On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies

Donghee Lee[†*] Jongmoo Choi[†] Jong-Hun Kim[†]
Sam H. Noh[§] Sang Lyul Min[†] Yookun Cho[††] Chong Sang Kim[†]
[†]Department of Computer Engineering
Seoul National University
Seoul 151-742, Korea
http://ssrnet.snu.ac.kr
http://archi.snu.ac.kr

[§]Department of Computer Engineering
Hong-Ik University
Mapo-Gu Sangsoo-Dong 72-1
Seoul 121-791, Korea
http://www.cs.hongik.ac.kr/~noh

Abstract

We show that there exists a spectrum of block replacement policies that subsumes both the Least Recently Used (LRU) and the Least Frequently Used (LFU) policies. The spectrum is formed according to how much more weight we give to the recent history than to the older history, and is referred to as the LRFU (Least Recently/Frequently Used) policy. Unlike many previous policies that use limited history to make block replacement decisions, the LRFU policy uses the complete reference history of blocks recorded during their 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 LRU and LFU implementations. The LRFU policy is applied to buffer caching, and results from trace-driven simulations show that the LRFU performs better than previously known policies for the workloads we considered. This point is reinforced by results from our integration of the LRFU into the FreeBSD operating system.

1 Introduction

The Least Recently Used (LRU) and the Least Frequently Used (LFU) block replacement policies are popular cache block replacement policies due to their simplicity and efficiency. In this paper, we show that between these seemingly independent two policies, there exists a spectrum of policies that subsumes these two policies. This spectrum of policies, which we refer to as the Least Recently/Frequently Used (LRFU) policy, inherits the benefits of the two policies resulting in a policy that is superior to both as well as other policies that have been previously suggested [1, 2, 3]. In the remainder of this section, we first give the motivation behind the development of the LRFU policy in the buffer caching realm. In so doing, we discuss some qualities that merit the LRFU policy over other policies. Then, we review some of the previous policies that have been proposed for buffer caching.

1.1 Motivation

The study of cache block replacement policies is, in essence, a study of the characteristics or behavior of workloads to a system. Specifically, it is a study of access patterns to blocks within the cache. Based on the recognition of access patterns through acquisition and analysis of past behavior or history, replacement policies resolve to identify the block that will be used furthest down in the future, so that that block may be replaced when needed. The LRU policy does this by attaining the recency of block references while the LFU policy considers the frequency of block references. These respective policies are inherently assuming that future behavior of the workload will be dominated by the recency or frequency factors of past behavior. Similarly, most previously proposed policies can also be placed in either category differing only in aspects of how much history to use and how to use this information [1, 2, 3]. Furthermore, for each policy, the rules for acquiring and analyzing the history are fixed for all workloads and system configurations. We discuss in the following why this is not sufficient, and that for various stages of system activity and configurations the acquisition of history and its analysis must also adequately adapt.

Consider the graphs in Fig. 1. These graphs show the influence of the recency and frequency factors of past references on the likelihood of future references. The x-axis is the recency factor and the y-axis the frequency factor, and the shades of each graph shows the probability of being re-referenced (the darker the shades, the higher the probability) for blocks with a particular recency and frequency value. The recency factor refers to the time units passed since its last reference, while the frequency is the number of references made to the block since its inclusion to the cache. For example, consider in Fig. 1(a), a block with frequency count of 10 and which was referenced 50 time units in the past (denoted by the 'x' mark). The graph shows that this block has roughly a 25% chance of being re-referenced before being replaced.

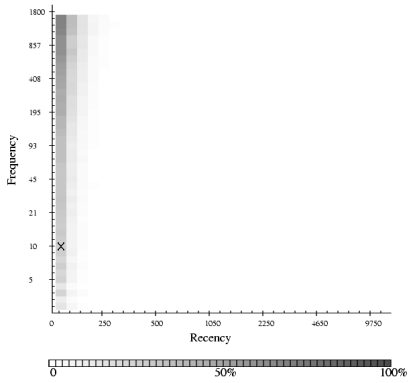
All of these graphs were obtained based on the off-line optimal algorithm for various cache sizes using a real-life trace, specifically the DB2 trace [3]. Details regarding this trace are given in Section 4.

Note from the graphs in the figure that as the cache size changes the shading pattern gradually changes as well. When the cache size is small, the darker shades run along

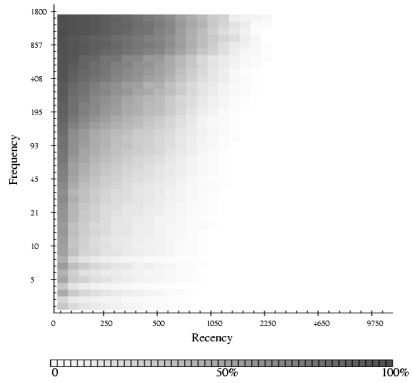
*Currently with the Dept. of Information Engineering, Cheju National University, Korea.

†The author wishes to acknowledge the financial support of the Korea Research Foundation made in the program year of 1998.

