is user defined parameter. The more value of σ indicates more effect of reference on future pattern. In the algorithm value of σ is chosen as 1.

TABLE 1. Sample historical references.

Time	Block ID
2.11	10
2.12	20
2.13	30
2.14	20

Using Parzen window technique, the function of possibility of reference of block B is estimated as,

$$P(B) = \frac{1}{N}\sum_{i=1}^{k}\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-1}{2}\left(\frac{T-\tau_i \mod P}{\sigma}\right)^2} \quad (3)$$

In equation (3), P(B) is not exactly probability density function because

$$\int_{-\infty}^{+\infty} P(B)dt = \frac{k}{N} \quad (4)$$

Equation (4) may not be equal to 1 in general case. For any interval between T_1 and T_2 the probability of reference of block B is estimated by equation (5)

$$\int_{T_1}^{T_2} P(B)\, dt \quad (5)$$

For example probability of referring block in time interval 11.00 am to 11.05 am is

$$\int_{660}^{665} P(B)\, dt \quad (6)$$

If M = $\{B_1, B_2, B_3, \ldots, B_l\}$ is a block set under consideration then,

$$P\{B_1, B_2, B_3, \ldots, B_l\} = \sum_{i=1}^{l} P(B_i) \quad (7)$$

Assuming there is periodicity of reference, modeling probability using above method offers following advantages:

- The above mentioned method gives way to estimate probability in specified reference of time.
- The estimated probability is increased if block is referred more in the specified interval.
- The accuracy of estimated probability is increased if T is increased and it converges to actual probability.
- For any interval, the probability of reference is non-zero unless the block is not referred in the interval T.

Sample historical references are generated as shown in Table 1:

The probability of reference (P-value) of block ID 20 for the time interval 2:15 to 2.20 is calculated as,

$$P_{bi} = \int_{2.15}^{2.20}\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-1}{2}(\frac{t-2.12}{\sigma})^2}dt + \int_{2.15}^{2.20}\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-1}{2}(\frac{t-2.14}{\sigma})^2}dt \quad (8)$$

The probability value of each block is calculated and the ranks are assigned using it. The value of σ determines the effect of reference on the probability of the future reference of the same block and it is determined using experimentation by choosing the value which gives minimum number of cache misses.

C. SELECTING BLOCKS FOR THE WORKING SET

For each time interval, blocks are sorted in descending order of historical probability and first K blocks are chosen where K is selected between 10 % to 20 % of the buffer cache size. The value of K decides the size of the working set based on the historical data. If the historical data shows more repetitions then K can be set to a higher value. The value of K cannot be very high otherwise, effect of recent references will not be considered.

D. ASSIGNING RANKS TO THE WORKING SET BLOCKS

Within each working set, a block(s) with the highest probability will be given rank 0. The block(s) with next highest probability will be given rank 1 and so on. Blocks with equal probability will be having the same rank. Rank indicates the historical probability of a block which is derived from recent weeks historical data. Lower the rank value higher is the probability of a block of getting referenced in near future. Every cached block will have two values associated with it. They are

- P-value: P-value indicates the predictive value (importance) of a cached block which depends on its historical rank union its reference behavior after joining the buffer cache.
- L-value: L-value indicates recency value, i.e., LRU sequence value of a cached block. In LRU sequence, least recently referred block will be having sequence value 0 and the most recently referred block will be having sequence value N-1 in the cache full condition. All other cached blocks will be having LRU sequence values between 0 and N-1. For a particular block B, this value depends on how many blocks in the cache is having its last reference before B's last reference. Here N is the size of a cache. Unlike P-value, LRU sequence value called L-value will be unique for each cached block.

E. SNAPSHOT OF PORTION OF SAMPLE RUN OF GENERAL PREDICTIVE POLICY WITH FIXED P-VALUE

Block Reference Trace:
30 90 91 10 92 90 30 96 20 10 10 40 97 20 25 30 10 40 50
ID's of the Working Set Blocks
10 20 25 30
Ranks of the Working Set Blocks:
00 01 02 03
Buffer Cache Size: 5
Beta = 2 (Constant)
Buffer Cache

Note: First 5 faults will fill the buffer cache. References till the first 5 faults are not shown.

The following table demonstrates the working principle of predictive caching which is using the fixed historical rank (in this example) of each cached block along with recency value to organize the cache.

Reference	High				Low	Miss or Hit
92	10	30	92	91	90	Miss
P-value	1	3	4	4	4	
L-value	3	0	4	2	1	
90	10	90	30	92	91	Hit
P-value	1	3	3	4	4	
L-value	2	4	0	3	1	
30	10	30	90	92	91	Hit
P-value	1	2	3	4	4	
L-value	1	4	3	2	0	
96	10	30	90	96	92	Miss
P-value	1	2	3	4	4	
L-value	0	3	2	4	1	
20	10	20	30	90	96	Miss
P-value	1	2	2	3	4	
L-value	0	4	2	1	3	
10	10	20	30	90	96	Hit
P-value	1	2	2	3	4	
L-value	4	3	1	0	2	
10	10	20	30	90	96	Hit
P-value	1	2	2	3	4	
L-value	4	3	1	0	2	
40	10	20	30	90	40	Miss
P-value	1	2	2	3	4	
L-value	3	2	1	0	4	
97	10	20	30	90	97	Miss
P-value	1	2	2	3	4	
L-value	3	2	1	0	4	
20	20	10	30	90	97	Hit
P-value	0	1	2	3	4	
L-value	4	2	1	0	3	
25	20	10	25	30	90	Miss
P-value	0	1	2	2	3	
L-value	3	2	4	1	0	
30	20	30	10	25	90	Hit
P-value	0	1	1	2	3	
L-value	2	4	1	3	0	
10	20	10	30	25	90	Hit
P-value	0	1	1	2	3	
L-value	1	4	3	2	0	
40	20	10	30	25	40	Miss
P-value	0	1	1	2	4	
L-value	0	3	2	1	4	
50	20	10	30	25	50	Miss
P-value	0	1	1	2	4	
L-value	0	3	2	1	4	

Total Block References : 19
No. of Faults : 12
No. of Hits : 07
Hit Ratio : 36.84 %

IV. PREDICTIVE CACHE MANAGEMENT ALGORITHMS

We have designed and implemented two novel cache management algorithms which are described in the following sections. One of the common features amongst them is that they both are based on predictive optimization and uses the predictive working sets described in the above section. If the system is currently working in time interval 10.00 am to 11.00 am then working set corresponding to time interval 10.00 am to 11.00 am will be used, If the system is currently working in time interval 6.00 pm to 7.00 pm then a working set corresponding to time interval 6.00 pm to 7.00 pm will be used.

A. PREDICTIVE LRLFU ALGORITHM

Organization of a buffer Cache: The blocks in a buffer cache are kept sorted with respect to P-value and L-value, i.e., cached blocks are maintained in increasing order of their P-value, and blocks having equal P-value are arranged in decreasing order of their L-value. Here higher P-value means lower predictive probability and higher L-value means more recently referenced cached block. The sequence formed by following the above conditions is termed as sorted sequence. Every replacement occurs at the lowest end of this sequence which is also called as low end or replace end of the cache. This end always contains the least recently referred block amongst cached blocks having lowest predictive importance (,i.e., highest P-value). Few key features related to the organization of a predictive buffer cache are as follows.

The High, i.e., Safe end of a buffer cache contains a block with lowest P-value. In case there are multiple blocks with lowest P-value, High end contains the block with lowest P-value and highest L-value (within lowest P-value blocks).

In other words, the Safe end contains the most recently referred block amongst the cached blocks with highest predictive importance (numerically lowest P-value).

