LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies

Donghee Lee, *Member*, *IEEE*, Jongmoo Choi, *Member*, *IEEE*, Jong-Hun Kim, *Member*, *IEEE*, Sam H. Noh, *Member*, *IEEE*, Sang Lyul Min, *Member*, *IEEE*, Yookun Cho, *Member*, *IEEE*, and Chong Sang Kim, *Senior Member*, *IEEE*

Abstract—Efficient and effective buffering of disk blocks in main memory is critical for better file system performance due to a wide speed gap between main memory and hard disks. In such a buffering system, one of the most important design decisions is the block replacement policy that determines which disk block to replace when the buffer is full. In this paper, we show that there exists a spectrum of block replacement policies that subsumes the two seemingly unrelated and independent Least Recently Used (LRU) and Least Frequently Used (LFU) policies. The spectrum is called the LRFU (Least Recently/Frequently Used) policy and is formed by how much more weight we give to the recent history than to the older history. We also show that there is a spectrum of implementations of the LRFU that again subsumes the LRU and LFU implementations. This spectrum is again dictated by how much weight is given to recent and older histories and the time complexity of the implementations lies between O(1) (the time complexity of LRU) and $O(\log_2 n)$ (the time complexity of LFU), where n is the number of blocks in the buffer. Experimental results from trace-driven simulations show that the performance of the LRFU is at least competitive with that of previously known policies for the workloads we considered.

Index Terms—Buffer cache, LFU, LRU, replacement policy, trace-driven simulation.

1 Introduction

To bridge the wide speed gap between main memory and hard disks, much research has been performed on buffering disk blocks in main memory. Such a buffering system is called a buffer cache [1] and one of its most important design decisions is the block replacement policy that decides the block to be replaced when the buffer cache is full. Efficient and effective block replacement policies have been the topic of much research in both the systems [2], [3], [4], [5], [6], [7] and database [8], [9], [10], [11], [12] areas. Of these, the Least Recently Used (LRU) and the Least Frequently Used (LFU) block replacement policies constitute the two main streams. The LRU policy and its variants base their replacement decision on the recency of references, while the LFU policy and its variants base their decision on the frequency of references. In this paper, we show that, between these seemingly unrelated and independent two policies, there exists a spectrum of policies, with the LRU and LFU policies existing as the two extreme points. This spectrum of policies, which we refer to as the Least Recently/Frequently Used (LRFU) policy, inherits the benefits of the two policies and allows a flexible trade-off between recency and frequency of references in basing the replacement decision. The decision to lean toward the LRU or the LFU is made through the use of a parameter λ, which essentially determines how much more weight we give to the recent history than to the older history.

In this paper, we also show that there is a spectrum of implementations of the LRFU that again subsumes the LRU and LFU implementations. That is, for each point in the spectrum of policies of the LRFU, there exists a corresponding implementation that lies between the LRU and LFU. Hence, the spectrum of implementations has time complexity that ranges between O(1), which is the time complexity for the LRU policy, and $O(\log_2 n)$, which is the time complexity for the LFU policy, where n is the number of blocks in the buffer cache.

The remainder of this paper is organized as follows: In the next section, we review some of the previous policies that have been proposed for buffer caching. In Section 3, we describe the LRFU policy in detail. Its implementation is discussed in Section 4. In Section 5, we discuss two extensions to the LRFU policy, namely, the issue of optimized implementation and correlated references [11], [12]. In Section 6, we compare the performance of the LRFU policy with that of previous policies through trace-driven simulations and discuss the performance impact of the

- *D. Lee is with the School of Computer, Communications, and Commerce, Hanyang University, Seoul 133-791, Korea. E-mail: dhlee@ssrnet.snu.ac.kr.*
- *J. Choi is with the Ubiquix Company, Samwha B/D 2F, 204-4, Nonhyon-Dong, Kangnam-gu, Seoul 135010, Korea. E-mail: choijm@ubiquix.com.*
- *J.-H. Kim is deceased.*
- *S.H. Noh is with the School of Information and Computer Engineering, College of Engineering, Hong-Ik University, Sangsu Dong, Mapo Gu, Seoul, 121-791, Korea. E-mail: samhnoh@hongik.ac.kr.*
- *S.L. Min, Y. Cho, and C.S. Kim are with the School of Computer Science and Engineering, College of Engineering, Seoul National University, Seoul, 151-742, Korea. E-mail: symin@dandelion.snu.ac.kr, cho@ssrnet.snu.ac.kr, cskim@sparc.snu.ac.kr.*

Manuscript received 10 Mar. 2000; revised 9 Apr. 2001; accepted 28 June 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111694.

control parameter λ and the correlated references. Finally, we conclude this paper in Section 7.

