LRFU (Least Recently/Frequently Used) Replacement Policy: A Spectrum of Block Replacement Policies[†‡]

Donghee Lee[*] Jongmoo Choi[*] Jong-Hun Kim[*]
Sam H. Noh[**] Sang Lyul Min[*] Yookun Cho[*] Chong Sang Kim[*]

March 1 1996
SNU-CE-AN-96-004

[*]Dept. of Computer Engineering, Seoul Nat'l Univ., San 56-1 Shinlim-dong, Kwanak-gu, Seoul 151-742, Korea.
[**]Dept. of Computer Engineering, Hong-Ik Univ., 72-1 Sangsoo-dong, Mapo-gu, Seoul 121-791, Korea.

Abstract

The LRU (Least Recently Used) and LFU (Least Frequently Used) replacement policies are two extreme replacement policies. The LRU policy gives weight to only one reference for each block, that is, the most recent reference to the block while giving no weight to older ones representing one extreme, and the LFU gives equal weight to all references representing the other extreme. These extremes imply the existence of a spectrum between them. In this paper we propose such a spectrum which we call the LRFU (Least Recently/Frequently Used) policy. The LRFU subsumes both the LRU and LFU, and provides a spectrum of block replacement policies between them according to how much more we weigh the recent history than the older history. While most previous policies use limited history in making block replacement decisions, the LRFU policy bases its decision on all of the reference history of each block recorded during cache residency. Nevertheless, the LRFU requires only a few words for each block to maintain such history. This paper also describes an implementation of the LRFU that again subsumes the native LRU and LFU implementations. This implementation of the LRFU has a time complexity that ranges between O(1) to O($\log n$) where n is the number of blocks in the cache. Results from trace-driven simulations show that there exist points on the spectrum at which the LRFU performs better than the previously known policies for the workloads we considered.

Keyword : Buffer Cache, LFU, LRU, Replacement Policy, Trace-driven Simulation.

[‡]This paper was supported (in part) by NON DIRECTED RESEARCH FUND, Korea Research Foundation.

1 Introduction

The gap between processor and disk speed is becoming wider as VLSI technologies keep advancing at an enormous rate. Though the density growth of disks has somewhat kept pace, the access time improvement has been comparatively very slow. To overcome the resultant speed gap, caching techniques have been used in various forms so that disk blocks that are likely to be accessed in the near future are kept in DRAM memory which is referred to as the buffer cache [1]. However, as the buffer cache size is necessarily limited a prudent use of this space is critical in alleviating the aforementioned speed gap. To this end, the block replacement policy that decides which block should be replaced on a miss must judiciously select the block (or blocks) to be evicted.

The development of efficient block replacement policies has been the topic of much research in both the systems [3, 7, 9] and database areas [4, 6, 8, 10]. For a block replacement policy to be effective, it must have the following two traits. First, it should make good use of observations made from past behaviors of blocks to distinguish between the blocks that are likely to be referenced in the near future and those that are not. In general, the workload continuously changes over time and an effective replacement policy should be able to adapt itself according to the workload evolution. A good policy should be able to distinguish not only between the hot and cold blocks but also between the blocks that are hot but are getting colder and those that are cold but are getting hotter.

Second, the policy should allow for an efficient implementation in terms of both space and time complexities. For this to be realized, the space needed to maintain information about the past behavior of a block should be bounded. Furthermore, the time complexity of the buffer cache management algorithm must be kept to the minimum. Preferably, the algorithm should have an O(1) or O($\log n$) time complexity where n is the number of blocks in the buffer cache.

Previous works on block replacement policies have been derived from two rather independent avenues. One group of policies are those that base their decision on the frequency of references, and the other are policies based on the recency of references. The Least Frequently Used (LFU) policy is an example of the former while the Least Recently Used (LRU) is an example of the latter. Both of the policies have their merits as well as disadvantages.

The LFU policy keeps track of the number of references to each block, and the block selected for replacement is the block that has the least number of references. This policy is based on the presumption that the block that has been more frequently referenced in the past is more likely to be referenced in the near future. In view of the two traits for an effective replacement policy, first, although the LFU policy considers all the references in the past it

cannot distinguish between references that occurred far back in the past and the more recent ones. Thus, it does not adapt well to changing workloads. For example, the LFU policy may evict currently hot blocks instead of currently cold blocks that had been hot in the past. In terms of implementation, this policy generally uses a priority queue [11] to order the blocks according to their reference counts. The time complexity of this implementation is $O(\log n)$ where n is the number of blocks in the buffer cache.

The LRU policy replaces a block that has not been referenced for the longest time [5]. This policy makes its decision on very little information, that is, only the time of the most recent reference to each block. As a result, it cannot discriminate well between frequently and infrequently referenced blocks. One advantage of this policy, however, is that it is very adaptive to changes in reference patterns compared to other policies [8]. Another advantage is that it allows for a very efficient implementation. A single linked list that orders the blocks according to the times of their most recent references suffices to implement this policy, and this implementation has a constant time complexity.

While both the frequency based and recency based approaches have their own merits, an attempt to combine the benefits of the two had not yet been made. We make this effort in this paper and propose a spectrum of replacement policies, which we call the Least Recently/Frequently Used (LRFU) replacement policy, that inherently subsumes the LRU and LFU policies. The LRFU policy is effective as its decision on which block to evict is based on both the recency and frequency of all of the references to each block in the past. Nevertheless, the policy requires only bounded memory for each block to maintain information regarding its past behavior. Furthermore, the time complexity of the policy ranges from $O(1)$ to $O(\log n)$, where n is the number of blocks in the buffer cache. The complexity depends on how much more we weigh the recent history to the older history which is a controllable parameter in the LRFU policy.

In implementing this policy, we also take into consideration the correlated references mentioned in [8, 10] which may have considerable influence on performance. This factor is incorporated into the generic policy as another parameter that may be tuned to enhance the replacement decision.

Simulation experiments using real traces show that our policy performs better than other previously known policies. Through the experiments, we show the effects of the parameters of our policy, and also show that there is a range of values for the parameters where performance is best.

The rest of the paper is organized as follows. Section 2 surveys the related works. This section focuses on two recent papers, one by Robinson and Devarakonda [10] and the other by O'Neil *et al.* [8], that have been pivotal in this area. In Section 3, we describe the LRFU

policy in detail. Its implementation is discussed in Section 4. The correlated references are incorporated into the policy in Section 5. We compare the performance of the LRFU policy with those of previous policies in Section 6. Finally, we conclude this paper in Section 7.

2 Related works

This section surveys the studies that aim at exploiting both recency and frequency to overcome the deficits of the LRU and LFU policies. Before we introduce such studies we need to explain the concept of correlated references. In general, references to disk blocks have less locality compared to references to CPU caches or virtual memory pages [10]. However, references to a disk block still show strong short-term *locality of reference* once the disk block is referenced. Such clustered references are called correlated references and their examples in database systems are presented in [8, 10].

The concept of correlated references was first introduced in [10] by Robinson and Devarakonda. In this paper, a frequency based policy called the FBR (Frequency-based Replacement) policy is presented. The difference between the FBR and the conventional LFU is that the former replaces a block based on the frequency of non-correlated references that are obtained by making use of a special buffer called a *new section*. The major contribution of the FBR policy is in introducing the concept of correlated references and confirming the possibility of using the modified frequency as a criterion for block replacement. The simulation results given in [10] show that the FBR outperforms the LRU for the workloads considered.