The Low, i.e., Replace end of a buffer cache contains a block with highest P-value or in case there are multiple blocks with highest P-value, Low end contains the block with highest P-value and lowest L-value (within highest P-value blocks). In other words, the Replace end contains the least recently referred block amongst the cached blocks with lowest predictive importance (numerically highest P-value).

The predictive importance (P-value) of a cached block at any time instance is the union of its historical importance (Rank) and the relative access pattern it has exhibited after joining the cache. Here relative means with respect to access patterns of the other cached blocks.

Assumptions and Terminologies:

RefT: Reference Trace.

N-RefT: Number of references in a Reference Trace.

BC: Buffer Cache.

N: Buffer cache size.

Blk: Referred Block.

P-value: Predictive value indicating block reference probability of a cached block Blk.

L-value: LRU sequence value of a cached block Blk.

Rank: historical value (rank) of a working set block. ***WS:*** A Working Set.

DefRank: Default historical value for every non-working set block = N-1.

Beta: a scope variable; which is set to some algorithmic constant indicating the amount of weight-age given to the current reference pattern of a normal block (non-working set block).

Isemptyframe (): a procedure which returns -1 if a buffer Cache is full else return 0.

DeleteShiftandPlace (Blk): deletes the low end block and insert new block at the sorted position, i.e., the resultant sequence should have the ascending order on P-value and for the blocks with same P-value, descending order sequence on L-value.

ShiftandPlace (Blk): insert new block at the sorted position.

WS(Blk).Rank: Rank of block Blk in WS

Move (Blk): Relocates the Blk according to its changed P-value and L-value.

B. THE WORKING OF PREDICTIVE LRLFU ALGORITHM

When page fault is serviced, working set blocks will join the cache at different points in the hot region based on its historical rank, after that based on the reference pattern it follows relative to other cached blocks, it will either move towards the safe end or towards the replace end. Hence P-value of a block at any given point of time during its stay in the cache is the union of its historical rank and the relative reference behavior it exhibits after joining the cache.

Whenever a fault occur for a particular block X, X gets cached and P-value is computed for it. For the first time,

Algorithm 1 Predictive LRLFU Algorithm

```
1:  Input: RefT, N-RefT, BC, N
2:  for i=1; i ≤ N-RefT; i++ do        ▷ processing a reference trace
3:      Blk = RefT(i)
4:      if Blk ∈ BC then                                  ▷ Cache Hit
5:          if Blk ∈ WS then
6:              Blk.P-value = Max( 0, ceil (0.5 * Blk.P-value) + ceil (0.5 * WS(Blk).Rank) − Beta)
7:          else
8:              Blk.P-value = Max( 0, ceil (0.5 * Blk.P-value) + ceil (0.5 * DefRank) − Beta)
9:          end if
10:         Move (Blk);
11:     else
12:         if Blk ∈ WS then                                  ▷ Cache Miss
13:             Blk.P-value = ceil (0.5 * N) + ceil (0.5 * WS(Blk).Rank) − Beta
14:         else
15:             Blk.P-value = ceil (0.5 * N) + ceil (0.5 * DefRank) − Beta
16:         end if
17:         if isemptyframe () != -1 then
18:             ShiftandPlace (Blk);
19:         else
20:             DeleteShiftandPlace (Blk);
21:         end if
22:     end if
23: end for
```

P-value of a faulted block is initialized to its rank value in case of a working set block and to a default rank value for any normal block. Initialization of P-value of any block which is brought to cache on fault occurrence is given below:

1) Block X belongs to the working set of T. In this case, block X's P-value will be initialized to the rank of X.
2) Block X does not belong to the working set of T. In this case, block X's P-value will be initialized to the default rank (lowest possible rank) which is N-1. Here N is the buffer cache size.

When any working set block joins the buffer cache, its initial P-value is computed by ceil (0.5 * N) + ceil (0.5 * Rank) − Beta, where N is a buffer Cache Size, Rank is the rank of a working set block and Beta is the value of a scope variable. For normal blocks, P-value of block is computed by ceil (0.5 * N) + ceil (0.5 * defRank) − Beta. Lower the numerical value of the rank higher is the predictive importance of a block, and more close to the safe end it will join the cache. Higher the numerical value of the rank lower is the predictive importance of a block, and more close to the replacement end it will join the cache. When a working set block is re-referenced while present in cache its P-value is recomputed, using its current P-value and historical rank as follows:

P-value = Max(0, ceil (0.5 * P-value) + ceil (0.5 * Rank) − Beta)

For normal blocks P-value is recomputed as,
P-value = Max(0, ceil (0.5 * P-value) + ceil (0.5 * defRank) − Beta)

C. USE OF SCOPE VARIABLE BETA

If Beta variable is set to LOW value (Beta value in the range of 0% to 20% of Buffer Cache Size) algorithm ensures the following things: (assuming buffer full condition)

- Normal blocks will join the buffer cache in its low portion and will get quickly replaced due to the next new blocks joining the cache.
- Usually a cached block move towards the safe end after quick re-references to it. It can move close to the safe end by scoring over other cached blocks based on current access patterns. However, normal block may find slow (small) transition towards hot end on each re-reference. It may get locked after certain amount of transition, and after that re-references will not be able to displace it further towards safe end. Eventually, when the access patterns changes and its percentage of references gets reduced, it will flow towards the replace end and will be replaced. This also protects stabilized working set blocks in the cache, from getting replaced due to a short burst of dense and large number of references to normal blocks (Explained in Example 1 below).

Example 1

- Let B be a Working Set Block and X be a Normal Block. For the sake of simplicity we assume that, between references there is no alteration in location of block B or block X due to references to other blocks.
  For Block B:
  Assume N (Buffer Cache Size) = 10, Rank of block, B (R) = 4, scope variable Beta = 2;
  On first reference to block B, P-value = ceil (0.5*10) + ceil (0.5*4) − 2 = 5;
  On second reference, P-value = ceil (0.5*5) + ceil (0.5*4) − 2 = 3;
  On third reference, P-value = ceil (0.5*3) + ceil (0.5*4) − 2 = 2;
  On fourth reference, P-value = ceil (0.5*2) + ceil (0.5*4) − 2 = 1;
  On fifth reference, P-value = ceil (0.5*1) + ceil (0.5*4) − 2 = 1;
  Now calculating the same for a normal block X
  For normal blocks Rank = N.
  As N = 10, Rank of block X (R) = 10;
  N (Buffer Cache Size) = 10, scope variable Beta = 2;
  For Block X:
  for beta = 2;
  On first reference to block X, P-value = ceil (0.5*10) + ceil (0.5*10) − 2 = 8;
  On second reference, P-value = ceil (0.5*8) + ceil (0.5*10) − 2 = 7;
  On third reference, P-value = ceil (0.5*7) + ceil (0.5*10) − 2 = 7;
  On fourth reference, P-value = ceil (0.5*7) + ceil (0.5*10) − 2 = 7; and so on...
  if we change scope variable Beta to 3;
  For Block X:
  for beta = 3
  On first reference to block X, P-value = ceil (0.5*10) + ceil (0.5*10) − 3 = 7;
  On second reference, P-value = ceil (0.5*7) + ceil (0.5*10) − 3 = 6;
  On third reference, P-value = ceil (0.5*6) + ceil (0.5*10) − 3 = 5;
  On fourth reference, P-value = ceil (0.5*5) + ceil (0.5*10) − 3 = 5; and so on... **For Block X:**
  for beta = 4
  On first reference to block X, P-value = ceil (0.5*10) + ceil (0.5*10) − 4 = 6;
  On second reference, P-value = ceil (0.5*6) + ceil (0.5*10) − 4 = 4;
  On third reference, P-value = ceil (0.5*4) + ceil (0.5*10) − 4 = 3;
  On fourth reference, P-value = ceil (0.5*3) + ceil (0.5*10) − 4 = 3; and so on...
  As demonstrated in the above example P-value of a working set block is coming down to 1 after few references to it irrespective of what the beta value is.
  For normal block with scope variable Beta = 2, P-value is not decreasing below 7 and with Beta = 3 it is coming down upto 5 but not going below 5, with Beta = 4 P-value is coming down to 3 but not going below 3. For higher value of Beta it may come further down towards 0 or 1. Hence Beta decides the scope of normal blocks to move to safe end. Moving towards the safe end increases the stay of a block in the cache.