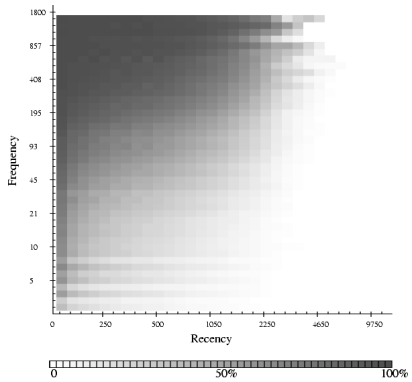
To appear in the Proceedings of the 1999 ACM SIGMETRICS Conference, May 2-4, 1999, Atlanta, Georgia



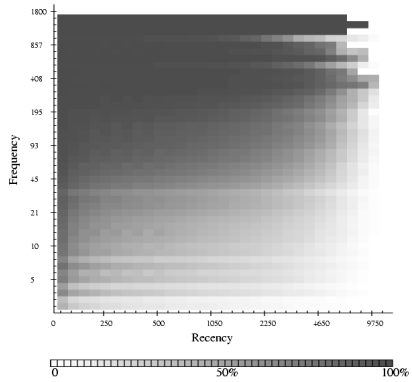
(a) **Cache Size 20**



(b) **Cache Size 100**



(c) **Cache Size 200**



(d) **Cache Size 500**

Figure 1: Change in the influence of recency and frequency factors on the probability of re-reference as cache size changes.

the vertical y-axis meaning that the recency aspect of past behavior is much more important than the frequency aspect, hence policies such as the LRU or some variant of it should be used. However, as the cache size becomes large, much of the darker shadings (that is, higher probability of re-reference) are on the top of the graphs running along the horizontal x-axis. This means that the frequency aspect is now more important than the recency aspect, hence appropriate policies reflecting this change must be used to hold in the cache those blocks that are more likely to be re-referenced.

This observation tells us that the best policy would be that which adequately incorporates the recency and frequency aspects according to the system configuration and workload. The LRFU is a policy that does so. The LRFU policy represents a spectrum of policies that lies between the LRU and LFU policies, subsuming both the LRU and LFU policies. Each policy within this spectrum represents a balance between the recency and frequency factors of past behavior, at one end considering only the recency aspect and at the other end considering only the frequency aspect. Furthermore, the LRFU policy uses the complete reference history of blocks recorded during their cache residency. Nevertheless, the policy requires only a few words for each block to maintain information about its past references. Its implementation has a time complexity that ranges from O(1) to $O(\log_2 n)$ and again subsumes the LRU and LFU implementations. The exact time complexity depends on how much more we weigh the recency over the frequency factor, which is controlled by a parameter.

Before going into the details of the LRFU policy, a review of related works is given in the next subsection.

1.2 Related works

This subsection surveys the studies that aim to overcome the limitations of the LRU and LFU policies. Our discussion focuses on two recent papers, one by Robinson and Devarakonda [2] and the other by O'Neil *et al.* [1].

A frequency-based policy called the FBR (Frequency-based Replacement) was proposed by Robinson and Devarakonda [2]. The difference between the FBR and the conventional LFU is that the former replaces blocks based on the frequency of references whose short-term locality has

been *factored out* via a special buffer called a *new section* [2]. The new section consists of k most recently referenced blocks where k is implementation dependent. When there is a hit to a block in the new section, the corresponding reference is considered to be correlated to a previous reference to the same block and the reference count of the block is not incremented. This is motivated by the observation that the reference count that increments on every reference can be misleading and the modified reference count is a more accurate indicator of the probability that the block will be re-referenced in the near future. Through simulation studies, it is shown that the FBR outperforms the LRU for the workloads that were considered [2].

O'Neil *et al.* present the LRU-K replacement policy that bases its replacement decision on the time of the Kth-to-last reference to the block [1]. In other words, its replacement decision is based on the reference density [4] observed during the past K references. Thus, when K is large, it can discriminate well between frequently and infrequently referenced blocks. On the other hand, when K is small, it can remove cold blocks quickly since such blocks would have a wider span between the current time and the Kth-to-last reference time.

However, the LRU-K ignores the recency of the $K-1$ references, and considers only the distance of the Kth reference. This violates the rule of thumb that the more recent behavior predicts the future better. For example, assume that {7, 31, 35} and {7, 9, 25} are the reference histories of blocks a and b, respectively. Then, LRU-3 would treat both blocks equally since their third-to-last reference times are the same (that is, 7) although, intuitively, block a is more likely to be referenced in the near future since its last and second-to-last references are more recent. For this reason, the LRU-K is not very adaptive to changing workloads when K is large. Also, it incurs an O(K) space overhead to keep the history of the last K references, though it is noted that a large K value may not be necessary in practice [1]. Further, since the LRU-K requires that all of the last K reference times of each block be maintained, blocks that have not acquired all its K reference history must be handled as special cases. 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 be replaced again before acquiring all the K reference times. To cope with this problem, the LRU-K maintains the history of a block for an extended period of time after the block is replaced from the buffer cache.

As previously mentioned, one advantage of the LRU-K is that it can quickly remove cold blocks from the buffer cache when K is small. Johnson and Shasha propose a block replacement policy called 2Q starting from a similar motivation [3]. In this approach, a missed block is initially placed in a special buffer called the A1 queue. 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 is replaced 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. The time complexity of the 2Q policy is O(1), which is significantly lower than the O($\log_2 n$) time complexity of the LRU-K policy.