In [8], O'Neil *et al.* present the LRU-K replacement policy that bases its replacement decision on the time of the K'th-to-last non-correlated reference to each block. As the LRU-K considers the last K references, it can discriminate well between frequently and infrequently referenced blocks. It can also remove cold blocks quickly since such blocks would have a wider span between the current time and the K'th-to-last reference time.

However, the LRU-K does not combine recency and frequency in a unified manner. It ignores the recency of the K-1 references, and considers only the distance of the K'th reference from the current time. This violates the rule of thumb that the more recent behavior predicts the future better. For example, assume that {1, 24, 25} and {1, 2, 25} are the reference histories of blocks a and b, respectively. Then, LRU-3 would treat both blocks equally, although intuitively, block a is more likely to be referenced in the near future since its second-to-last reference is more recent. For this reason, the LRU-K does not adapt well to evolving workloads when K is large. Also, it incurs overhead to keep the history of the last K references though a large K value may not be necessary in practice.

The LRU-K requires that all of the last K reference times of each block be maintained in order to decide a victim block. A block that does not have all of the last K reference times must be regarded as a special case. If the history of a block is not saved when the block is replaced from the buffer cache, a considerable length of time may be needed to reacquire its history, and in some cases, it may not even be able to acquire all the K reference times before it is evicted again. To cope with this problem, the LRU-K maintains the history of each block for an extended period of time after the block is removed from the buffer cache.

As previously mentioned, one advantage of the LRU-K is that it quickly removes cold blocks from the buffer cache when K is small. Johnson and Shasha propose a block replacement policy called 2Q [6] that starts from a similar motivation. In this approach, there is a special buffer called the A1 queue into which a missed block is initially placed. A block in the A1 queue is promoted to the main buffer cache only when it is re-referenced while in the A1 queue. Otherwise, it will be evicted when it becomes the LRU block in the A1 queue. This allows cold blocks to be removed quickly from the buffer cache as in the LRU-K. This is in line with the sLRU policy proposed by Karedla *et al.* [7]. The 2Q policy has an advantage over the LRU-K in that its time complexity is O(1) compared to O($\log n$) for LRU-K.

Buffer management schemes, in general, have also been extensively studied in the database arena [4] (also see the references therein). However, many of its algorithms make use of information that is deduced from query optimizer plans. Since such information is usually not available for general file caching, the applicability of these schemes is limited to database systems.

Another approach of interest is the application-controlled file caching scheme [3] where the user has control over the block replacement decisions. This certainly is a promising approach but beyond the scope of this paper.

3 The Least Recently/Frequently Used (LRFU) policy

This section describes the proposed LRFU policy. Unlike the LFU and LRU policies that consider either frequency or recency only, the LRFU policy takes into account both the frequency and recency of references in its replacement decision. Furthermore, unlike the LRU-K policy that considers only the last K references to a block this policy considers all of the past references to a block to appraise the likelihood that the block may be referenced in the near future. Nevertheless, the policy still requires only bounded memory and its implementation overhead is comparable to those of the LFU and LRU policies as we will see in the next section.

The LRFU policy associates a value with each block. This value is called the CRF (Combined Recency and Frequency) value and quantifies the likelihood that the block may be referenced in the near future. Each reference to a block in the past contributes to this value and a reference's contribution is determined by a *weighing function* $\mathcal{F}(x)$ where x is the time span from the reference in the past to the current time. For example, assuming that block b is referenced at times 1, 2, 5 and 8 and, that the current time (t_c) is 10, then the CRF value of block b at t_c, denoted by $\mathcal{C}_{t_c}(b)$, is computed as

$$\mathcal{C}_{t_c}(b) = \mathcal{F}(10-1) + \mathcal{F}(10-2) + \mathcal{F}(10-5) + \mathcal{F}(10-8) = \mathcal{F}(9) + \mathcal{F}(8) + \mathcal{F}(5) + \mathcal{F}(2).$$

In general, $\mathcal{F}(x)$ would be a decreasing function to give more weight to more recent references and, therefore, a reference's contribution to the CRF value is proportional to the recency of the reference. We define the CRF value of a block more formally as follows.

Definition 1 *Assume that the system time can be represented by an integer value by using a system clock and that at most one block may be referenced at any one time. The CRF value of a block b at time t_{base}, denoted by $\mathcal{C}_{t_{base}}(b)$, is defined as*

$$\mathcal{C}_{t_{base}}(b) = \sum_{i=1}^{k} \mathcal{F}(t_{base} - t_{b_i})$$

where $\mathcal{F}(x)$ is the weighing function and $\{t_{b_1}, t_{b_2}, \cdots, t_{b_k}\}$ are the reference times of block b and $t_{b_1} < t_{b_2} < \cdots < t_{b_k} \leq t_{base}$.

The proposed LRFU policy replaces a block whose CRF value is minimum. This policy is different from the LFU policy where every reference contributes the same value regardless of its recency. The policy also differs from the LRU policy in that not only is the most recent reference to a block considered in the replacement decision but also all the other references to the block in the past are considered as well.

Intuitively, if $\mathcal{F}(x) = 1$ for all x, then the CRF value degenerates to the reference count. Thus, the LRFU policy with $\mathcal{F}(x) = 1$ is simply the LFU policy.

Property 1 *If $\mathcal{F}(x) = c$ for all x where c is a constant, then the LRFU policy replaces the same block as the LFU policy.*

To show that the LRFU policy also subsumes the LRU policy, we give an example of $\mathcal{F}(x)$ that makes the LRFU policy replace the same block as the LRU policy. Assume that block a was most recently referenced at time t and another block b was referenced at every time step

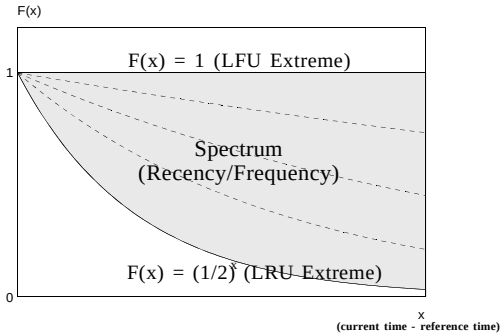


Figure 1: Spectrum of LRFU according to the function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ where x is (current_time − reference_time).

starting from time 0, but the most recent reference to it was made at time $t-1$. Then the CRF values of a and b at current time t_c are $\mathcal{C}_{t_c}(a) = \mathcal{F}(t_c - t)$ and $\mathcal{C}_{t_c}(b) = \sum_{t'=0}^{t-1} \mathcal{F}(t_c - t')$. Although block b has been referenced many more times than block a, the LRU policy will replace block b in favor of block a. For the LRFU policy to mimic this behavior, $\mathcal{C}_{t_c}(a)$ must be larger than $\mathcal{C}_{t_c}(b)$, thus $\mathcal{F}(t_c - t) > \sum_{t'=0}^{t-1} \mathcal{F}(t_c - t')$. By generalizing the above condition, we have the following.