If Beta variable is set to HIGH value (Beta value in the range of 21% to 50% of Buffer Cache Size) algorithm ensures the following things: (assuming buffer full condition)

- While joining the buffer cache, normal blocks will be inserted around the middle of the cache, ensuring relatively longer stay to them before replacement.
- Normal blocks will find bigger scope for moving into a high portion towards Safe End, by scoring over other cached blocks (normal as well as the working set blocks) based on its positive differences with their current access patterns.
- A normal block whose recent access patterns are relatively more frequent as compared to some better positioned blocks, can move ahead of those blocks to further increase its stay in the cache.
- Based on the higher current access patterns, a normal block may overtake a better positioned working set blocks also.

A block whose current access pattern is frequent, its stay will give more hits to the cache performance as long as its access pattern stays frequent. In normal conditions, when the working set blocks are exhibiting patterns close to their

historical patterns they will remain stable in the cache in its high portion. In case few of them lost their frequent patterns for some period of time, they cannot be driven by the normal blocks towards replacement. This is true only with Beta variable set to LOW. In case Beta variable is set to a HIGH value, the working set blocks with reversed patterns can be driven to replacement by normal blocks with higher frequency of access.

Other way, if B is a working set block, and between any two references to block B if there are y number of references to the other blocks, with Beta set to LOW, only a subset of y (corresponding to a working set block references) may push block B towards replace end. If B is a normal block then all the y references to other blocks will push B towards replace end till it gets replaced.

Sample Case Consider the cached blocks X and Y both having high frequency of access as compared to other cached blocks. Assume X to be a working set block and Y to be a normal block. At the time of joining, X will be placed in the HIGH portion of the cache whereas Y will be placed in the LOW portion of the cache.

- Case 1: scope variable Beta set to LOW
  - In this case, block X can cross all the cached blocks due to its higher frequency of access.
  - Block Y can cross only cached normal blocks due to its higher frequency of access.
- Case 2: scope variable Beta set to HIGH
  - In this case both blocks X & Y can reach the safe end.
    - If current access frequency of normal block Y is more as compared to current access frequency of a working set block X then taking some time, block Y may cross block X. Time taken depends on the difference in their ranks and the difference between their current access patterns. In case if the access frequency of block Y is less, then block Y cannot move ahead of block X towards safe end.

One of the important difference between a working set block and a normal block is that at each re-reference a working set block will take relatively larger steps (number of blocks crossed) towards safe end as compared to normal block.

Significance of L-value of a cached block

- In the cache, each block has the unique L-value. Hence it can be used to break the tie when there are multiple blocks having lowest predictive importance.
- As the cache is ordered based on increasing P-value and decreasing L-value, victim block is always present at the low end.
- In case of multiple cached blocks having highest P-value, low end of the cache always contains a block with lowest L-value among the blocks with highest P-value.
- Hence replacing low end block means replacing the least recently used block amongst the group of lowest predictive probability blocks in the cache.

D. ADVANTAGES OF PREDICTIVE LRLFU ALGORITHM

- One of the problems in database cache optimization is handling time consuming infrequent queries which requires large amount of sequential disk block access. While processing such queries, whole existing buffers will be replaced with the blocks required for executing these queries. This will lead to removal of the existing and stable working set in the cache and can results in lot of cache misses afterwards. The blocks of such long infrequent queries will not appear in the working set due to its infrequent access pattern and with the policy of predictive LRLFU algorithm all of them will join the buffer cache close to replace end (with parameter Beta set around LOW value) and will be replaced quickly. In case of quick re-references also such blocks will stay in the LOW portion of the cache.
- Second advantage of LRLFU cache management policy is that it gives very high performance in handling suitable workloads in which the historically frequent blocks are having the frequent patterns in the current trace.

E. DISADVANTAGES OF ABOVE IMPLEMENTATION OF PREDICTIVE LRLFU ALGORITHM

- ***Slow-adaptability to reversals of historical patterns due to static value of scope variable:***
  Predictive LRLFU algorithm can give considerable improvement in the hit ratio especially when higher percent of the current trace references are periodic in nature. But this algorithm has some disadvantage when the working set blocks gets cached but does not deliver substantial number of hits. Additional stay of such blocks may enforce some of the better access probability blocks to leave the cache. To avoid this, we can make additional provision in our algorithm to improve the adaptability to handle the lost memorized patterns or to maintain good performance despite of less percentage of periodic references. Section 4.6 explains the new algorithm having a online adaptability to reversal in the frequent patterns of the working set blocks.
- ***Some computational overheads:***
  There are some computational overheads of computing P-value, updating L-value for each referred block for its every reference, and maintaining the list sorted according to P-value, L-value combination. These overheads can be controlled effectively by using suitable data structures like linked list and hash map as described in the section 4.7. We can use the same data structures to improve time efficiency of Adaptive LRLFU.

F. ADAPTIVE LRLFU ALGORITHM

Adaptive LRLFU is modified Predictive LRLFU with additional online adaptibility. Like Predictive LRLFU, in adaptive LRLFU also scope variable Beta decides the division of cache into two portions, which are, a working set portion and a non-working set portion. Beta also decides the scope of normal block in prolonging its stay in the cache based on its current access patterns. Additionally, Adaptive LRLFU algorithm keeps track of recently replaced data elements from a buffer cache. It maintains this record in dummy LRU list. The size of dummy list is equal to the size of a buffer cache. In case of a reference to some working set block, if there is a miss in a buffer cache and hit in the dummy list, then the referred block is deleted from the dummy list and added to a buffer cache with its P-value computed using its rank value specified in the current time intervals working set. In case of a reference to some normal block, if there is a miss in a buffer cache and hit in the dummy list, then the referred block is deleted from the dummy list and added to a buffer cache with its P-value computed using Default Rank value overridden with $\frac{1}{2}$ of actual default rank value which is half the size of a buffer cache.

Adaptability:

It alters the value of scope variable Beta by obeying the following rules:

If a new block reference is a miss in a buffer cache but hit in the dummy list and that reference is for the working set block, then reduce the scope of the normal blocks by reducing Beta by 1, provided Beta is more than 1% of the buffer cache size. In case the new reference is for the normal block and it is a miss in a buffer cache and hit in a dummy list then increase the value of Beta by 1, provided Beta is less than 50% of a buffer cache size thereby increasing the scope for normal blocks to reach the safe end. Hence Adaptive LRLFU adjusts the scope of normal blocks by examining a current buffer cache misses in the history of recently replaced blocks, to minimize the negative effect of reversed or lost memorized patterns. It also gives improved position to normal blocks which are recently replaced and referred again.

G. TIME EFFICIENT IMPLEMENTATION OF PREDICTIVE LRLFU

Hashmap is a hashmap and Cachelinked List is a doubly linked list of node type BLOCKNODE having HEAD and TAIL node pointers pointing to first and last node respectively in the CacheLinked List. BLOCKNODE has following fields: BLOCKID, PREV, NEXT, P-value, L-value;

BLOCKID corresponds to key in hashmap.