Buffer management schemes have also been extensively studied in the database arena [5] (also see the references therein). However, many of its algorithms make use of information deduced from query optimizer plans. Another similar approach that exploits external information is the application-controlled file caching scheme [6]. These schemes are promising approaches but are beyond the scope of this paper.

1.3 The Remainder of the Paper

The remainder of this paper is organized as follows. In Section 2, we describe the LRFU policy in detail. Its implementation is discussed in Section 3. In Section 4, we compare the performance of the LRFU policy with those of previous policies through trace-driven simulations. In Section 5, we discuss a couple of practical issues, namely its actual deployment in a real operating system and the extension of the policy where the parameter of the LRFU is changed periodically according to workload evolution. Finally, we conclude this paper in Section 6.

2 The Least Recently/Frequently Used (LRFU) policy

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 will 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, assume that block b was referenced at times 1, 2, 5, and 8 and that the current time (t_c) is 10. Then, its CRF value at t_c, denoted by $\mathcal{C}_{t_c}(b)$, is computed as

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

$\mathcal{F}(x)$ essentially reflects the influence of the recency and frequency factors of a block's history in projecting the likelihood of it being re-referenced. In general, $\mathcal{F}(x)$ is a decreasing function to give more weight to more recent references. 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 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 a 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 the block with the minimum CRF value. This policy differs from the LFU policy in that the contribution of each reference is not always the same but depends on its recency. The policy also differs from the LRU policy in that it considers not only the most recent reference but also possibly all the other references in the past.

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.*

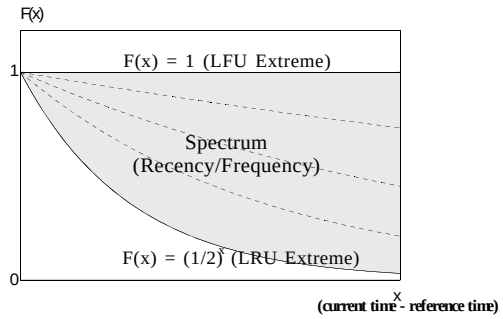


Figure 2: Spectrum of LRFU according to the function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ where x is (current_time − reference_time).

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 that another block b was referenced at every time step starting from time 0 but its most recent reference was made at time $t-1$. The LRU policy will replace block b in favor of block a although block b has been referenced many more times than block a. For the LRFU policy to mimic this behavior, the CRF value of a must be larger than that of b at current time t_c, i.e., $\mathcal{C}_{t_c}(a) = \mathcal{F}(t_c - t) > \mathcal{C}_{t_c}(b) = \sum_{t'=0}^{t-1} \mathcal{F}(t_c - t')$. 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.*

$$\forall i \;\; \mathcal{F}(i) > \sum_{j=i+1}^{k} \mathcal{F}(j) \qquad \textit{for any } k \textit{ where } k \geq i+1$$

A class of functions that satisfy both Property 1 and Property 2 is $\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$ where $p \geq 2$ and λ ranges from 0 to 1. This class of functions where $p = 2$ is shown in Fig. 2. An intuitive meaning of λ in this function is that a block's CRF value is reduced to $\frac{1}{p}$ of the original value after every $\frac{1}{\lambda}$ time steps. For example, if λ is 0.0001, a block's CRF value is reduced to $\frac{1}{p}$ after every 10000 time steps. This control parameter λ allows a trade-off between recency and frequency. In addition, this function has the property that it gives more weight to more recent references, which is consistent with the principle of temporal locality. As λ approaches 0, the LRFU policy moves 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 moves towards a recency-based policy, and when λ is equal to 1 (i.e., $\mathcal{F}(x) = (\frac{1}{p})^x$ for $p \geq 2$), the LRFU policy degenerates to the LRU policy. (Note that $\mathcal{F}(x) = (\frac{1}{p})^x$ for $p \geq 2$ satisfies Property 2.) The spectrum (Recency/Frequency) in Fig. 2 is where the LRFU policy differs from both LFU and LRU, assuming $p = 2$.

3 Implementation of the LRFU policy

From the description of the LRFU policy in the previous section, one can observe that all history of a block is retained and that the CRF values must constantly be updated. As is, the LRFU policy is unimplementable, and for the LRFU policy to be of any practical value these issues must be rectified.

3.1 Maintaining all reference history

In general, computing the CRF value of a block requires that the reference times of all the past references to that block be maintained. This obviously requires unbounded memory and thus, makes the policy unimplementable. We show in the following that if the weighing function $\mathcal{F}(x)$ has either the $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ or $\mathcal{F}(x+y) = \mathcal{F}(x) + \mathcal{F}(y)$ properties, the storage and computational overheads can be reduced drastically such that this policy becomes not only implementable but also efficient. For the remainder of the paper, we concentrate on the first property as the second 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 $\mathcal{C}_{t_{b_k}}(b)$ is derived from $\mathcal{C}_{t_{b_{k-1}}}(b)$ as follows:*

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

Let δ be $t_{b_k} - t_{b_{k-1}}$.

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

Property 3 states that if $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ then the CRF value at the time of the Kth reference can be computed from the time of the $(K-1)$th reference and the CRF value at that time. Similar derivation shows that $\mathcal{C}_{t_c}(b)$, which is the CRF value of block b at current time t_c, can be computed by multiplying $\mathcal{F}(\delta)$ and $\mathcal{C}_{t_{b_k}}(b)$ where $\delta = t_c - t_{b_k}$. This shows that, at any time, the CRF value can be computed using only two variables for each block, and these are all the history the block needs to maintain. Note that the function $\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$ for $p \geq 2$ explained in the previous section has the $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ property.