Property 2 *If $\mathcal{F}(x)$ satisfies the following condition, then the LRFU policy replaces the same block as the LRU policy.*

$$\forall i \;\; \mathcal{F}(i) > \sum_{j=i+1}^{\infty} \mathcal{F}(j).$$

A class of functions that includes a function with property 1 and also a function with property 2 is $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ where λ ranges from 0 to 1. This class of functions is shown in Figure 1. It has a control parameter λ that allows a trade-off between recency and frequency. For example, as λ approaches 0, the LRFU policy leans towards a frequency-based policy. Eventually when λ is equal to 0 (i.e., $\mathcal{F}(x) = 1$), the LRFU policy is simply the LFU policy. On the other hand, as λ approaches 1, the LRFU policy leans towards a recency-based policy, and when λ is equal to 1 (i.e., $\mathcal{F}(x) = (\frac{1}{2})^x$), the LRFU policy degenerates to the LRU policy. (Note that $\mathcal{F}(x) = (\frac{1}{2})^x$ satisfies property 2.) The spectrum (Recency/Frequency) shown in Figure 1 is where the LRFU policy differs from both LFU and LRU.

4 Implementation of the LRFU policy

In general, computing the CRF value of a block requires the reference times of all the previous references to that block. This obviously requires unbounded memory and, thus the policy may not be implementable. Furthermore, since a reference's contribution to the CRF value changes over time, the CRF value changes as well. This necessitates recomputing the CRF value of every block at each time step, again making the policy unimplementable. We show in the following that if the weighing function $\mathcal{F}(x)$ has a certain property, the storage and computational overheads can be reduced drastically such that this policy not only becomes implementable but also efficient. We identify two such properties which are: $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ and $\mathcal{F}(x+y) = \mathcal{F}(x) + \mathcal{F}(y)$. For the remainder of this paper, we concentrate on the first case as the second case can be handled analogously to the first one.

Property 3 *If $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ for all x and y, then $\mathcal{C}_{t_{b_k}}(b)$, which is the CRF value of block b at time t_{b_k}, is derived from $\mathcal{C}_{t_{b_{k-1}}}(b)$ as follows;*

$$\mathcal{C}_{t_{b_k}}(b) = \sum_{i=1}^{k} \mathcal{F}(t_{b_k} - t_{b_i}) = \mathcal{F}(t_{b_k} - t_{b_k}) + \sum_{i=1}^{k-1} \mathcal{F}(t_{b_k} - t_{b_i}) = \mathcal{F}(0) + \sum_{i=1}^{k-1} \mathcal{F}(t_{b_k} - t_{b_i}).$$

Let δ be $t_{b_k} - t_{b_{k-1}}$.

$$\begin{aligned}
\mathcal{C}_{t_{b_k}}(b) &= \mathcal{F}(0) + \sum_{i=1}^{k-1} \mathcal{F}(t_{b_k} - t_{b_i}) = \mathcal{F}(0) + \sum_{i=1}^{k-1} \mathcal{F}(\delta + t_{b_{k-1}} - t_{b_i}) \\
&= \mathcal{F}(0) + \sum_{i=1}^{k-1} \mathcal{F}(\delta)\mathcal{F}(t_{b_{k-1}} - t_{b_i}) = \mathcal{F}(0) + \mathcal{F}(\delta) \sum_{i=1}^{k-1} \mathcal{F}(t_{b_{k-1}} - t_{b_i}) \\
&= \mathcal{F}(0) + \mathcal{F}(\delta)\mathcal{C}_{t_{b_{k-1}}}(b).
\end{aligned}$$

Property 3 shows that if $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ then the CRF value at the time of the K'th reference can be computed using the time of the (K-1)'th reference and the CRF value at that time. Likewise, $\mathcal{C}_{t_c}(b)$, which is the CRF value of block b at current time t_c, can be computed by multiplying $\mathcal{F}(\delta)$ to $\mathcal{C}_{t_{b_k}}(b)$ where $\delta = t_c - t_{b_k}$. This implies that, at any time, the CRF value can be computed using only two variables for each block, and these are all the history each block needs to maintain.

The function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ explained in the previous section has the above property. In addition to the $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ property, this function has the property that it gives more weight to more recent references, which is consistent with the principle of temporal locality. For this weighing function an intuitive meaning of λ is that a block's CRF value is halved after every $\frac{1}{\lambda}$ time steps. For example, if λ is equal to 0.0001, a block's CRF value is

halved every 10000 time steps. In the remainder of this paper, we concentrate only on the weighing function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$.

Recall that the LRFU policy replaces a block whose CRF value is minimum. Therefore, it is necessary that the blocks be ordered according to their CRF values. However, with the exception of $\mathcal{F}(x) = 1$ $(= (\frac{1}{2})^{0x})$, the CRF value of a block changes with time. This, in general, requires that the CRF value of every block be updated at each time step and that blocks be reordered according to the new CRF values again at each time step. However, such updates and reordering are not needed if $\mathcal{F}(x)$ is such that $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$. In particular, updates and reordering of blocks are needed only upon a block reference. We prove this in the following.

Property 4 *If $\mathcal{C}_t(a) > \mathcal{C}_t(b)$ and neither a nor b has been referenced after t, then $\mathcal{C}_{t'}(a) > \mathcal{C}_{t'}(b)$ for all $t' \geq t$.*

Proof. Let $\delta = t' - t$. Since $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$, $\mathcal{C}_{t'}(a) = \mathcal{F}(\delta)\mathcal{C}_t(a)$ and $\mathcal{C}_{t'}(b) = \mathcal{F}(\delta)\mathcal{C}_t(b)$. Also, since $\mathcal{F}(x) > 0$ for all x and $\mathcal{C}_t(a) > \mathcal{C}_t(b)$ we have $\mathcal{C}_{t'}(a) = \mathcal{F}(\delta)\mathcal{C}_t(a) > \mathcal{F}(\delta)\mathcal{C}_t(b) = \mathcal{C}_{t'}(b)$. □

Since the relative ordering between two blocks does not change until either of them is referenced, the reordering of blocks need only be done upon a block reference. Figure 2 gives an algorithm that is invoked when a block is referenced. The algorithm uses a heap[1] data structure to maintain the ordering of blocks according to their CRF values.

In the algorithm, H is the heap data structure, t_c is the current time and $LAST(b)$ and $CRF_{last}(b)$ are the time of the last reference to block b and its CRF value at that time, respectively. The algorithm first checks whether the requested block b is in the buffer cache. If it is, the algorithm updates its CRF value and the time of the last reference. The updated CRF value may be larger than those of b's descendants, thus violating the heap property of the sub-heap rooted by b. The algorithm uses the Restore() routine to restore the heap property of this sub-heap. Note that only the sub-heap rooted by b need be restored. This results from the fact that b's ancestors in the heap, which previously had CRF values smaller than the CRF value of b, cannot have CRF values larger than that of b after b is referenced.

In the other case where the block is not in the buffer cache, the missed block is fetched from disk and its CRF value and the time of the last reference are initialized. Using the

[1] A heap is a completely balanced binary tree that has the following property called the heap property [11].