2 Related Works

The study of block replacement policies is, in essence, a study of relating past access patterns with future access behavior. 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 [13], [14]. The LRU policy does this by attaining the recency of block references while the LFU policy considers the frequency of block references. One problem of the LRU policy is that, since it uses only the time of the most recent reference to each block, it cannot discriminate well between frequently and infrequently referenced blocks. On the other hand, the problem with the LFU policy is that it cannot distinguish between references that occurred far back in the past and the more recent ones in recording the frequency and, thus, cannot adapt well to changing workloads.

In the following, we survey the studies that aim to overcome the limitations of the LRU and LFU policies. Our discussion focuses on two papers, one by Robinson and Devarakonda [12] and the other by O'Neil et al. [11]. Before describing these studies we need to explain the concept of correlated references. In general, references to disk blocks have less locality than references to CPU caches or virtual memory pages [12]. However, references to a disk block still exhibit 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 given in [11], [12].

A frequency-based policy, called the FBR (Frequency-Based Replacement), in which the notion of a correlated reference was introduced was proposed by Robinson and Devarakonda [12]. The main 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 area called a *new section* [12]. The new section consists of $k_{new\_section}$ most recently referenced blocks, where the value of $k_{new\_section}$ is implementation dependent. When there is a hit to a block in the new section, the corresponding reference is considered to be correlated with a previous reference to the same block and the reference count of the block is not incremented. In addition to the new section, there are two more sections in the buffer cache: the old section and the middle section. In the FBR, replacement is confined to blocks in the old section which consists of $k_{old\_section}$ least recently referenced blocks, where $k_{old\_section}$ is an implementation-dependent parameter. The middle section is located between the new section and the old section in the LRU stack and is intended to allow blocks to build up their reference counts before becoming eligible for replacement.

In the FBR, the block to be replaced when the buffer cache is full is the block in the old section with the smallest reference count if it does not exceed C_{max}, where the value of C_{max} is implementation dependent; otherwise, the least recently referenced block is replaced. The FBR policy also uses a periodic aging mechanism where every reference count is halved whenever the average reference count exceeds a predetermined maximum value A_{max}. This mechanism is intended to prevent blocks with large reference counts from being fixed in the cache, a problem commonly called the cache pollution problem.

O'Neil et al. present the LRU-*K* replacement policy that bases its replacement decision on the time of the *K*th-to-last reference to the block [11]. In other words, its replacement decision is based on the reference density [15] observed during the past K references. Thus, when K is large and the reference shows a stationary pattern, the LRU-*K* 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 *K*th-to-last reference time.

The LRU-*K* ignores the recency of the $K-1$ references and considers only the distance of the *K*th 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, the LRU-*K* incurs an $O(K)$ space overhead to keep the history of the last K references. This may not be a serious problem since, in practice, smaller K values such as 2 or 3 are preferred for better performance [11]. Another potential problem with the LRU-*K* is that, since it 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 [10]. 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, called the Am queue, when it is rereferenced while in the A1 queue. Otherwise, it is replaced when it becomes the LRU block in the A1 queue. When a block is replaced, its history is retained in a queue called the Aout queue. When a block misses in the buffer cache but its history is still in the Aout queue, it is placed in the Am queue directly without going through the Ain queue. These mechanisms quickly remove from the buffer cache blocks that are referenced only once, while retaining in the buffer cache, for an extended period of time, blocks that are repeatedly referenced. 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 [9] (also see the references therein). However, many of its algorithms make use of information deduced externally, such as from query optimizer plans. Since such information is usually not available in general file caching, these schemes are applicable only to database systems.

Other approaches have been proposed that exploit external information or hints, such as the application-controlled file caching scheme [3], or that make use of information generated internally, such as the DEAR scheme [16]. These schemes are promising approaches, but are beyond the scope of this paper.