3.2 Keeping the CRF values in order

As the LRFU policy replaces the block with the minimum CRF value, it is necessary that the blocks be ordered according to their CRF values. Generally, however, a reference's contribution to the CRF values changes over time, hence, the CRF value of a block changes with time as well. This 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. Fortunately, with

```
1.    if b is already in the buffer cache
2.    then
3.        CRF_last(b) = F(0) + F(t_c − LAST(b)) * CRF_last(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 the
20(cont).             smaller CRF value at the current time
21.           if F(t_c − LAST(b)) * CRF_last(b)
21(cont).             > F(t_c − LAST(smaller)) * CRF_last(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. ————————————————————————————————
```

Figure 3: Buffer cache management algorithm.

$\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$ for $p \geq 2$, the relative ordering between two blocks does not change until either of them is referenced, hence reordering of blocks is needed only upon a block reference. We prove this in the following.

Property 4 *With $\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$ for $p \geq 2$, 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 $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$, we have $\mathcal{C}_{t'}(a) = \mathcal{F}(\delta)\mathcal{C}_t(a)$ and $\mathcal{C}_{t'}(b) = \mathcal{F}(\delta)\mathcal{C}_t(b)$. Also, since $\mathcal{F}(x) > 0$ for all x and $\mathcal{C}_t(a) > \mathcal{C}_t(b)$, the following inequality holds $\mathcal{C}_{t'}(a) = \mathcal{F}(\delta)\mathcal{C}_t(a) > \mathcal{F}(\delta)\mathcal{C}_t(b) = \mathcal{C}_{t'}(b)$. □

We have presented, thus far, $\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$ for any $p \geq 2$, to be an adequate weighing function for the LRFU policy. For the remainder of this paper, we concentrate only on the weighing function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ as its range covering both the LFU and LRU is between 0 and 1, which is common in other studies that involve control parameters.

3.3 The Algorithm

The algorithm that is invoked upon a block reference is given in Fig. 3.

Like many other replacement algorithms that base their decision on the ordering of blocks by a given criterion, the LRFU uses the heap data structure to maintain the ordering of blocks according to their CRF values (the root has the smallest CRF value). In the algorithm in Fig. 3, 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 recalculates its CRF value, updates the time of the last reference, and, if needed, restores the heap property of the sub-heap rooted by b. 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. Then, the root block of the heap is replaced with the newly fetched block and the heap property is restored. In addition, if the replaced block is dirty, it is written-back to the disk.

As in other replacement algorithms that use the heap data structure, in the LRFU the maximum number of *swap* operations is equal to the height of the heap minus one, i.e., $\lceil \log_2(n+1) \rceil - 1$. The only additional overhead of the LRFU over other policies is due to the invocations of $\mathcal{F}(x)$ when CRF values are compared, the maximum number of which is bounded above by $2 \times (\lceil \log_2(n+1) \rceil - 1)$ since $\mathcal{F}(x)$ is invoked twice at each level of the heap.

3.4 Optimized Implementation of the LRFU policy

The $O(\log_2 n)$ 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 $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ also lends itself to a spectrum of implementations whose time complexities depend on the value of λ. In this implementation spectrum, the points corresponding to the LFU and the LRU have $O(\log_2 n)$ and $O(1)$ time complexities, respectively, which are equal to the time complexities of their native implementations.

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

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

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

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

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

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

Taking $\log_{\frac{1}{2}}$ on both sides, we have

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

Finally, simplifying this equation then gives

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

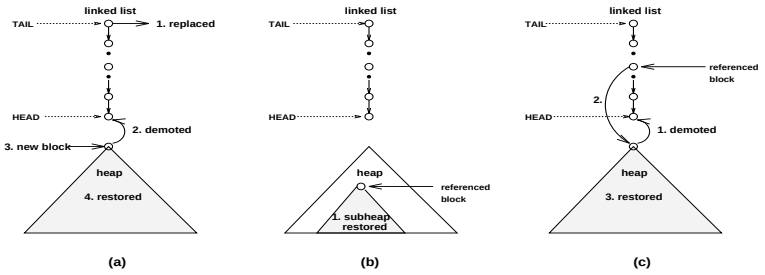


Figure 4: Optimized implementation of the LRFU policy.

This property states that a block whose most recent reference was made earlier than $d_{threshold}$ time units ago cannot have a CRF value larger than $\mathcal{F}(0)$, which is the CRF value of the currently referenced 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 states that the number of blocks that have CRF values larger than $\mathcal{F}(0)$ is bounded above by $d_{threshold}$. Hence, we maintain $d_{threshold}$ blocks in the heap and the remaining blocks in a linked list such that any block maintained in the heap has a larger CRF value than that of any block in the linked list. With this setting, the CRF value of the blocks in the linked list cannot be larger than $\mathcal{F}(0)$ since the number of blocks that can 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. When the requested block is not in the buffer cache, the block at the tail of the linked list is replaced and the block at the root of the heap is demoted to the head of the linked list (cf. Fig. 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 heap with time complexity $O(\log_2 d_{threshold})$. 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 depending on whether 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. Fig. 4(b)). 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. Fig. 4(c)). Then, the restore operation is performed on the entire heap. The time complexity for both cases is $O(\log_2 d_{threshold})$ and the two assertions are maintained. In summary, for all the cases the time complexity of the optimized LRFU implementation is $O(\log_2 d_{threshold})$.