1. it is empty, or
2. the key in the root is smaller than that in either child and both subtrees have the heap property.

```
1.  if b is already in the buffer cache
2.  then
3.      CRF_last(b) = F(0) + CRF(b)
4.      LAST(b) = t_c
5.      Restore(H, b)
6.  else
7.      fetch the missed block from the disk
8.      CRF_last(b) = F(0)
9.      LAST(b) = t_c
10.     victim = ReplaceRoot(H, b)
11.     if victim is dirty
12      then
13.         write-back the victim to the disk
14.     fi
15. fi
16. ————————————————————————————————
17. Restore(H, b)
18      if b is not a leaf node
19.     then
20.         let smaller be the child that has a smaller CRF value at the current time
21.         if CRF(b)> CRF(smaller)
22.         then
23.             swap(H, b, smaller)
24.             Restore(H, smaller)
25.         fi
26.     fi
27. end Restore
28. ————————————————————————————————
29. ReplaceRoot(H, b)
30.     victim = H.root
31.     H.root = b
32.     Restore(H, b)
33.     return victim
34. end ReplaceRoot
35. ————————————————————————————————
36. CRF(b)
37.     return F(t_c − LAST(b)) * CRF_last(b)
38. end CRF
39. ————————————————————————————————
```

Figure 2: Buffer cache management algorithm.

ReplaceRoot() routine the algorithm replaces the block that has the minimum CRF value (i.e., the one at the root of the heap) with the newly fetched block and, then, restores the heap property. If the replaced block is dirty, it is written-back to the disk. Since both the Restore() and ReplaceRoot() routines require traversing at most the height of the heap, the algorithm terminates in time $O(\log n)$.

To see how the algorithm works, consider the example given in Figure 3. In the figure, let us assume that there are 7 buffers in the buffer cache and $\mathcal{F}(x) = (\frac{1}{2})^{\frac{1}{8}x}$. We further assume that the current time $t_c = 9$. In the figure, the heap at time $t = 8$ is given on the top. In the heap, each node is denoted by a triple $(block\ number, LAST(b), CRF_{last}(b))$. Such a heap, for example, can be constructed by the following reference string: $\{(t = 0, block\ 2)$, $(t = 1, block\ 12)$, $(t = 2, block\ 11)$, $(t = 3, block\ 1)$, $(t = 4, block\ 6)$, $(t = 5, block\ 23)$, $(t = 6, block\ 1)$, $(t = 7, block\ 8)$, $(t = 8, block\ 8)\}$. Consider a reference to block 11 that is made at the current time (i.e., $t = 9$). Since the referenced block is already in the buffer cache, first, its CRF value and the time of the last reference are updated. The new CRF value is $1 + (\frac{1}{2})^{\frac{7}{8}}$ that is equal to $\mathcal{F}(0) + \mathcal{F}(t_c - LAST(b)) * CRF_{last}(b)$ where $t_c = 9$, $LAST(b) = 2$, and $CRF_{last}(b) = 1$. Then the heap property of the sub-heap rooted by this block is restored. When this restore operation is performed, the node corresponding to the currently referenced block is swapped with the node for block 23 which has a smaller CRF value among the two children of the current node.

Consider another reference that is made to block 18 at $t = 10$. Since this block is not in the buffer cache, first, the block with the minimum CRF value should be replaced to make room for the missed block. The block at the root of the heap, block 2 in this case, is such a block. Then the missed block (after it is fetched from disk) becomes the new root of the heap and the restore operation is performed on the entire heap. The figure at the bottom of Figure 3 shows the heap after this restore operation.

The $O(\log n)$ time complexity of the LRFU policy is comparable to that of the LFU policy. However, this time complexity is considerably higher than the $O(1)$ time complexity of the LRU policy, which is simply the LRFU policy with $\lambda = 1$. In the following, we show that the LRFU policy with weighing function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ also lends itself to a spectrum of implementations whose time complexity depends on the value of λ. Consider the following property.

Property 5 *In the LRFU policy with* $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$, *there exists a threshold distance* $d_{threshold}$ *such that*

$$\forall\ d \geq d_{threshold},\ \ \mathcal{F}(0) > \sum_{i=d}^{\infty} \mathcal{F}(i).$$

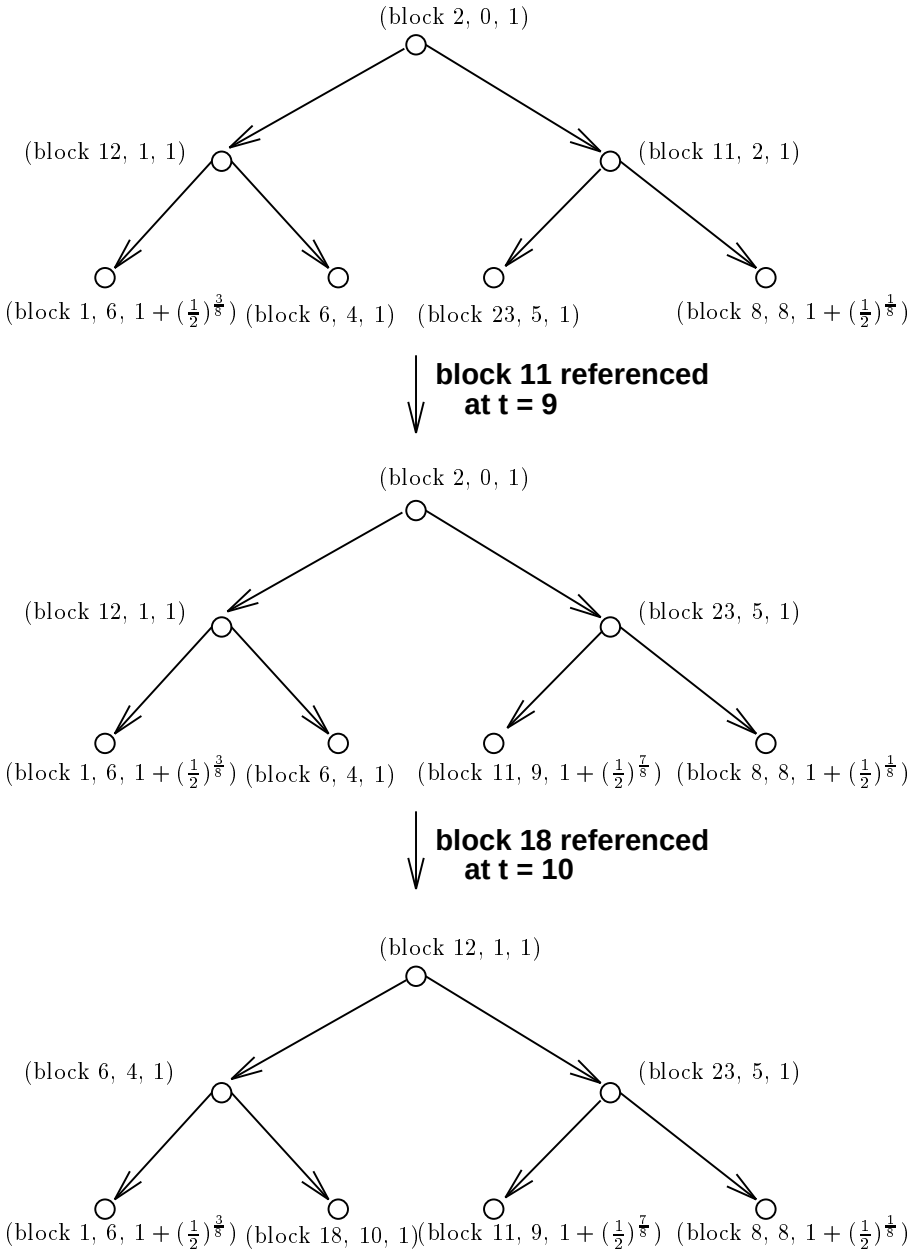


Figure 3: Buffer cache management algorithm example.