3 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.

Definition 1. *Assume that the system time is represented by an integer value and is incremented by one on each block reference. 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 $t_{b_1} < t_{b_2} < \cdots < t_{b_k} \leq t_{base}$.

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

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

The weighing function $\mathcal{F}(x)$ essentially reflects the influence of the recency and frequency factors of a block's past references in projecting the likelihood of its rereference in the future. In general, $\mathcal{F}(x)$ is a monotonically nonincreasing 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.

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 all the other references in the past.

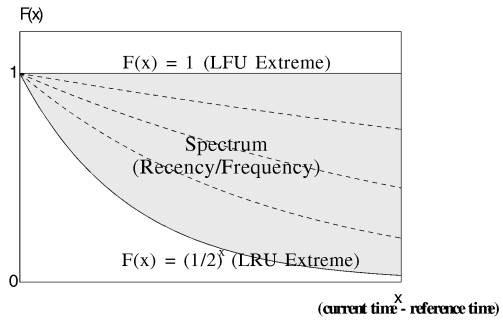

Fig. 1. Spectrum of LRFU according to the function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$, where x is $(\text{current\_time} - \text{reference\_time})$.

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 positive 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 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.*

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

A class of functions that satisfy both Property 1 and Property 2 is $\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$, where x is the difference between the current time and the time of a reference in the past, $p \geq 2$, and λ ranges from 0 to 1. This class of functions, where $p = 2$, is shown in Fig. 1. Note that each function in this class is monotonically nonincreasing with x to give more weight to more recent references with smaller backward distances, which is consistent with the principle of temporal locality. 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 10,000 time steps. This control parameter λ allows a trade-off between recency and frequency in projecting the likelihood of future references. 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 toward 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. 1 is where the LRFU policy differs from both LFU and LRU, assuming $p = 2$.

4 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 practical value, these issues must be rectified.

4.1 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 the $\mathcal{F}(x+y) = \mathcal{F}(x)\mathcal{F}(y)$ property as the function $\mathcal{F}(x) = (\frac{1}{p})^x$ ($p \geq 2$) in the previous section does, the storage and computational overheads can be reduced drastically such that this policy becomes not only implementable but also efficient.

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

$$\mathcal{C}_{t_{b_k}}(b) = \mathcal{F}(0) + \mathcal{F}(\delta)\mathcal{C}_{t_{b_{k-1}}}(b),$$

where $\delta = t_{b_k} - t_{b_{k-1}}$.

Proof. See [17]. □

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 $\delta = t_c - t_{b_k}$ assuming that the kth reference is the most recent reference to the block. 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.

4.2 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 value changes over time and, thus, 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 $\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$ for $p \geq 2$, the relative ordering between two blocks does not change until either of them is referenced and, hence, reordering of blocks is needed only upon a block reference, as the following property shows.

```
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. t_c = t_c + 1
16. fi
17. ──────────────────────────────────────
18. Restore(H, b)
19      if b is not a leaf node
20.     then
21.         let smaller be the child that has
                the smaller CRF value at the current time
22.         if F(t_c − LAST(b)) * CRF_last(b) >
                F(t_c − LAST(smaller)) * CRF_last(smaller)
23.         then
24.             swap(H, b, smaller)
25.             Restore(H, smaller)
26.         fi
27.     fi
28. end Restore
29. ──────────────────────────────────────
30. ReplaceRoot(H, b)
31.     victim = H.root
32.     H.root = b
33.     Restore(H, b)
34.     return victim
35. end ReplaceRoot
36. ──────────────────────────────────────
```

Fig. 2. Buffer cache management algorithm for the LRFU algorithm.

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. See [17]. □

We have presented, thus far, $\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$ for $p \geq 2$ 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 the range of control parameter λ covering both the LFU and LRU is between 0 and 1, which is common in other studies that involve control parameters.

4.3 The Algorithm

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. 2 that is invoked upon a block reference, 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.