On the LRU extreme of this optimized LRFU implementation (i.e., when $\lambda = 1$), $d_{threshold}$, which is given by $\lceil \frac{\log_{\frac{1}{2}}(1-(\frac{1}{2})^{\lambda})}{\lambda} \rceil$, is equal to 1. Thus, only one block needs to 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 O(1). On the other hand, as we move towards the LFU extreme, the number of blocks that should be maintained in the heap increases. Eventually, on the LFU extreme (i.e., when $\lambda = 0$), $d_{threshold}$ is equal to ∞ and, thus, every block should be maintained in the heap. As a result, the time complexity becomes $O(\log_2 n)$. This again coincides with the time complexity and the data structure of the native LFU implementation. Fig. 5 shows the spectrum of the LRFU implementations.

4 Experimental results

In this section, we discuss the results from trace-driven simulations performed to assess the effectiveness of the proposed LRFU policy. We used two different types of real workload traces: file system traces from the Sprite network file system [7] and database traces that consist of the DB2 trace used by Johnson and Shasha [3] and the OLTP trace used by both O'Neil *et al.* [1] and Johnson and Shasha [3].

The Sprite trace contains requests to a file server from client workstations for a two-day period. Among the client workstations, we selected three with the most requests (client workstations 54, 53 and 48) and simulated their buffer caches. Client workstation 54 made 203,808 references to 4,822 unique blocks, client workstation 53 made 141,223 references to 19,990 unique blocks, and client workstation 48 made 133,996 references to 7,075 unique blocks where the block size is 4 Kbytes. For all the three traces, no client caching was assumed in order to acquire the complete record of block references. Readers are referred to the paper by Baker *et al.* [7] for more details regarding the Sprite trace.

The DB2 trace was obtained from a commercial installation of DB2 and contains 500,000 references to 75,514 unique blocks [3]. 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 [1].

The performance of the LRFU policy is compared with those of the LRU, LRU-2, and 2Q policies. In the simulation, we used the weighing function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ explained in Section 3. For the 2Q and LRFU policies, the choice of parameters, specifically, the length of the A1 queue and the λ values, respectively, influence the performance of the policies. For our experiments, the length of the A1 queue was based on the suggestions of the authors [3], and among these results, the best ones are reported. Likewise for the LRFU, we report the best findings of the experiments. For all the policies, we factored out the references that were correlated with prior references as described in [2].

4.1 Comparison of the LRFU policy with other policies

Figs. 6 and 7 show the hit rates of the LRFU policy as a function of the cache size for the Sprite and database traces,

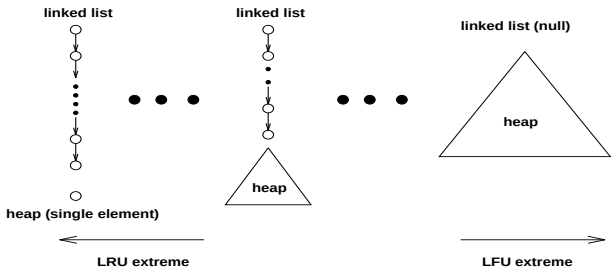


Figure 5: Spectrum of the LRFU implementations.

respectively. Note that the y-axis does not start from 0 and that the scales are different for each of the figures. This arrangement is intended to show clearly the relative performance of the policies.

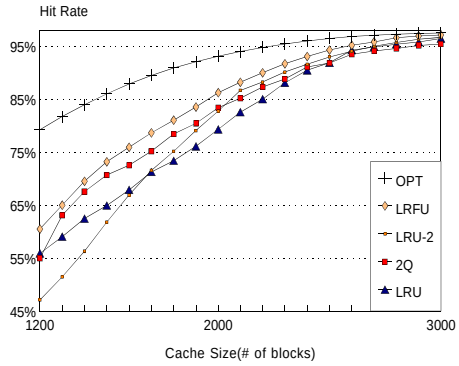
In the figures, for most cases, the LRFU policy performs best, while the LRU-2 and 2Q policies show similar performance, giving and taking at particular cache sizes. The 2Q policy performs strongly when the cache size is small, occasionally performing better than the LRFU policy. (The reason behind this is explained below.) However, its hit rate starts to converge early, that is, at a smaller cache size, than other policies. As was shown in earlier results, the LRU policy performs the worst [1, 3]. However, it performs reasonably well when the cache size is large.

Experiments were performed with increasing cache sizes until there was less than 1% change for all policies. The corresponding cache size ranges, when this is observed, differ widely depending on the locality of the workload represented by the trace; for the Sprite trace of client 54 its range is over $\frac{1}{2}$ of the footprint (3000 blocks vs. 4822 unique blocks in the trace); on the other hand, for the Sprite trace of client 48, it is below $\frac{1}{7}$ of the footprint (1000 blocks vs. 7075 unique blocks in the trace). When the cache size is beyond the above range, only a few misses will occur due to the lack of cache space and the miss rate will largely be affected by cold start misses, i.e., misses that occur when blocks are referenced for the first time. After this point, bigger caches and better replacement policies will improve the performance marginally; only look-ahead schemes and larger block sizes will be helpful.