In particular, the minimum of such $d_{threshold}$ values is given by $\lceil \frac{\log_{\frac{1}{2}}(1-(\frac{1}{2})^{\lambda})}{\lambda} \rceil$.

Proof. Let d' be such a $d_{threshold}$. Then, d' should satisfy

$$\begin{aligned} \mathcal{F}(0) = 1 &> \sum_{i=d'}^{\infty} \mathcal{F}(i) \\ &= (\frac{1}{2})^{\lambda d'} + (\frac{1}{2})^{\lambda(d'+1)} + (\frac{1}{2})^{\lambda(d'+2)} + \cdots \\ &= (\frac{1}{2})^{\lambda d'}(1 + (\frac{1}{2})^{\lambda} + (\frac{1}{2})^{2\lambda} + \cdots) \\ &= (\frac{1}{2})^{\lambda d'}(\frac{1}{1-(\frac{1}{2})^{\lambda}}) \end{aligned}$$

Multiplying both sides by $1-(\frac{1}{2})^{\lambda}$ yields

$$1 - (\frac{1}{2})^{\lambda} > (\frac{1}{2})^{\lambda d'}$$

Taking $\log_{\frac{1}{2}}$ on both sides yields

$$\log_{\frac{1}{2}}(1 - (\frac{1}{2})^{\lambda}) < \lambda d'$$

Simplifying this equation then gives

$$d' \geq \lceil \frac{\log_{\frac{1}{2}}(1 - (\frac{1}{2})^{\lambda})}{\lambda} \rceil \quad \square$$

This property states that a block whose most recent reference was made earlier than $d_{threshold}$ time units ago cannot have a CRF value that is larger than $\mathcal{F}(0)$, which is the CRF value of the currently requested block. Conversely, for a block to have a CRF value larger than $\mathcal{F}(0)$, its most recent reference must have been made within $d_{threshold}$ time units. This implies that the number of blocks that have CRF values larger than $\mathcal{F}(0)$ is bounded above by $d_{threshold}$ since we assume that at most one request can be made in each time step.

In the optimized implementation of the LRFU policy which is explained in the following, we maintain $d_{threshold}$ blocks in the heap as in the LFU and the remaining blocks in a linked list as in the LRU. The blocks that are maintained in each data structure are determined such that the CRF value of any block maintained in the heap is larger than that of any block in the linked list. With these settings, the CRF value of the blocks in the linked list cannot be larger than $\mathcal{F}(0)$ since the number of blocks that have CRF values larger than $\mathcal{F}(0)$ is bounded above by $d_{threshold}$ and the number of blocks maintained in the heap is $d_{threshold}$.

The optimized LRFU implementation operates as follows. First, for the case where a requested block is not in the buffer cache, the block at the tail of the linked list is replaced

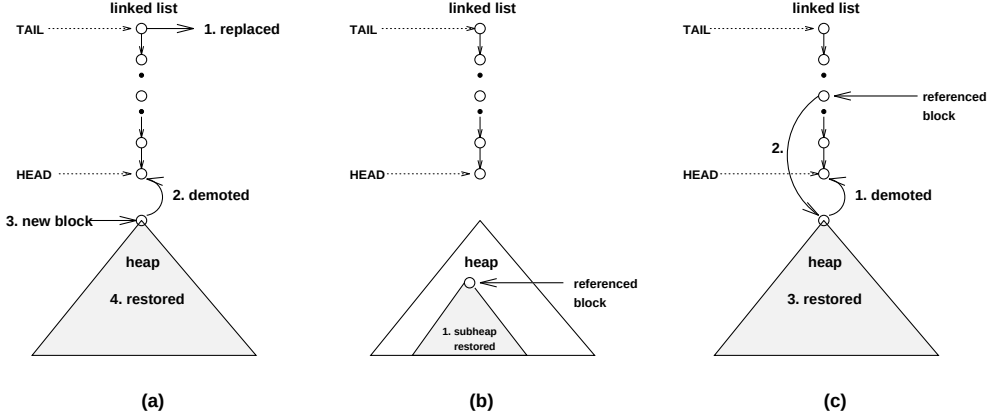


Figure 4: Optimized implementation of the LRFU policy.

and the block at the root of the heap whose CRF value now becomes smaller than $\mathcal{F}(0)$ by the passage of time is demoted to the head of the linked list (cf. Figure 4(a)). Then, the currently requested block, which has $\mathcal{F}(0)$ as its CRF value, becomes the new root of the heap and the restore operation is performed on the entire heap. These operations take only $\mathrm{O}(\log d_{threshold})$ time since the number of blocks in the heap is $d_{threshold}$. Further, the assertions that the CRF value of the blocks in the heap is larger than that of the blocks in the linked list and that the CRF value of the blocks in the linked list is smaller than $\mathcal{F}(0)$ are maintained.

The other case where the requested block is in the buffer cache can further be divided into two cases where the currently referenced block is in the heap or in the linked list. First, consider the case where the currently requested block is in the heap. Here, the restore operation needs to be performed only for the sub-heap rooted by the currently requested block (cf. Figure 4(b)). This again takes $\mathrm{O}(\log d_{threshold})$ time and the aforementioned two assertions are maintained. In the other case where the currently requested block is in the linked list, the block corresponding to the root of the heap is demoted to the head of the linked list and the currently requested block becomes the new root (cf. Figure 4(c)). Then, the restore operation is performed on the entire heap. These operations take $\mathrm{O}(\log d_{threshold})$ time as before and the aforementioned assertions are maintained. In summary, in all the cases considered, the time complexity of the optimized LRFU implementation is $O(\log d_{threshold})$.

On the LRU extreme of this optimized LRFU implementation (i.e., when $\lambda = 1$), $d_{threshold}$ equals to 1. Thus only one block need be maintained in the heap. This implies that all the blocks in the buffer cache can be maintained by a single linked list. This corresponds to the native LRU implementation and its time complexity is $\mathrm{O}(1)$. On the other hand, as we move towards the LFU extreme (i.e., when $\lambda = 0$), the number of blocks that should be

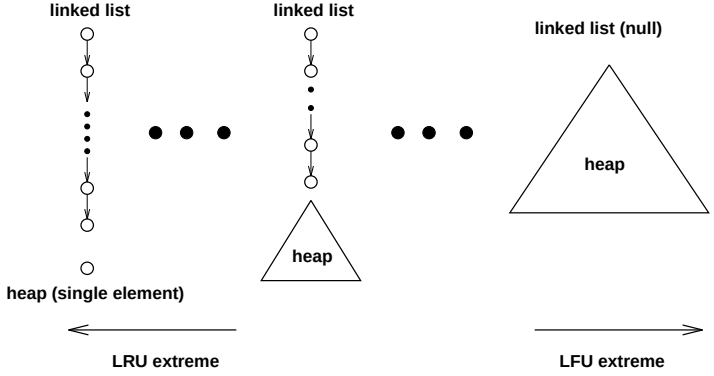


Figure 5: Spectrum of the LRFU implementations.

maintained in the heap increases. Eventually, on the LFU extreme $d_{threshold}$ is equal to ∞ and, thus, all the blocks in the buffer cache should be maintained in the heap. As a result, the time complexity becomes $O(\log n)$ where n is the number of blocks in the buffer cache. This again coincides with the time complexity and the data structure of the native LFU implementation. Figure 5 shows the spectrum of the LRFU implementations.