CachelinkedList is always sorted on increasing P-value, decreasing L-value. This order will be maintained at each insertion of a new blocknode (in case of cache miss) and at each updation of P-value, L-value of referred blocknode (in case of a cache hit). Time efficient Predictive LRLFU works as follows:

Terminologies used: Rank is the historical rank of the referred working set blocks and DefRank is the default rank used for normal blocks, Beta is scope variable. The various procedures called in the algorithm works as follows:

Algorithm 2 Time Efficient Predictive LRLFU

```
1:  Input: hashmap Hashmap, block reference trace RefT, size of trace N-RefT, Buffer Cache Size N.
2:  for i=1; i ≤ N-RefT; i++ do        ▷ processing a reference trace
3:      if RefT(i) ∈ Hashmap then                ▷ Cache Hit
4:          Hits++
5:          Blocknode = Hashmap.get(RefT(i))
6:          Blocknode.recomputeP-value ()    ▷ recomputes P-value
7:          Blocknode.recomputeL-value ()    ▷ recomputes L-value
8:          Cachelinkedlist.updateposition(Blocknode)
9:      else
10:         Faults++;
11:         if number of nodes in CachelinkedList == N then
12:             Remove the tail of the Cachelinkedlist and the corresponding blocknode from the hashmap.  ▷ deletes block at replace end of the cachelinkedlist and the corresponding entry in the hashmap
13:         end if
14:         Blocknode = Cachelinkedlist.addnewblock (RefT(i))
15:         Hashmap.put (RefT(i), Blocknode)
16:     end if
17: end for
```

- ***Cachelinkedlist.addnewblock (Bi)*** will create a new node and sets Bi as BLOCKID, sets P-value field to ceil (0.5 * N) + ceil (0.5 * Rank) - Beta; if it is a working set block or to ceil (0.5 * N) + ceil (0.5 * DefRank) - Beta; if the block is not in the working set. After that the procedure will scan the sorted linked list from the head node and will add the new node before a node whose P-value is either numerically greater than or equal to that of new node. If no such node was found till the end of CacheLinked List then new node will become the TAIL node of CacheLinked List. If head node has numerically greater or equal P-value than the new node, then new node will become the HEAD node. Procedure returns the New Node which will be added along with its key to HashMap.
- ***Cachelinkedlist.updateposition (Bi)*** will start from referred Blocknode whose P-value & L-value is just updated, scanning left towards head and relocates (if required) the Blocknode at its appropriate position to restore the sorted order. Due to recomputed P-value, L-value, Bi might have get out of order in the otherwise sorted list. Here we are scanning only left starting from Blocknode Bi because only referred nodes P-value may have changed (improved), hence it may be relocated to more closer position to safe end.

TABLE 2. Dataset characteristics for workloads set-A.

Sr.No.	Dataset	N	Blks	P	HS	R	RA
1	DS1	612217	65321	14.45 %	21.48 %	37.69 %	26.38 %
2	DS2	625966	71529	21.24 %	26.00 %	24.51 %	28.25 %
3	DS3	831476	92079	27.45 %	10.08 %	30.31 %	32.16 %
4	DS4	21221366	2597474	21.10 %	14.12 %	34.45 %	30.33 %

TABLE 3. Dataset characteristics for workloads set-B.

Sr.No.	Dataset	N	Blks	P	HS	R	RA
1	DS5	671724	44605	04.05 %	87.15 %	04.11 %	4.69 %
2	DS6	451274	67783	03.21 %	78.35 %	05.44 %	13.00 %
3	DS7	421561	111039	01.12 %	01.47 %	24.07 %	73.34 %
4	DS8	24532574	2549518	10.84 %	6.92 %	14.20 %	68.04 %

H. TIME COMPLEXITY OF TIME EFFICIENT PREDICTIVE LRLFU

At each updation of P-value, L-value of cached block (due to its rereference) only X+Y comparisons are required ($\theta(n + m)$). Here X indicates the number of blocknodes whose P-value is between the old P-value and the new P-value of the referred block. Y is the number of blocknodes in the CacheLinked List having same P-value as the new P-value of referred block. Worst case complexity of our algorithm is O(N) as in worst case N = X+Y. In best case, our algorithm will have time complexity of O(1).

When a hit occurs. only referred block may get relocated to a new position due to its recomputed P-value and changed L-value. The only possibility is new position of a referred block may be closer to safe end than its previous position. Rest of the block sequence remains unvisited as there is no change in P-value, L-value of any other cached block. Here as we are doing insertion in a sorted linked list, the time complexity of insert operation (which corresponds to reading a new block in a cache) can be reduced to $O(Log_2N)$ from O (N). The time complexity of delete operation (which corresponds to creating a space for a new requested block by removing a victim block from the cache) is O (1) as we are deleting TAIL node at each deletion. Deletion happens at each new Blocknode addition to cache when the cache is full. Search complexity is also O (1) as we are using a hashmap having time complexity of O (1) for get () and put () operation.

V. EXPERIMENTATIONS

A. DATASETS AND EXPERIMENTAL ENVIRONMENT

We have implemented the proposed methods and existing methods in java using NetBeans IDE on UNIX platform. We have performed trace based simulation to evaluate the performance of proposed algorithms in comparison with the implemented existing algorithms.

We have developed a data set generator application DBGEN which generates reference traces having mixture of periodic references, sequential and hierarchical references, repeatable references and random references. Using DBGEN we have generated total 8 datasets out of which dataset1-dataset4 and dataset5-dataset8 contains one hour reference traces for seven days. The first group contains the workloads with balanced mixture of different types of references, whereas second group contain the workloads having high percent of one type of references. Each reference in these datasets are time stamped between time interval 10.00 am to 11.00 am. Dataset4 and Dataset8 contains 24-hours reference trace for 7 days in which each reference is having a timestamp between 12:00 am - 11.59 pm. In implementation we have used numbers between 1 to 1440 to uniquely identify total 1440 minutes during a day-time. These numbers are used to timestamp each reference in the datasets. The characteristics of all the 8 datasets are given in Tables 2 and 3.

For the six 1-hour datasets, out of traces of seven days, traces of six days are used as training datasets for finding historical probability of referred blocks, defining the working sets, and giving ranks to the working set blocks. Reference trace of seventh day is used as test dataset for evaluating performance of algorithms. For the two 24- hour datasets, the working sets for all 24 time intervals is defined by consulting reference traces of six days and seventh day's trace is used for performance evaluation purpose.

The existing algorithms and proposed algorithms are evaluated on test traces with different cache sizes. We have used hit ratio as the performance measure to compare various algorithms. Since hit ratio is affected by the buffer cache size, for various sizes of buffer cache, hit ratio of the buffer management policies are recorded and comparison is shown for all the 8 datasets. Additionally we have used Zipf synthetic trace and Sprite network trace for evaluating the performance of our algorithms.

Zipf follows a Zipf-like distribution where the probability of the i^{th} block being accessed is proportional to $(i/N)^{\alpha}$, where α is equal to log a / log b, where a and b are between 0 and 1, and N are the total number of distinct blocks referred. This approximates common access patterns in web applications that a few blocks are frequently accessed and others are accessed much less often. Zipf trace has 600000 I/O operations and 781 MB of data size. Performance comparison on Zipf is shown in Figure 14.