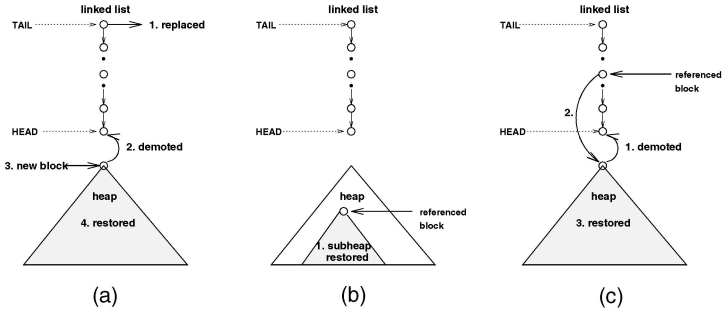

Fig. 3. Optimized implementation of the LRFU policy.

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 subheap 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. Finally, the current time t_c is incremented by one to reflect the progress of the virtual time due to the reference.

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., $\lceil \log_2(n+1) \rceil - 1$, where n is the number of blocks in the buffer cache. The only additional overhead of the LRFU over other policies is due to the invocations of $\mathcal{F}(x)$ when CRF values are compared.

5 Extensions to the LRFU Policy

In this section, we discuss two extensions to the LRFU policy presented in the previous section. The first concerns an optimized implementation of the policy, where we show that the time complexity of the LRFU policy ranges between O(1) and $\mathrm{O}(\log_2 n)$, depending on the value of control parameter λ. The second extension concerns the incorporation of the notion of correlated references into the LRFU framework. We show that this can be done seamlessly without complicating the implementation.

5.1 Optimized Implementation of the LRFU Policy

The $\mathrm{O}(\log_2 n)$ 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, the buffer cache is divided into multiple partitions as in 2Q [10] and FBR [12]. In the spectrum, the points corresponding to the LRU and the LFU have O(1) and $\mathrm{O}(\log_2 n)$ 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

$$\left\lceil \frac{\log_{\frac{1}{2}}(1 - (\frac{1}{2})^{\lambda})}{\lambda} \right\rceil.$$

Proof. See [17]. □

This property states that a block whose most recent reference was made earlier than $d_{threshold}$ time units ago (where one time unit corresponds to one block reference) 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, it is possible to 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. 3a). Then, the 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 be further divided into two cases depending on whether the requested block is in the heap or in the linked list. First, consider the case where the requested block is in the heap. Here, the restore operation needs to be performed only for the subheap rooted by the requested block (cf. Fig. 3b). In the other case, where the 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

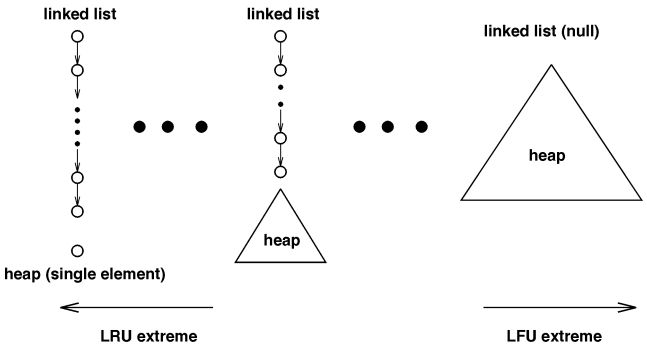

Fig. 4. Spectrum of the LRFU implementations.

requested block becomes the new root (cf. Fig. 3c). 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

$$\left\lceil \frac{\log_{\frac{1}{2}}(1 - (\frac{1}{2})^{\lambda})}{\lambda} \right\rceil,$$

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 toward 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 $\mathrm{O}(\log_2 n)$. This again coincides with the time complexity and the data structure of the native LFU implementation. Fig. 4 shows the spectrum of the LRFU implementations.

5.2 LRFU with Correlated References

In this section, we show how the notion of correlated references can seamlessly be incorporated into the LRFU framework. The notion of correlated references was 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 [11], [12]. In congruence with the description given by Robinson and Devarakonda [12], all the references to a block within a correlated period are treated as a single noncorrelated reference to the block.

To incorporate the concept of correlated references more formally, we introduce a masking function $\mathcal{G}_c(x)$

$$\mathcal{G}_c(x) = \begin{cases} 0 & x \le c \\ 1 & x > c, \end{cases}$$

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

With this masking function, the CRF value is calculated as follows:

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

where $\mathcal{C}'_{t_c}(b)$ is the CRF value of b at t_c when correlated references are considered. This revision affects neither the way the CRF value is calculated nor the basic structure of the buffer cache management algorithm as the following property shows.