5 LRFU with correlated references

In this section, we describe the LRFU policy considering the correlated references. In this variation, all the references within a correlated period are treated as a single non-correlated reference. This is motivated by the observation that the recency and frequency of higher level operations such as transactions in database systems can predict the future better than the recency and frequency of lower level disk accesses [8, 10]. To incorporate the concept of correlated references more formally, we introduce a masking function $\mathcal{G}_c(x)$.

$$\mathcal{G}_c(x) = \begin{cases} 0 & : \quad x \leq c \\ 1 & : \quad x > c \end{cases}$$

where c is the correlated period that determines how far two references should be separated to be considered as not being correlated.

Incorporating the masking function $\mathcal{G}_c(x)$, the calculation of the CRF value of a block b at current time t_c, denoted by $\mathcal{C}'_{t_c}(b)$, is revised as follows:

$$\mathcal{C}'_{t_c}(b) = \mathcal{F}(t_c - t_{b_k}) + \sum_{i=1}^{k-1} \mathcal{F}(t_c - t_{b_i}) * \mathcal{G}_c(t_{b_{i+1}} - t_{b_i}).$$

However, this revision affects neither the way $\mathcal{C}_{t_{b_k}}(b)$ is calculated nor the basic structure of the buffer cache management algorithm for the weighing function of interest. We prove this in the following.

Property 6 *If $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ for all x and y, then $\mathcal{C}'_{t_{b_k}}(b)$, which is the CRF value of block b at time t_{b_k} when correlated references are considered, is derived from $\mathcal{C}'_{t_{b_{k-1}}}(b)$ as follows;*

$$\begin{aligned}
\mathcal{C}'_{t_{b_k}}(b) &= \mathcal{F}(t_{b_k} - t_{b_k}) + \sum_{i=1}^{k-1} \mathcal{F}(t_{b_k} - t_{b_i}) * \mathcal{G}_c(t_{b_{i+1}} - t_{b_i}) \\
&= \mathcal{F}(0) + \sum_{i=1}^{k-1} \mathcal{F}(t_{b_k} - t_{b_i}) * \mathcal{G}_c(t_{b_{i+1}} - t_{b_i})
\end{aligned}$$

Let δ be $t_{b_k} - t_{b_{k-1}}$.

$$\begin{aligned}
\mathcal{C}'_{t_{b_k}}(b) &= \mathcal{F}(0) + \sum_{i=1}^{k-1} \mathcal{F}(\delta + t_{b_{k-1}} - t_{b_i}) * \mathcal{G}_c(t_{b_{i+1}} - t_{b_i}) \\
&= \mathcal{F}(0) + \sum_{i=1}^{k-1} \mathcal{F}(\delta) * \mathcal{F}(t_{b_{k-1}} - t_{b_i}) * \mathcal{G}_c(t_{b_{i+1}} - t_{b_i}) \\
&= \mathcal{F}(0) + \mathcal{F}(\delta) * \sum_{i=1}^{k-1} \mathcal{F}(t_{b_{k-1}} - t_{b_i}) * \mathcal{G}_c(t_{b_{i+1}} - t_{b_i}) \\
&= \mathcal{F}(0) + \mathcal{F}(\delta) * [\mathcal{F}(t_{b_{k-1}} - t_{b_{k-1}}) * \mathcal{G}_c(t_{b_k} - t_{b_{k-1}}) + \sum_{i=1}^{k-2} \mathcal{F}(t_{b_{k-1}} - t_{b_i}) * \mathcal{G}_c(t_{b_{i+1}} - t_{b_i})] \\
&= \mathcal{F}(0) + \mathcal{F}(\delta) * [\mathcal{F}(0) * \mathcal{G}_c(t_{b_k} - t_{b_{k-1}}) + \sum_{i=1}^{k-2} \mathcal{F}(t_{b_{k-1}} - t_{b_i}) * \mathcal{G}_c(t_{b_{i+1}} - t_{b_i})] \\
&= \mathcal{F}(0) + \mathcal{F}(\delta) * [\mathcal{F}(0) * \mathcal{G}_c(t_{b_k} - t_{b_{k-1}}) + \mathcal{C}'_{t_{b_{k-1}}}(b) - \mathcal{F}(t_{b_{k-1}} - t_{b_{k-1}})] \\
&= \mathcal{F}(0) + \mathcal{F}(\delta) * [\mathcal{F}(0) * \mathcal{G}_c(\delta) + \mathcal{C}'_{t_{b_{k-1}}}(b) - \mathcal{F}(0)]
\end{aligned}$$

6 Experimental results

In this section we discuss the results obtained from a trace-driven simulation. We chose two different types of real workload traces. Specifically, one is the Sprite network file system trace [2] representing file system activities, and the other is database traces that consists of the DB2 trace used in [6] and the OLTP trace used in both [6] and [8].

The Sprite trace contains two days worth of requests to a file server from application programs running on client workstations. Of the workstation clients, we selected three clients with the most requests (client workstations 54, 53 and 48) and simulated the buffer caches of these client workstations. Client 54 made 203,808 references to 4,822 unique blocks, client 53 made

141,223 references to 19,990 unique blocks, and client 48 made 133,996 references to 7,075 unique blocks.

To reiterate the descriptions of the database traces given in [6], the DB2 trace comes from a commercial installation of DB2 and contains 500,000 references to 75,514 unique blocks. The OLTP trace is a one hour block reference trace to a CODASYL database. This trace consists of 914,145 references to 186,880 unique blocks. We note that the traces are those used in the previous papers [6, 8], and were obtained from the authors of these papers.

Comparisons are made with the LRU-2 and 2Q policies which were implemented according to the descriptions in [6, 8]. Our results also allow comparison with the FBR policy as this policy is equivalent to the LRFU policy considering the correlated references at the LFU extreme.

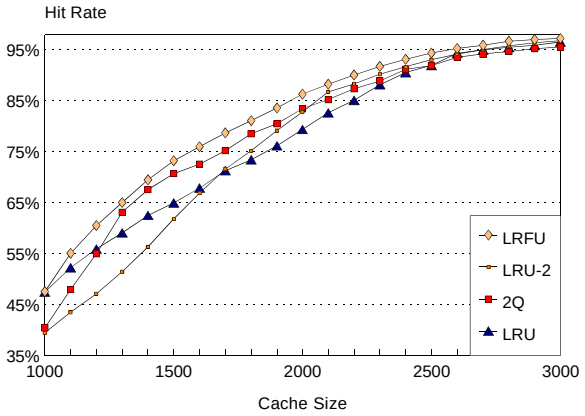
6.1 Comparison of the LRFU policy with other policies

Figures 6 and 7 show the hit rates of the LRFU policy as a function of the cache size for the Sprite and database traces, respectively. The hit rates are compared with those of previously proposed policies, namely, the LRU, the LRU-2, and the 2Q policies. (Note that the scales are different for each of the figures.)