The Sprite network trace is a real workload trace containing requests to a file server from a client workstation making 165,472 block references to 6,975 unique blocks with the block size of 4 Kbytes is used for comparing the performance

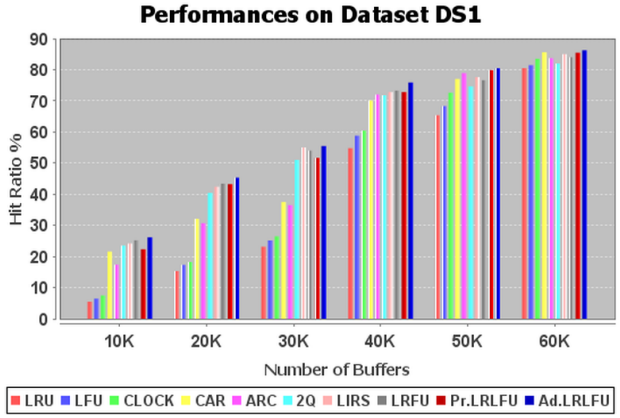

FIGURE 4. Performance comparison on Dataset DS-1 (1 hour trace).

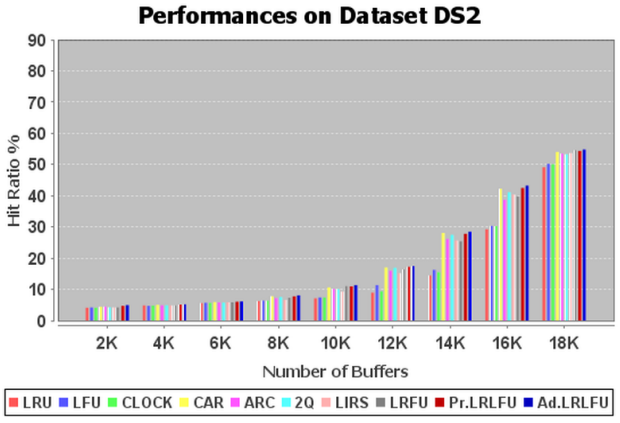

FIGURE 5. Performance comparison on Dataset DS-2 (1 hour trace).

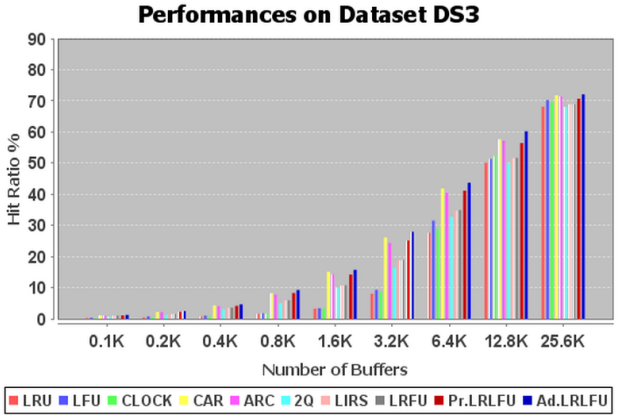

FIGURE 6. Performance comparison on Dataset DS-3 (1 hour trace).

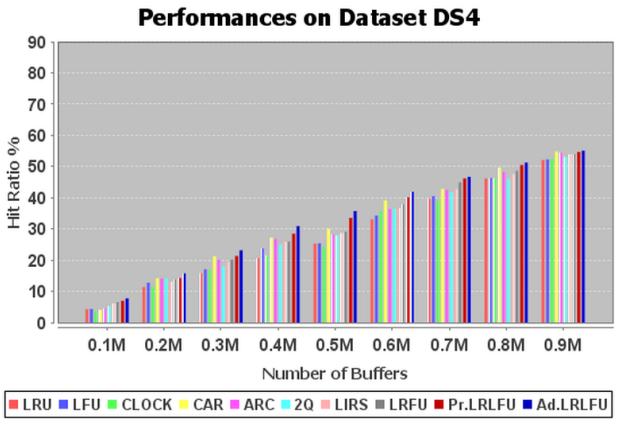

FIGURE 7. Performance comparison on Dataset DS-4 (24 hours trace).

of AD-LRLFU with that of the other best performing algorithms. The performance comparison is shown in Figure 14. For all these experimentations, we have taken initial value of Beta variable = 10% of the buffer cache size. We selected the tunable parameters of LRU-2, 2Q, MQ and LRFU for the best result for each cache size.

B. PERFORMANCE EVALUATION ON THE REAL AND SYNTHESIZED TRACES

N : number of references in a reference trace.

Blks : number of unique blocks in a reference trace.

P : number of periodic references.

HS : number of sequential or hierarchical references.

R : number of repeatable references.

RA : number of random references.

The performance comparison of various algorithms along with the proposed algorithms on Dataset1-Dataset4 are shown below in Figure 4-7.

Dataset1-Dataset4 contains several looping patterns of non-uniform repeat intervals mixed with references to periodic, sequential and randomly accessed blocks. The performance comparisons on these datasets are shown in Fig. 4 to Fig. 7. LRU and Clock considers recency, but recency of a block depends on its own reference activity as well as the recent reference activities of other blocks. As both these algorithms does not take frequency into consideration, they suffers from poor performance on all these datasets. Localities of several blocks will get dispersed due to reference activities of sequential and random blocks. Hence blocks will have varying access patterns, due to which reference frequencies of many frequent blocks will be hard to distinguish in several localities. Hence frequency based algorithms like LFU and 2Q suffers from performance degradation. Most of the adaptive algorithms CAR, LIRS, ARC and recency and frequency based algorithms like LRFU quantifies the likelihood of the block of getting accessed in the near future, by using the historical information. Hence they are capable of replacing the block with weak locality. However these algorithms are unable to detect the access patterns of the historical blocks because of which there performance is considerably less than Adaptive LRLFU algorithm, especially for smaller sizes of the buffer cache. The performance gain in terms of hit ratio for the Adaptive-LRLFU compared to other best performing algorithms LIRS, ARC, CAR was 9%, 8%, and 6% respectively on dataset DS-4.

The performance comparison of our algorithms along with the some of the best performing other algorithms on Dataset5-Dataset8 are shown in Fig. 8 to Fig. 11. All these

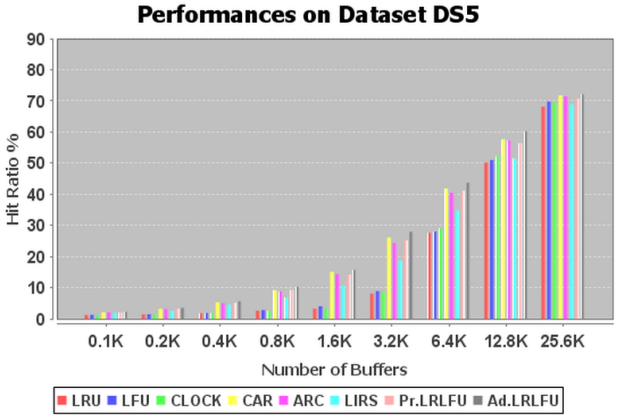

FIGURE 8. Performance comparison on Dataset DS-5 (1 hour trace).

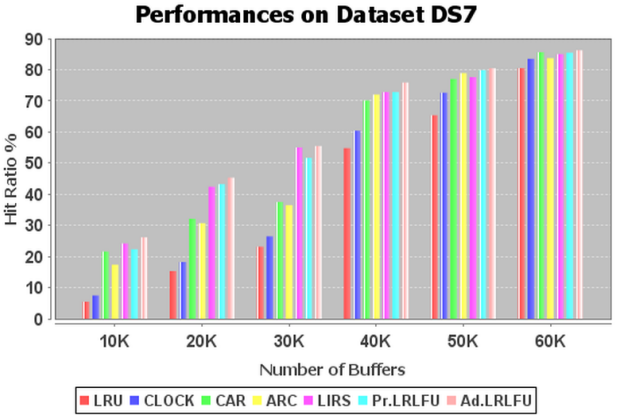

FIGURE 10. Performance comparison on Dataset DS-7 (1 hour trace).