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 the time of the kth reference when correlated references are considered, can be derived from $\mathcal{C}'_{t_{b_{k-1}}}(b)$ and $t_{b_{k-1}}$ as follows:*

$$\mathcal{C}'_{t_{b_k}}(b) = \mathcal{F}(0) + \mathcal{F}(\delta) * [\mathcal{F}(0) * \mathcal{G}_c(\delta) + \mathcal{C}'_{t_{b_{k-1}}}(b) - \mathcal{F}(0)],$$

where $\delta = t_{b_k} - t_{b_{k-1}}$.

Proof. See [17]. □

6 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 [18] and database traces that consist of the DB2 trace used by Johnson and Shasha [10] and the OLTP trace used by both O'Neil et al. [11] and Johnson and Shasha [10].

The Sprite trace contains requests to a file server from client workstations for a two-day period. Among the client workstations, we selected the 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.

The DB2 trace was obtained from a commercial installation of DB2 and contains 500,000 references to 75,514 unique blocks [10]. 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 [11]. We note that the traces are those used in previous researches [10], [11] and were obtained from the authors of those papers.

The performance of the LRFU policy is compared with that of the LRU, LFU, LRU-2, 2Q, and FBR policies. The LRU-2, 2Q, and FBR policies were simulated according to the descriptions in [11], [10], [12]. The results of the LRU-2 and 2Q policies were obtained while changing their correlated periods and the best results were selected for each policy. In the 2Q policy, the length of the A1 queue influences the performance. We performed experiments with different A1 queue lengths suggested in [10] and,

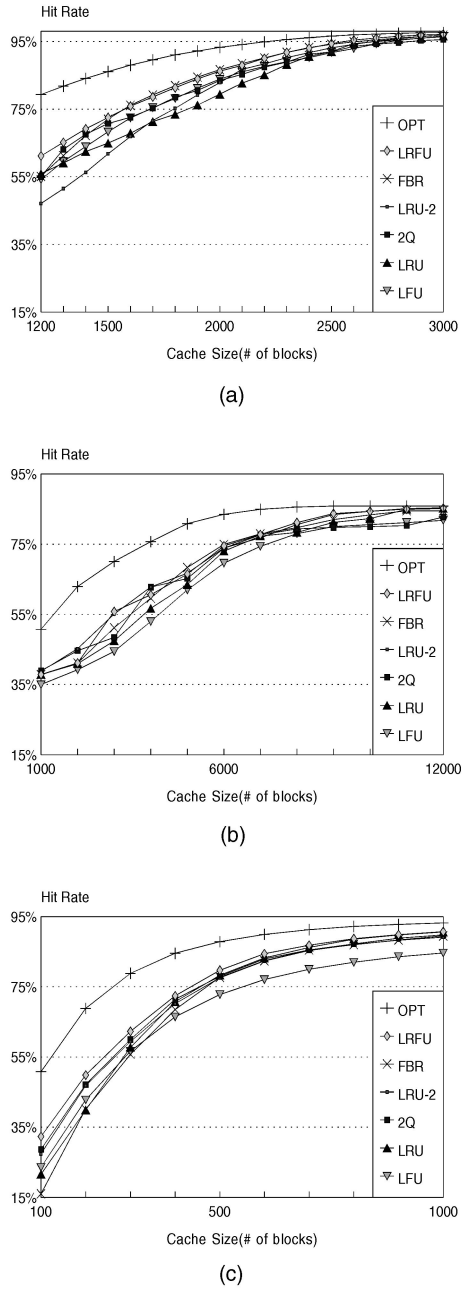

Fig. 5. Comparison of LRFU with other policies using the Sprite trace. (a) Client workstation 54 in the Sprite trace. (b) Client workstation 53 in the Sprite trace. (c) Client workstation 48 in the Sprite trace.

among them, the best results are reported. Likewise for the LRFU, we report the best results of the experiments with different λ values. The correlation period in the LRFU is set to 60 percent of the cache size with a maximum of 2,000 for all the experiments. For the FBR policy, we allocate $1/4$ of the cache for the new section and split the remainder in half, that is, $3/8$ of the cache for the middle and old sections, respectively, with $C_{max} = 4$ and $A_{max} = 100$ [12].