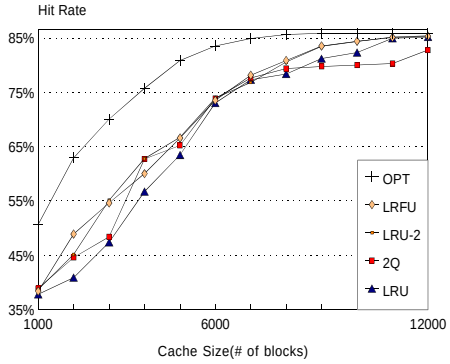
When the cache size is small, the hit rates of the LRFU policy is only comparable to those of the LRU-2 and 2Q policies for the Sprite trace (client workstation 53) and the DB2 trace. The LRFU policy even performs worse than these policies for some cache sizes. This is because in our experiments the LRU-2 and 2Q policies retain the history of references even after the blocks are replaced from the buffer cache [3]. Thus, when the block is brought back into the buffer cache it starts with a good knowledge of its past behavior. However, we chose not to allow this for the LRFU policy as this is a better representation of the real world.

After making these initial observations, we modified our simulation program so that the LRFU policy also retains the blocks' past history (i.e., $LAST(b)$ and $CRF_{last}(b)$) even after they are replaced as in the LRU-2 and 2Q policies. When this modification is made, the LRFU policy surpasses the LRU-2 and 2Q policies even for small cache sizes for client workstation 53 in the Sprite trace and the DB2 trace as can be seen in Table 1.

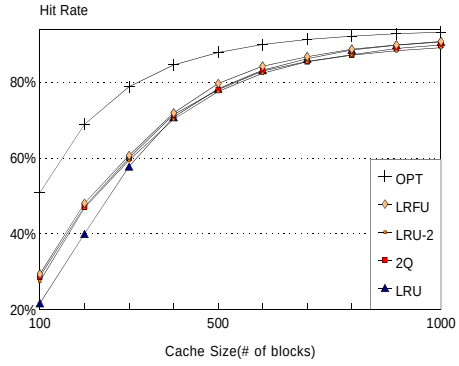
For comparison purposes, Figs. 6 and 7 also give the hit rate of the off-line optimal replacement policy, which replaces the block that will not be referenced for the longest



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

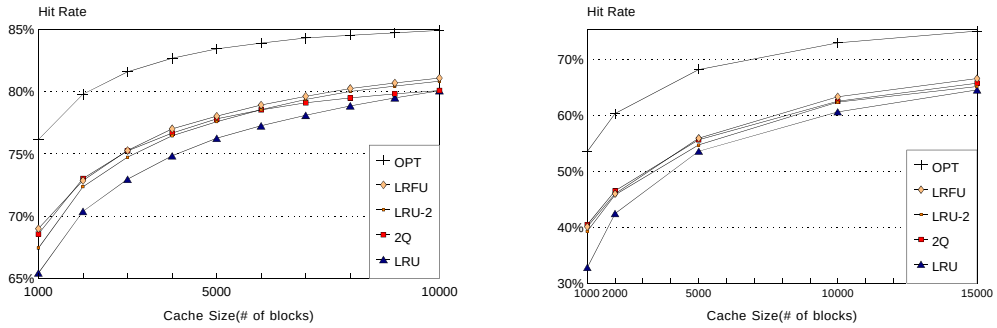


(b) **Client workstation 53 in the Sprite trace**



(c) **Client workstation 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 when the associated block is replaced for the LRFU as was done with the LRU-2 and 2Q policies.

Cache Size (blocks)	LRU	LRU-2	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

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

Cache Size (blocks)	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

(b) **DB2**

time.

4.2 Effects of λ on the performance of the LRFU policy

Fig. 8 shows the effect of λ on the hit rate for various cache sizes. All the figures in Fig. 8 have similar shapes. The hit rate initially increases as the λ value increases, that is, as the policy moves from the LFU extreme to the LRU extreme. After reaching a peak point, the hit rate drops slightly and then remains stable, decreasing very slowly until λ reaches 1. It can also be noted that as the cache size increases the peak hit rate is reached at a smaller λ value. This rather enlightening result indicates that as the cache size increases (which is the current trend) more weight must be given to older references, and that deciding the block to be replaced must not be made in a near-sighted manner. This result can also be used to determine an appropriate λ value for a given system configuration.