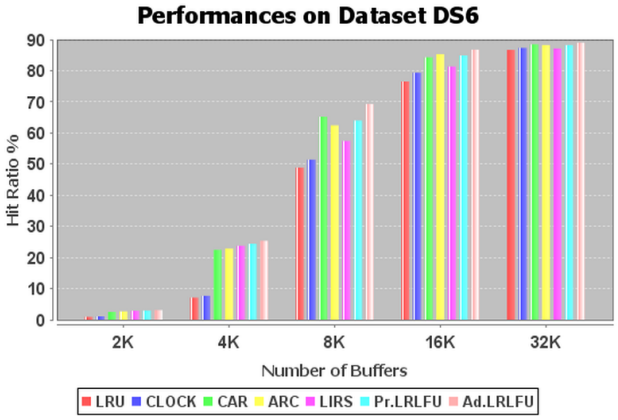

FIGURE 9. Performance comparison on Dataset DS-6 (1 hour trace).

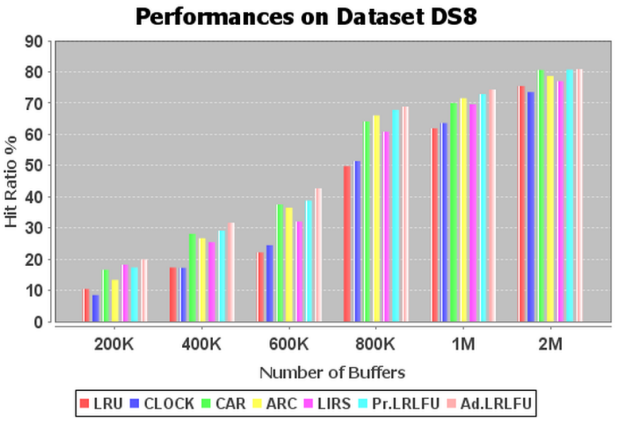

FIGURE 11. Performance comparison on Dataset DS-8 (24 hours trace).

datasets either have long sequences of sequential scans or long burst of random references separated by small percentage of other types of references. In general, all the scan resistant algorithms like ARC, LIRS, CAR as well as Predictive and Adaptive LRLFU performs well on these datasets. Additionally, both the proposed algorithms outperforms the best performing CAR and LIRS on these datasets. As all these datasets are having either high percentage of sequential references or high percentage of random references, with the increase in number of buffers, the trends in the performance improvement of adaptive algorithms gradually stabilizes. On all the four datasets, Adaptive LRLFU has a better hit ratio than the other algorithms, because in addition to quick online adaptability for changing patterns, Adaptive LRLFU takes localities of periodic/random/repeatable references into consideration which gives it a best performance. The dynamically changing parameter Beta gives ability to Adaptive LRLFU to fluctuate from frequency to recency (of periodic as well as non-periodic references) and back, all within a single workload. The performance gain in terms of hit ratio for the Adaptive-LRLFU compared to other best performing algorithms LIRS, ARC, CAR was 6%, 4%, and 3% respectively on dataset DS-8.

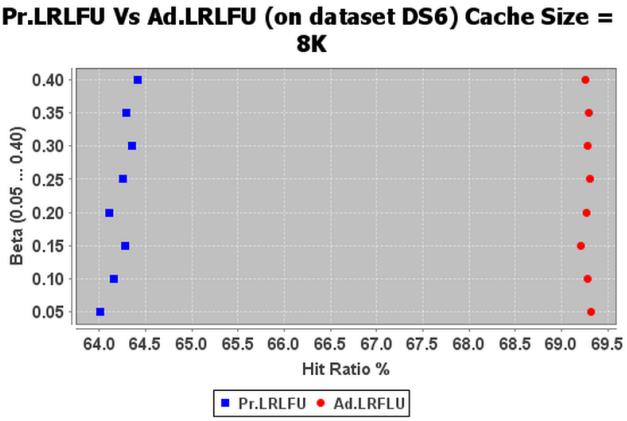

FIGURE 12. Pr.LRLFU Vs Ad.LRLFU Performance Comparison for Different Scope, i.e., Beta Values (on dataset DS-6) Cache Size = 8K.

Sensitivity of Proposed Algorithms for scope parameter Beta: We have checked the performance gain of Predictive LRLFU and Adaptive LRLFU for different values of scope parameter Beta ranging from 0.05 to 0.35. We have varied the value of Beta from 0.05 * buffer cache size to 0.35 * buffer cache size at a step of 0.05 on datasets DS6 and DS8. On the tested workloads, with the change in beta there

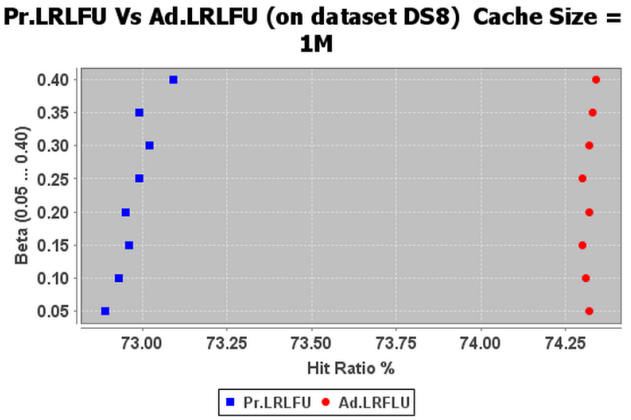

FIGURE 13. Pr.LRLFU Vs Ad.LRLFU Performance Comparison for Different Scope, i.e., Beta Values (on dataset DS-8) Cache Size = 1M.

is a moderate and stable change in the hit ratio. However, Predictive LRLFU is clearly more sensitive to the change in Beta value, as it is observed that the percentage of times we get measurable change in hit ratio with the change in beta is higher in Predictive LRLFU as compared to Adaptive LRLFU.

In general, Increase in Beta value increases the performance gain of Predictive LRLFU on workloads with high percentage of random references. Whereas it gives its best performance with minimum Beta value when tested on workloads with high percentage of periodic and repeatable references. Against this, Adaptive LRLFU is very less sensitive to the changes in value of Beta variable. This is because value of Beta variable is implicitly and dynamically adjusted in Adaptive LRLFU with the changing patterns of block references. The results are shown in Fig. 12 and Fig. 13. Performance comparison on Zipf and Sprite network trace are shown in Fig. 14 and Fig. 15 respectively. Adaptive LRLFU outperforms the competitor algorithms LIRS, ARC, CART and CLOCK on these traces. The major difference between adaptability of competitor algorithms like LIRS, ARC, CAR, CART and our proposed algorithm is that the competitor algorithms while adapting to changing access patterns, do not distinguish between replaced-recently-and-referred-again blocks based on their historical patterns. Whereas, dynamically changing value of the beta and consideration given to historical patterns enables Adaptive LRLFU to recover quickly from the change in locality of working set blocks as well as blocks which are not in working set [explained in detail in section 4, subsection F]. Also due to utilization of periodic reference patterns of the working set blocks, our proposed algorithm is showing more gains (in terms of hit ratio) particularly at smaller cache sizes as compared to CART, ARC and LIRS algorithms which is demonstrated in Fig. 14 and Fig. 15. The DB2 database trace used in researches of papers [3], [35] contains 500,000 references to 75,514 unique blocks. The OLTP trace contains references to a CODASYL database for a one-hour period. This trace consists of 914,145 references to 186,880 unique blocks [35].