Experiments were performed with increasing cache sizes until there was less than 1 percent change for all policies and for LFU, LRU-2, 2Q, and LRFU, and the history of past references was retained even after the corresponding blocks were replaced from the buffer cache.

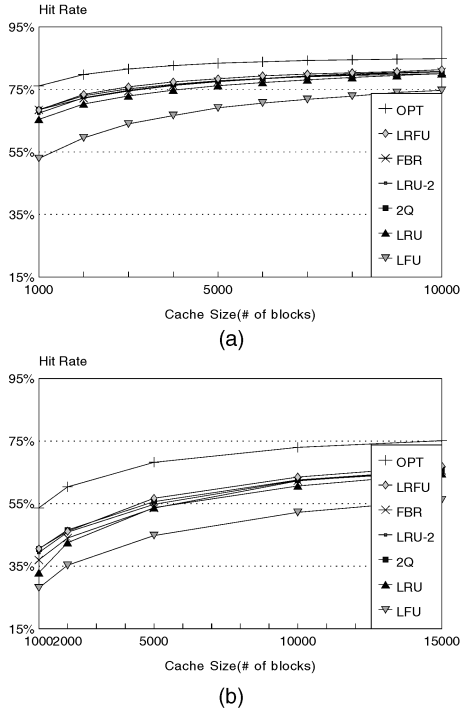

Fig. 6. Comparison of LRFU with other policies using the database trace. (a) DB2. (b) OLTP.

6.1 Comparison of the LRFU Policy with Other Policies

Figs. 5 and 6 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 the LRU, LFU, LRU-2, 2Q, and FBR policies.

The results in the figures show that the LRFU policy performs at least competitively with other policies throughout the cache sizes we simulated, while the LRU-2, 2Q, and FBR policies show comparable performance at particular cache sizes. For example, the 2Q policy performs very well at small cache sizes. However, its hit rate starts to converge early, that is, at a smaller cache size, than other policies. For large cache sizes, the FBR policy shows performance that is nearly as good as that of the LRFU policy. The LRU and LFU policies show the worst performance due to their shortcomings explained in Section 2.

For comparison purposes, Figs. 5 and 6 also give the hit rate of the offline optimal replacement policy, which replaces the block that will not be referenced for the longest time. Notice that there is a huge gap between the hit rate of the optimal policy and those of the online policies. This gap results from the fact that the optimal policy makes use of knowledge about future references which is not available to the online policies.

6.2 Effects of λ on the Performance of the LRFU Policy

Fig. 7 shows the effect of λ on the hit rate for various cache sizes. All the figures in Fig. 7 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

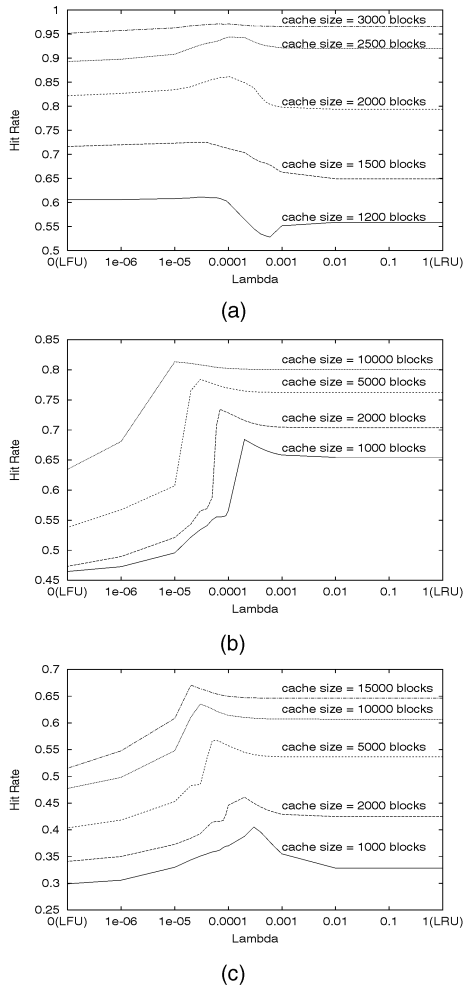

Fig. 7. Effects of λ on the LRFU policy using Sprite and database traces. (a) Sprite trace: Client 54. (b) DB2. (c) OLTP.