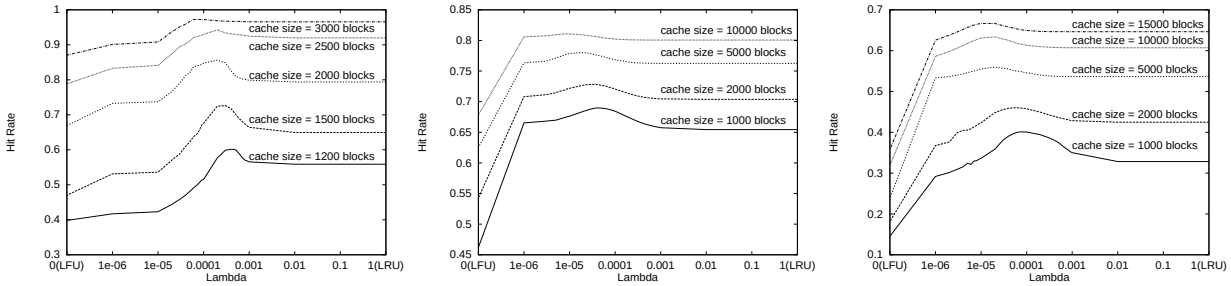
5 A Couple of Practical Issues

We have, thus far, provided a solid theoretical framework regarding the existence of a spectrum of policies that subsumes the LRU and LFU policies as well as a simulation-based evaluation of its performance. In this section, we briefly discuss issues in regards to actual deployment of the LRFU. The first is its actual integration into an existing operating system and the second is the self-adaptation of the λ value. The findings presented here are only preliminary and thus, our presentation is also concise.

5.1 Integration into the FreeBSD operating system

The LRFU policy was integrated into the FreeBSD operating system running on a Pentium PC. The LRU list within the FreeBSD that maintains the blocks based on their recency was replaced by the LRFU policy implemented as a linked list and a heap. For benchmarking purposes, we used the SPEC SDET benchmark [8] that simulates a multiprogramming environment. The benchmark consists of about 150 UNIX commands including *spell*, *nroff*, *diff*, *make*, and *find*. The benchmark was run three times and the results were averaged.

Fig. 9(a) shows the buffer cache hit rates of the SPEC SDET for various cache sizes as λ increases from 0 to 1. Since the working set size of the SDET benchmark is relatively small, the hit rates are very high for all cache sizes. However, we still observe that the peak of the LRFU outperforms the LRU (represented by LRFU with $\lambda = 1$). Note that the λ value that gives the peak hit rate moves to the LFU extreme as the cache size increases, which is consistent with the simulation results in Section 4. Fig. 9(b) gives the SDET throughput (scripts/hour) for the original FreeBSD LRU policy and the new LRFU implementation, which is calculated from the execution times of various commands in the benchmark and thus, takes into account overheads associated with each implementation. The results show that even a slight increase in the hit rate, as shown in Fig. 9(a), results in a considerable improvement on the throughput due to the huge penalty of physical disk I/Os. For example, when the cache size is 100 blocks, the hit rate difference between the LRU and the LRFU is less than 0.5% but this leads to more than a 4% difference in throughput.

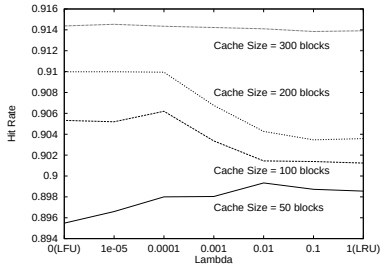


(a) **Sprite trace: Client 54** (b) **DB2** (c) **OLTP**

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

5.2 Self-adapting λ value

As the λ value influences the performance of the LRFU policy, this gives rise to a need for a mechanism that dynamically adjusts this value according to the workload evolution such that the best performance results.



(a) **Hit Rates**

Cache Size	LRU	LRFU
50	68.5	69.1
100	70.4	73.7
200	70.9	73.7
300	72.3	74.4

(b) **Throughputs (scripts/hour)**

Figure 9: Performance of LRFU integrated into the FreeBSD operating system using the SPEC SDET benchmark: (a) The hit rates for various cache sizes and λ values and (b) the comparison of throughputs for the LRU and LRFU policies.

In this subsection, we make a preliminary attempt to address that issue by periodically adjusting λ depending on whether the hit rate has improved during the last period. For example, if the hit rate of period i is better than that of period $i-1$ and the λ value for period i is larger (smaller) than the λ value for period $i-1$, the λ value is incremented (decremented). On the other hand, if the hit rate of period i is worse than that of period $i-1$ and the λ value for period i is larger (smaller) than the λ value for period $i-1$, the λ value is decremented (incremented). However, a problem with this simplistic approach is that the improvement (degradation) of the hit rate may result from better (worse) locality of the workload rather than a better (worse) choice of λ.

Table 2: Results of the adaptive LRFU policy.

Cache Size	LRU	LRFU (Non-adaptive)	Adaptive LRFU
1200	0.5588	0.6049	0.5872
1400	0.6247	0.6952	0.6688
1600	0.6789	0.7601	0.7478
1800	0.7346	0.8112	0.8017
2000	0.7939	0.8634	0.8461
2200	0.8511	0.9009	0.8851
2400	0.9057	0.9317	0.9199
2600	0.9437	0.9530	0.9492
2800	0.9552	0.9672	0.9641
3000	0.9657	0.9726	0.9707

(a) Client workstation 54 in the Sprite trace

Cache Size	LRU	LRFU (Non-adaptive)	Adaptive LRFU
1000	0.6544	0.6899	0.6772
2000	0.7038	0.7285	0.7213
3000	0.7295	0.7527	0.7463
4000	0.7483	0.7701	0.7575
5000	0.7625	0.7802	0.7652
6000	0.7725	0.7891	0.7754
7000	0.7809	0.7962	0.7815
8000	0.7885	0.8024	0.7951
9000	0.7949	0.8068	0.7997
10000	0.8006	0.8107	0.8023