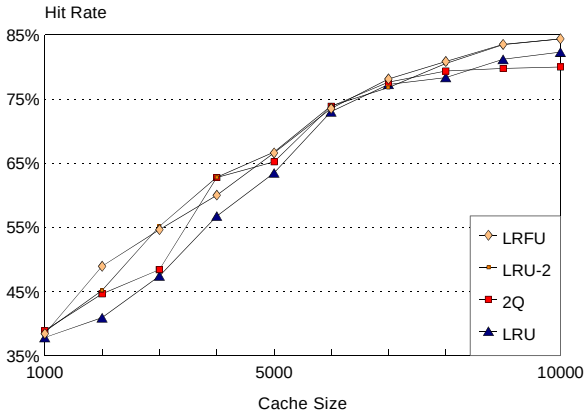
In the simulation, we used the weighing function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ explained in Section 4. We also treated correlated references as a single non-correlated reference using the masking function $\mathcal{G}_c(x)$, where c is the correlated period. The results of the LRU-2 and 2Q policies were obtained when their correlation periods are either 20% or 30% of the cache size as suggested in [6], and the better results were selected for each policy. Similarly we used the values of λ and c that give the best performance for the LRFU policy. The effect of these parameters on the performance of the LRFU policy will be discussed later in this section.

Some general observations can be made. For most cases, the LRU policy performs the worst. This is the same observation made in previous works [6, 8]. However, we can see that the LRU policy performs reasonably well when the cache size is large. Also, for most cases, the LRFU policy has the highest hit rates, while the LRU-2 and 2Q policies show similar performance, giving and taking at particular cache sizes. The 2Q policy performs rather strongly when the cache size is small, occasionally performing better than the LRFU policy. (The reason behind this is explained below.) However, its hit rate starts to converge earlier, that is, at a smaller cache size, than other policies.

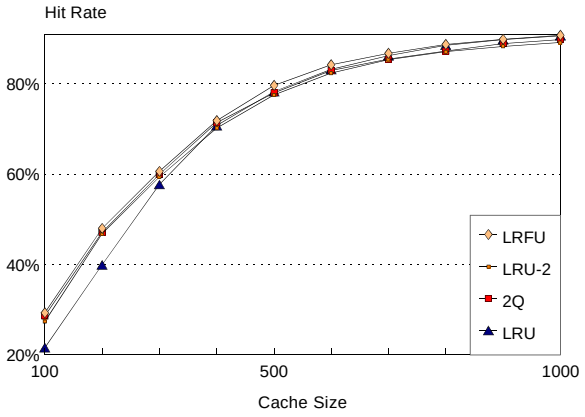
Though the LRFU policy performs best for most cases, when we take a closer look at Figures 6(b) (client 53 in the Sprite trace) and 7(a) (DB2 trace) we notice that when the



(a) **Client 54 in the Sprite trace**

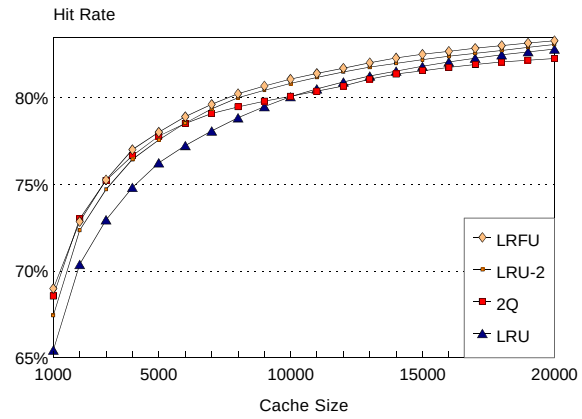


(b) **Client 53 in the Sprite trace**

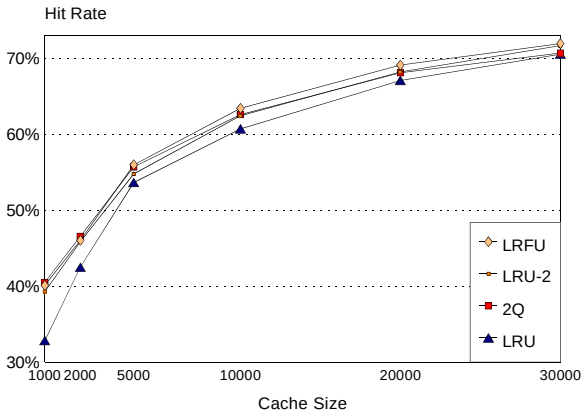


(c) **Client 48 in the Sprite trace**

Figure 6: Comparison of LRFU with other policies using the Sprite trace.



(a) **DB2**



(b) **OLTP**

Figure 7: Comparison of LRFU with other policies using the database trace.

Table 1: Comparison of hit rates when history is kept for LRFU when the associated block is evicted like the LRU-2 and 2Q policies.

(a) **Client 53 in the Sprite trace**

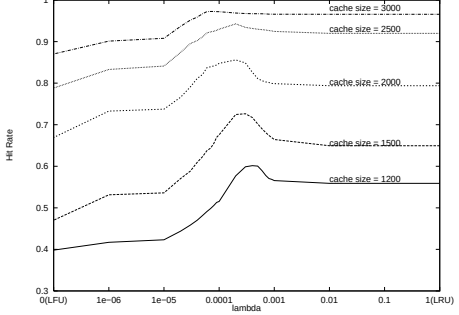
Cache Size	LRU	LRU2	2Q	LRFU
1000	0.3782	0.3872	0.3892	0.3908
2000	0.4091	0.4516	0.4461	0.4511
3000	0.4740	0.5512	0.4840	0.5753
4000	0.5672	0.6282	0.6277	0.6328
5000	0.6349	0.6674	0.6525	0.6935
6000	0.7303	0.7395	0.7387	0.7561
7000	0.7730	0.7681	0.7765	0.7872
8000	0.7836	0.8056	0.7938	0.8190
9000	0.8120	0.8347	0.7977	0.8380
10000	0.8232	0.8436	0.8002	0.8443

(b) **DB2**

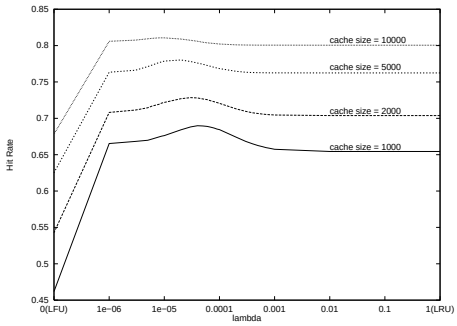
Cache Size	LRU	LRU-2	2Q	LRFU
1000	0.6544	0.6744	0.6856	0.6923
2000	0.7038	0.7235	0.7301	0.7333
3000	0.7295	0.7472	0.7524	0.7574
4000	0.7483	0.7645	0.7667	0.7720
5000	0.7625	0.7758	0.7780	0.7844
6000	0.7725	0.7854	0.7852	0.7916
7000	0.7809	0.7936	0.7909	0.7993
8000	0.7885	0.8001	0.7948	0.8046
9000	0.7949	0.8043	0.7981	0.8089
10000	0.8006	0.8082	0.8008	0.8122

cache size is small the LRFU policy is only comparable to the LRU-2 and 2Q policies, performing even worse than these policies in some cases. The reason behind this is that in the LRU-2 and 2Q policies blocks keep their history of references even after they are evicted from the buffer cache [6]. Thus, when the block is brought back into the buffer cache it starts with a good knowledge of its past behavior. However, in our simulations, we chose not to allow this for the LRFU policy as this is a better representation of the real world.