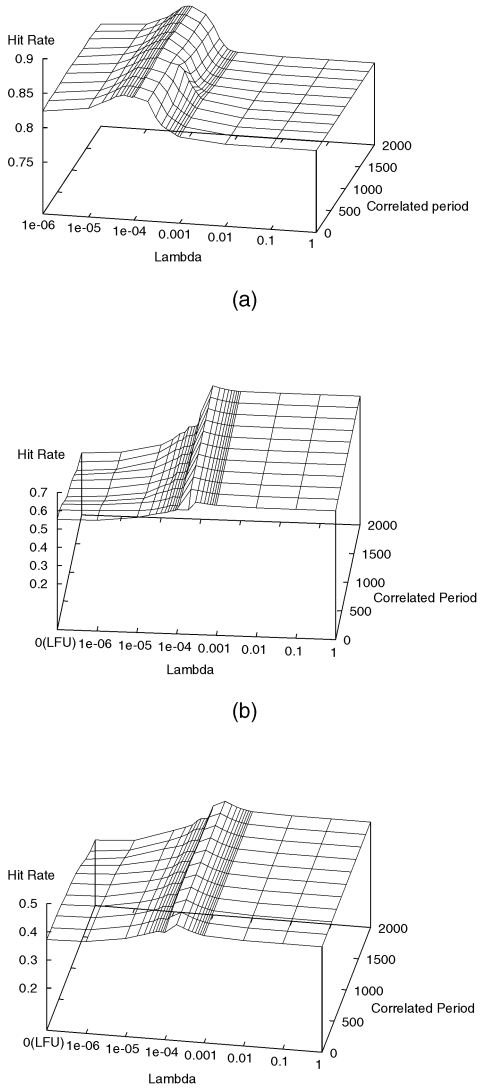

Fig. 8. Combined effects of λ and c on the LRFU policy. (a) Sprite trace: Client 54 (cache size = 2,000 blocks). (b) DB2 (cache size = 1,000 blocks). (c) OLTP (cache size = 2,000 blocks).

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 should be given to older references and that deciding the block to be replaced must not be made in a near-sighted manner.

6.3 Combined Effects of λ and Correlated Period on the Performance of the LRFU Policy

Fig. 8a, Fig. 8b, and Fig. 8c show the hit rate as a function of λ and c for the Sprite trace (client workstation 54) with a cache size of 2,000, for the DB2 trace with a cache size of 1,000, and for the OLTP trace with a cache size of 2,000, respectively. Overall, for all correlated periods, we observe a performance effect of λ that is similar to that in Fig. 7, i.e., the hit rate initially increases, reaches a peak, and drops slightly after the peak.

We also observe that the effect of the correlated period becomes more significant as the LRFU policy moves toward the LFU policy. However, the correlated period has very little effect as the LRFU policy moves toward the LRU extreme. This observation agrees with, and indirectly explains, the reason behind the improvement by the FBR policy [12]. An important 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 improved by considering the correlated references. Hence, the FBR policy benefited from the addition of the new section.

7 Conclusion

In many file systems, buffer caches are used to reduce the effective disk access time and the traffic to the disk subsystem. In a buffer cache, one of the most important design decisions is the block replacement policy that decides which block to replace when the buffer cache is full.

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 control parameter that determines how much more weight we give to recent references than older references. We also showed that the LRFU gives a spectrum of implementations whose time complexity ranges from O(1) to $O(\log_2 n)$, depending on the value of control parameter 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.

Results from performance evaluation through trace-driven simulation show that the LRFU policy performs at least competitively with other policies regardless of the cache size. The results also show that, as the cache size increases, the λ value that gives the optimal performance under the LRFU decreases, which indicates that more weight should be given to older references to prevent the replacement from being made in a near-sighted manner.