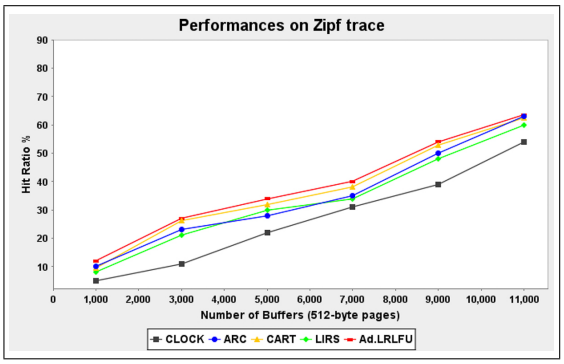

FIGURE 14. Performance comparison on Zipf trace.

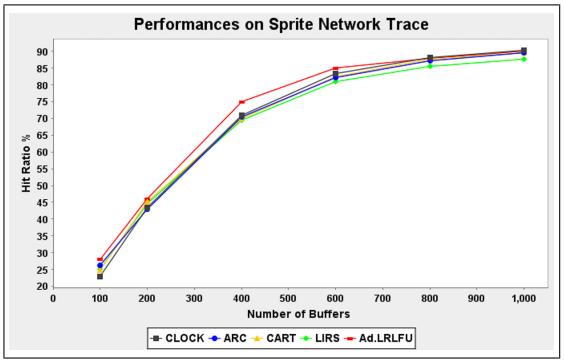

FIGURE 15. Performance comparison on Sprite network trace.

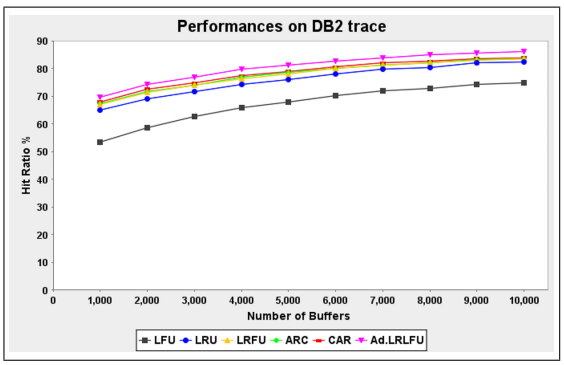

FIGURE 16. Performance comparison on commercial DB2 trace.

We have obtained these 2 database traces from the authors of these papers and the experimental results are shown in Figure 16 and Figure 17. On both traces Adaptive LRLFU has the highest hit ratio for all cache sizes we simulated.

C. RESULTS ANALYSIS

We found that the Adaptive LRLFU algorithm gives the best hit ratios in all the examined workloads. With test data set in general containing less percentage of periodic references, hit ratios of other best performing algorithms and proposed algorithms differs in the small range but still Adaptive LRLFU above all others maintains the highest hit ratio. With datasets containing more percentage of periodic refer-

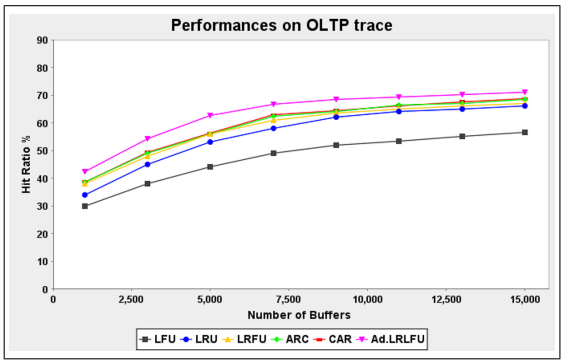

FIGURE 17. Performance comparison on OLTP trace.

ences especially more percentage of non-repetitive periodic references, both Predictive LRLFU and Adaptive LRLFU outperforms other recency and frequency based adaptive algorithms like ARC, CAR, LIRS, CART etc. considerably. The difference in hit ratio between Predictive LRLFU and Adaptive LRLFU is preliminarily due to percentage and distribution of random block references. As such blocks have irregular time or space interval between any two references to them, adaptability of Adaptive LRLFU is better provisioned to give better results than results of Predictive LRLFU.

On the workloads with higher percent of random workload blocks Adaptive LRLFU outperforms other best performing algorithms like CAR, CART, LIRS by a small margin. We also observed that increase in the percentage of random block references have lowest negative effect on the performance of Adaptive LRLFU algorithm as compared to any other algorithm. The increase in the buffer cache size converts some non-repetitive references to repetitive ones and decrease in the buffer cache size converts some of the repetitive references to non-repetitive ones. Keeping the workload constant, change in the buffer cache size leads to change in the ratio of number of non-repetitive references to working set blocks (WS) / number of non-repetitive references to non-working set blocks (NWS). The configurations of a buffer cache for which this ratio is higher, i.e., WS is heavier than NWS, Predictive LRLFU algorithm is close to Adaptive LRLFU performance wise and considerably ahead than the best performing known algorithms.

We also found that difference between hit ratios of proposed algorithms and existing known algorithms keeps increasing with the increase in the buffer cache size as long as Buffer Cache Size is less than Working Set Size. After a particular size of a buffer cache above the working set size the difference in hit ratios between proposed algorithms and existing known algorithms consistently decreases with the constantly increasing the buffer cache size. We also examine that after a particular size of a buffer cache above the working set size, the ratio of increase in the hit ratio percentage with the increase in the buffer cache size is lowest in Adaptive LRLFU as compared to existing known algorithms. Summarizing due to predictive and adaptive feature of Adaptive LRLFU it provide better hit ratios for lesser sizes of a buffer cache and allows more efficient utilization of memory resource.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced new database buffer management algorithms namely predictive LRLFU and Adaptive LRLFU algorithms. The common novelty of both the algorithms is using the periodicity in block access patterns for improving buffer management. Experimental results performed on the variety of datasets suggest that both the algorithms have got the considerable performance advantages over conventional algorithms considering hit ratio as the performance measure. In general, the experimentations acknowledge and demonstrate the following facts.

Due to a combination of predictive and adaptive approach in our proposed algorithms, we believe that the potential benefits of our algorithms, apart from cloud databases will persist in other conventional or non-conventional databases. Its adaptive nature [online and offline adaptability] helps in protecting its performance leverage while dealing with workloads having highly diversified access patterns. The self-tuning property [refreshing the working sets periodically] keeps its performance effective with evolving access patterns. Consideration of recent as well as historical access patterns enables proposed algorithms to learn near to exact future access probabilities of the cached blocks, due to which it maintains the cache replacement sequence accurately. This ensures that in majority of replacements, cached block with the weakest chance of getting referenced soon will be replaced.

We have used nine synthetic workloads, a real sprite network workload and a real DB2 and CODASYL OLTP database traces. in the experiments. The experimental results demonstrate the superiority of our proposed algorithms in comparison to the best of the known buffer management algorithms, especially in the periodicity dominant traces and for the smaller buffer cache sizes. Proposed AD-LRLFU beats the best results of LRU, LIRS, ARC, CAR, 2Q, CART and CLOCK on all the examined workload traces. In addition, the proposed PR-LRLFU and AD-LRLFU algorithms is time efficient having low runtime complexity. From the experimental results, we can conclude that our proposed algorithms improve the performance of the cache in a better way as compared to existing known algorithms. Considering the consistent performance gains of the proposed algorithms in the examined variety of cases we believe that they are good candidates to be used in cloud databases [26], [27] where several applications share the same database instance.