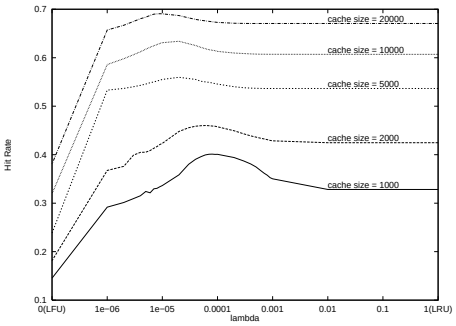
However, after making this initial observation we modified our simulation program in such a way that the blocks keep their past history (i.e., $LAST(b)$ and $CRF_{last}(b)$) even after they are evicted like the LRU-2 and 2Q policies. For this case, the LRFU policy surpasses the LRU-2 and 2Q policies even for small cache sizes for client 53 in the Sprite trace and the DB2 trace as can be seen in TABLE 1.



(a) **Client 54 in the Sprite trace**



(b) **DB2**



(c) **OLTP**

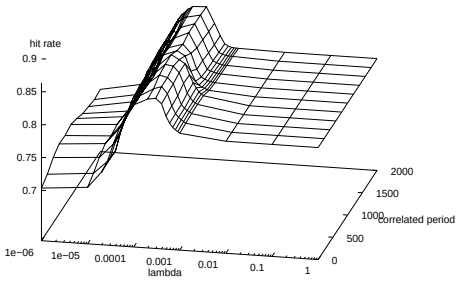
Figure 8: Effects of λ on the LRFU policy using Sprite and database traces.

6.2 Effects of λ on the performance of the LRFU policy

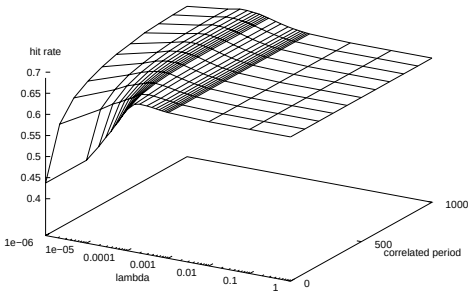
Figure 8 shows the influence of λ on the hit rate for various cache sizes assuming a fixed correlated period. All the figures in Figure 8 have similar shapes. The hit rate initially increases as the λ value increases, that is, the policy moves from the LFU extreme to the LRU extreme. After reaching a peak point, the hit rate drops slightly and then remains fairly stable decreasing very slowly until λ reaches 1. We also observe from the figures that as the cache size increases the peak hit rate is reached at a smaller λ value. This implies that as the cache size increases more weight must be given to older references, and that deciding the block to be evicted must not be made in a near-sighted manner. Based on this, appropriate λ values may be deduced as system configurations evolve.

6.3 Combined effects of λ and correlated period on the LRFU policy

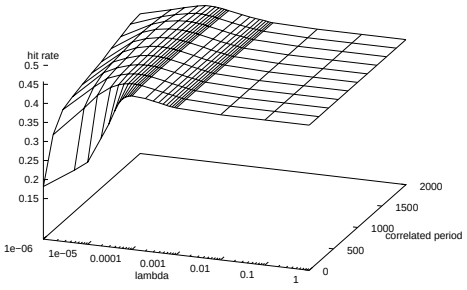
Figures 9(a), (b), and (c) show the hit rate, as a function of λ and c, for client workstation 54 with a cache size of 2000, for the DB2 trace with a cache size of 1000, for the OLTP trace



(a) **Client 54 in the Sprite trace** (cache size = 2000)



(b) **DB2** (cache size = 1000)



(c) **OLTP** (cache size = 2000)

Figure 9: Combined effects of λ and c on the LRFU policy.

with a cache size of 2000, respectively. Overall, for all correlated period values, we observe a performance effect of λ that is similar to the one shown in Figure 8, i.e., the hit rate initially increases, reaches a peak, and drops slightly after the peak.

Another observation is that the λ value giving the peak hit rate decreases for longer correlated periods. This is because as the correlated period increases, the access trend of higher level operations becomes more evident. Thus, giving more weight to the frequency of these higher level operations turns out to be beneficial to achieving higher hit rates.

Finally, we observe that the effect of the correlated period is significant when the LRFU policy leans towards the LFU policy. This is especially notable for the database traces. However, the correlated period has very little effect as the spectrum of policies moves to the LRU extreme. This observation agrees with, and indirectly explains the reason behind the improvement brought in by the FBR policy [10]. An interesting observation of the FBR policy was that there is a need for a new section to factor out locality. This notion is basically

the notion of a correlated period. We notice from our results that when λ is close to 0, that is, when the policy resides on the LFU extreme, the hit rate is greatly influenced by the correlated period. Hence, the FBR policy benefited from the addition of the new section.

7 Conclusion

In this paper, we have introduced the LRFU (Least Recently/Frequently Used) block replacement policy. While subsuming the well known LRU and LFU policies, the LRFU policy presents a spectrum of policies using a weighing function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ where λ is a controllable parameter. The λ value determines the weight given to recent and old history thereby providing a ground for an optimal combination of the effects of recency and frequency.

Unlike previous policies which consider only a limited reference history in their replacement decision, the LRFU policy makes use of all of the reference history of each block. We showed that this can be achieved with bounded memory when the weighing function $\mathcal{F}(x)$ meets a certain property such as $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$. We also showed that the proposed replacement policy lends itself to an efficient implementation whose time complexity ranges anywhere from O(1) to O($\log n$) depending on the value of λ, where n is the number of blocks in the buffer cache. This corresponds to the complexities of the native implementations of the LRU and LFU policies. We also considered the issue of correlated references within the proposed LRFU framework. This issue was incorporated into our policy by introducing a masking function $\mathcal{G}_c(x)$, where c is the correlated period, that is, the period within which references are considered to be correlated. We showed that including this issue does not alter our general framework.

Results from trace-driven simulation showed that the LRFU policy performs better than the LRU, the LRU-2, and the 2Q policies for the workloads considered. The effects of the controllable parameters λ and c for various cache sizes were discussed as well. General trends between the hit rate and λ and c values were observed for the workloads considered.

In this paper, we concentrated on the development of the framework behind the LRFU policy, focusing on the combination of two orthogonal aspects of memory references, that is, recency and frequency of references. As our results have shown, even our simple approach to combining the two has brought increased performance. We believe that incorporating other issues such as sequentiality will bring about further improvement. Other issues of interest are in finding weighing functions other than $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ that will bring about further improvement in performance. Applying our concept of combining recency and frequency in page and data placement and migration in distributed systems where there is a hierarchy of

buffer caches is also a direction for further research.

References

[1] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.

[3] P. Cao, E. W. Felten, and K. Li. Application-Controlled File Caching Policies. In *Proceedings of the Summer 1994 USENIX Conference*, pages 171–182, 1994.

[4] C. Faloutsos, R. Ng, and T. Sellis. Flexible and Adaptable Buffer Management Techniques for Database Management Systems. *IEEE Transactions on Computers*, 44(4):546–560, 1995.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[6] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450, 1994.

[7] R. Karedla, J. S. Love, and B. G. Wherry. Caching Strategies to Improve Disk System Performance. *IEEE Computer*, 27(3):38–46, March 1994.

[8] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 297–306, 1993.

[9] V. Phalke and B. Gopinath. An Inter-Reference Gap Model for Temporal Locality in Program Behavior. In *Proceedings of the 1995 ACM SIGMETRICS/PERFORMANCE Conference*, pages 291–300, 1995.

[10] J. T. Robinson and N. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference*, pages 134–142, 1990.

[11] J. D. Smith. *Design and Analysis of Algorithms.* PWS-KENT Publishing Company, Boston, MA, 1989.