(b) DB2

To rectify this problem, we use the LRU policy as the reference model to quantify how good (or bad) the locality of the workload has been and adjust λ according to the hit rate improvement (degradation) of the LRFU relative to that of the LRU. In other words, if $\frac{hit_i^{LRFU} - hit_{i-1}^{LRFU}}{hit_{i-1}^{LRFU}}$ is greater than or equal to $\frac{hit_i^{LRU} - hit_{i-1}^{LRU}}{hit_{i-1}^{LRU}}$ (where hit_i^X is the hit rate of policy X during period i), the λ value for period $i+1$ is updated in the same direction as it has been for periods $i-1$ and i. Otherwise, the direction is reversed. In both cases, the increment (decrement) of λ depends on the λ value for period i and is given by $\frac{1}{10}^{\lfloor \log_{\frac{1}{10}} \lambda \rfloor + 1}$. For example, if the λ value is 0.003, the increment (decrement) is 0.001, and if the current λ value is 0.1, the increment (decrement)

is 0.01. We refer to this dynamic version of LRFU as the adaptive LRFU.

Table 2 compares the adaptive LRFU with the non-adaptive LRFU and the LRU for the Sprite client 54 trace (which has the largest number of references among the three Sprite traces) and the DB2 trace. Note that we consider only the non-adaptive LRFU and the LRU for this comparison since the other policies discussed in Section 4 have their own control parameters that require tuning. In all the experiments, the initial λ value is set to 0.0001 and the period to 10,000 references.

The results show that the adaptive LRFU consistently outperforms the LRU. The results however also show that there is still a performance gap between the adaptive LRFU and the non-adaptive LRFU. We think that the performance gap results mainly from the following three factors. First, the λ value converges to its optimal value rather slowly. Thus, there is always a difference between the true optimal λ value for the period and the one actually used. Second, since the current method uses only the LRU as its reference model, its prediction about the locality of the workload is inaccurate when the workload follows the independent reference model [9] rather than the LRU stack model [10], and this inaccuracy is expected to be more profound at larger cache sizes. Finally, the traces used are relatively short to observe the long term behavior of the adaptive LRFU. We think that the policy explained in this subsection is just a first step to a truly adaptive LRFU policy and that much research is still needed towards this direction.

6 Conclusion

In this paper, we have shown that there exists a spectrum of policies that subsumes the well-known LRU and LFU policies, in the form of the LRFU (Least Recently/Frequently Used) block replacement policy. The LRFU policy provides 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 weights given to recent and old history thereby providing grounds for an optimal combination of the effects of recency and frequency factors of past references on the likelihood of future re-reference.

Unlike previous policies that consider only limited reference history in their replacement decision, the LRFU policy uses all the reference history of blocks. We showed that this can be achieved with only a few words for each block. We also showed that the LRFU allows for an efficient implementation whose time complexity ranges from $O(1)$ to $O(\log_2 n)$ depending on the value of λ where n is the number of blocks in the buffer cache. These time complexities correspond to the time complexities of the native implementations of the LRU and LFU policies. Finally, we provided a preliminary investigation of practical issues such as actual deployment in a real operating system and the adaptive LRFU where the value of the control parameter λ changes periodically as the workload evolves.

Results from trace-driven simulations showed that the LRFU policy performs better than the LRU, LRU-2, and 2Q policies for the workloads considered and the results were reinforced by benchmark results from our implementation in the FreeBSD operating system. The results also revealed the following two performance effects of the controllable parameter λ. First, as the λ value increases, the hit rate initially increases, reaches a peak, and drops slightly after the peak. Second, as the cache size increases, the peak hit rate is obtained at a smaller λ value. This suggests that more weight should be given to older references for larger caches.

One direction for future research is to develop a program reference model related to the LRFU policy. An example of such a reference model is one that has λ as its parameter and leads to the optimal performance under the LRFU with the corresponding λ value like the LRU stack model [10] for the LRU and the independent reference model [9] for the LFU. Another research direction is to improve on the adaptive LRFU presented in Section 5. Finally, applying our concept of combining recency and frequency to page and data placement and migration in distributed systems with a hierarchy of buffer caches is also a direction for future research.

Acknowledgment and Dedication

We would like to thank Gerhard Weikum, Theodore Johnson, and Pei Cao for providing us with traces as well as useful information regarding the experiments. Many thanks goes to the reviewers who provided enlightening comments.

We dedicate this paper to Jong-Hun Kim, a former student at Seoul National University, who, so suddenly, passed away early last year. Jong-Hun was an active participant in this research, and yet, did not have the chance to see its results published. We all miss his company.

References

[1] 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*, pp. 297–306, 1993.

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

[3] 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*, pp. 439–450, 1994.

[4] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 560–595, 1984.

[5] C. Faloutsos, R. Ng, and T. Sellis, "Flexible and Adaptable Buffer Management Techniques for Database Management Systems," *IEEE Transactions on Computers*, vol. 44, no. 4, pp. 546–560, 1995.

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

[7] 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*, pp. 198–212, 1991.

[8] SPEC, "SPEC SDM Release 1.1," Mar. 1992.

[9] G. S. Gao, "Performance Analysis of Cache Memories," *Journal of ACM*, vol. 25, no. 3, pp. 378–395, July 1978.

[10] J. R. Spirn, *Program Behavior: Models and Measurements*. Elsevier North-Holland, NY, 1977.