The proposed method uses the static intervals like 9-10 am, 10-11 am and so on for calculating predictive working set. This will give simple and less computational algorithm for replacement but it may not give optimal predictive working sets. Calculating correct intervals to get optimized predictive working sets and to further minimize the cache misses is in the future scope of our work.

REFERENCES

[1] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," *Algorithmica*, vol. 1, nos. 1–4, pp. 311–336, Nov. 1986.

[2] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, pp. 134–142, May 1990.

[3] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD Rec.*, vol. 22. no. 2, pp. 297–306, Jun. 1993.

[4] *Improving WWW Proxies Performance With Greedy-Dual-Size-Frequency Caching Policy*, Cherkasova, Ludmila, Hewlett-Packard Laboratories, 1998.

[5] J. Min *et al.*, "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," in *Proc. 4th Conf. Symp. Oper. Syst. Des. Implement.*, vol. 4, Oct. 2000, p. 9.

[6] D. Lee *et al.*, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Comput.*, vol. 50, no. 12, pp. 1352–1361, Dec. 2001.

[7] J. Song, and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS Perform. Eval. Rev. Meas. model. comput. syst.*, vol. 30, no. 1, pp. 31–42, 2002.

[8] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File Storage Technol.*, Mar. 2003, pp. 115–130.

[9] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proc. 3rd USENIX Conf. File Storage Technol.*, Mar. 2004, pp. 187–200.

[10] S. Jiang and X. Zhang, "Making LRU friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance," *IEEE Trans. Comput.*, vol. 54, no. 8, pp. 939–952, Aug. 2005.

[11] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.* Apr. 2005, p. 35

[12] H. Zhen, R. Lai, and A. Marquez, "On using cache conscious clustering for improving OODBMS performance," *Inf. Softw. Technol.* vol. 48, no. 11, pp. 1073–1082, Nov. 2006,

[13] I. R. Chiang, P. B. Goes, and Z. Zhang, "Periodic cache replacement policy for dynamic content at application server," *Decis. Support Syst.*, vol. 43, no. 2, pp. 336–348, 2007.

[14] S. Wan, Q. Cao, X. He, C. Xie, and C. Wu, "An adaptive cache management using dual LRU stacks to improve buffer cache performance," in *Proc. IEEE Int. Perform. Comput. Commun. Conf.*, Dec. 2008, pp. 43–50.

[15] Z.-W. Pan, D.-H. Xiang, Q.-W. Xiao, and D.-X. Zhou, "Parzen windows for multi-class classification," *J. Complex.*, vol. 24, nos. 5-6, pp. 606–618, 2008

[16] J. Park, H. Lee, S. Hyun, K. Koh, and H. Bahn, "A cost-aware page replacement algorithm for NAND flash based mobile embedded systems," in *Proc. 7th ACM Int. Conf. Embedded Softw.*, Oct. 2009, pp. 315–324.

[17] G. Archana, S. Lakshminarasimhachar, and S. Gopinathan, "Simulation and analysis of cache replacement algorithms," in *Proc. Int. Conf. Comput. Design*, Dec. 2010, pp. 1–41.

[18] G. Rexha, E. Elmazi, and I. Tafa, "A comparison of three page replacement algorithms," *FIFO, LRU, Optimal. Academic J. Interdisciplinary Stud.* vol. 27, no. 4, p. 2, 2015.

[19] L. Yanfei, B. Cui, B. He, and X. Chen, "Operation-aware buffer management in flash-based systems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2011, pp. 13–24.

[20] P. Jin, Y. Ou, T. Härder, and Z. Li, "AD-LRU: An efficient buffer replacement algorithm for flash-based databases," *Data Knowl. Eng.*, vol. 72, pp. 83–102, Feb. 2012.

[21] S. Trausti, "An experimental comparison of cache algorithms," Tech. Rep., 2012.

[22] S. Muthukumar and P. K. Jawahar, "Cache replacement for multi-threaded applications using context based data pattern exploitation technique," *Malaysian J. Comput. Sci.* vol. 26, no. 4, pp. 277–293, 2013.

[23] M. E. Consuegra *et. al*. (2015). "Analyzing adaptive cache replacement strategies." https://arxiv.org/abs/1503.07624

[24] P. Panda, G. Patil, and B. Raveendran, "A survey on replacement strategies in cache memory for embedded systems," in *Proc. IEEE Distrib. Comput. VLSI, Elect. Circuits Robot. (DISCOVER)*, Aug. 2016, pp. 12–17.

[25] C. C. Yang, P. Jin, L. Yue, and P. Yang, "Efficient buffer management for tree indexes on solid state drives," *Int. J. Parallel Program.*, vol. 44, no. 1, pp. 5–25, 2016

[26] K. Kumar and M. Kurhekar, "Economically efficient virtualization over cloud using docker containers," in *Proc. IEEE Int. Conf. Cloud Comput. Emerg. Markets*, Oct. 2016, pp. 95–100.

[27] K. Kumar and M. Kurhekar, "Sentimentalizer: Docker container utility over cloud," in *Proc. 9th IEEE Int. Conf. Adv. Pattern Recognit.*, Dec. 2017, pp. 1–6.

[28] Y. Youwei, Y. Shen, W. Li, D. Yu, L. Yan, and Y. Wang, "PR-LRU: A novel buffer replacement algorithm based on the probability of reference for flash memory," *IEEE Access*, vol. 5, pp. 12626–12634, 2017.

[29] A. K. Gupta and U. Shanker, "SPMC-CRP: A cache replacement policy for location dependent data in mobile environment," *Procedia Comput. Sci.*, vol. 125, pp. 632–639, Jan. 2018,

[30] M. Tinghuai, J. Qu, W. Shen, Y. Tian, A. Al-Dhelaan, and M. Al-Rodhaan, "Weighted greedy dual size frequency based caching replacement algorithm," *IEEE Access*, vol. 6, pp. 7214–7223, 2018.

[31] N. Carlsson and D. V. Eager. (2018). Caching in the clouds: Optimized dynamic cache instantiation in content delivery systems. https://arxiv.org/abs/1803.03914

[32] D. Carra, G. Neglia, and P. Michiardi. (2018). Elastic provisioning of cloud caches: A cost-aware TTL approach. https://arxiv.org/abs/1802.04696

[33] J. Al-Jaroodi and N. Mohamed, "Distributed cloud cache," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2018, pp. 350–351. doi: 10.1109/CCGRID.2018.00-33.

[34] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. Annu. USENIX Tech. Conf.*, Jun. 2001, pp. 91–104.

[35] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algoritm," in *Proc. 20th Int Conf. Very Large Databases*, pp. 439–450, Sep. 1994.

ATUL O. THAKARE received the B.E. degree in CSE from Nagpur University (currently RTM Nagpur University), Nagpur, India, in 1998, and the M.E. degree in CSE from Sant Gadge Baba Amravati University, Amravati, India, in 2013. He is currently pursuing the Ph.D. degree with the Visvesvaraya National Institute of Technology, Nagpur. He has a total of 16-year experience, six years in the IT industry, and ten years in academic profession.

PARAG S. DESHPANDE received the M.Tech. degree from IIT Bombay, Mumbai, India, and the Ph.D. degree from Nagpur University, Nagpur, India. He is currently a Professor with the Department of Computer Science and Engineering, Visvesvaraya National Institute of Technology, Nagpur. He has 30 years of academic experience. He has coauthored a number of research articles in various journals, conferences, and book chapters. He has authored several books, including *C & Data Structure*, *Data Warehousing Using Oracle*, and *SQL/PL SQL for Oracle 11g* (Wiley). His research interests include databases, data mining, and pattern recognition. He is a member of ISTE and SAE-India. He was a recipient of prestigious awards and honors for his excellence in academics and research.