One direction for future research is to develop a mechanism that dynamically adjusts the value of control parameter λ according to the evolution of the workload. We expect that this will improve the performance of the LRFU, further reducing the gap between its performance and that of the offline optimal replacement policy. Another 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 optimal performance under the LRFU with the corresponding λ value like the LRU stack model [19] for the LRU and the independent reference model [20] for the LFU. Finally, applying our concept of combining recency and frequency to data placement and migration in distributed systems with a hierarchy of buffer caches is also a direction for future research.

ACKNOWLEDGMENTS

The authors would like to thank Gerhard Weikum, Theodore Johnson, and Pei Cao for providing them with traces as well as useful information regarding the experiments. They would also like to thank the reviewers for their constructive and enlightening comments. A preliminary version of this paper appeared in the *Proceedings of the 1999 SIGMETRICS Conference*. This paper is dedicated to Jong-Hun Kim, a former student at Seoul National University, who, so suddenly, passed away in 1998. Jong-Hun was an active participant in this research and yet did not have the chance to see its results published. His company is gratly missed. This work was supported in part by the Ministry of Education under the BK21 program and by the Ministry of Science and Technology under the National Research Laboratory program.

REFERENCES

[1] M.J. Bach, *The Design of the UNIX Operating System.* Englewood Cliffs, N.J.: Prentice Hall, 1986.
[2] E.G, Coffman Jr. and P.J. Denning, *Operating Systems Theory.* Englewood Cliffs, N.J.: Prentice Hall, 1973.
[3] P. Cao, E.W. Felten, and K. Li, "Application-Controlled File Caching Policies," *Proc. Summer 1994 USENIX Conf.,* pp. 171-182, 1994.
[4] R. Karedla, J.S. Love, and B.G. Wherry, "Caching Strategies to Improve Disk System Performance," *Computer,* vol. 27, no. 3, pp. 38-46, Mar. 1994.
[5] V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," *Proc. 1995 ACM SIGMETRICS/PERFORMANCE Conf.,* pp. 291-300, 1995.
[6] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *Proc. 1997 ACM SIGMETRICS Conf.,* pp. 115-126, 1997.
[7] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," *Proc. 1999 ACM SIGMETRICS Conf.,* pp. 122-133, 1999.
[8] H. Chon and D. DeWitt, "An Evaluation of Buffer Management for Relational Database Systems," *Proc. 11th Int'l Conf. Very Large Data Bases,* 1985.
[9] C. Faloutsos, R. Ng, and T. Sellis, "Flexible and Adaptable Buffer Management Techniques for Database Management Systems," *IEEE Trans. Computers,* vol. 44, no. 4, pp. 546-560, Apr. 1995.
[10] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th Int'l Conf. Very Large Data Bases,* pp. 439-450, 1994.
[11] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm For Database Disk Buffering," *Proc. 1993 ACM SIGMOD Conf.,* pp. 297-306, 1993.
[12] J.T. Robinson and N.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. 1990 ACM SIGMETRICS Conf.,* pp. 134-142, 1990.
[13] L.A. Belady, "A Study of Replacement Algorithms for Virtual-Storage Computers," *IBM Systems J.,* vol. 5, no. 2, pp. 78-101, 1966.
[14] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, no. 2, pp. 78-117, 1970.
[15] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Trans. Database Systems,* vol. 9, no. 4, pp. 560-595, 1984.
[16] J. Choi, S.H. Noh, S.L. Min, and Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme," *Proc. 1999 USENIX Conf.,* pp. 239-252, 1999.
[17] D. Lee, J. Choi, J.-H. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," technical report, School of Computer Science and Eng., Seoul Nat'l Univ., 2001. http://archi.snu.ac.kr/symin/lrfutr.ps.
[18] M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout, "Measurements of a Distributed File System," *Proc. 13th ACM Symp. Operating Systems Principles,* pp. 198-212, 1991.
[19] J.R. Spirn, *Program Behavior: Models and Measurements.* North Holland, New York: Elsevier, 1977.
[20] G.S. Gao, "Performance Analysis of Cache Memories," *J. ACM,* vol. 25, no. 3, pp. 378-395, 1978.

Donghee Lee received the MS and PhD degrees in computer engineering, both from Seoul National University, Seoul, Korea, in 1991 and 1998, respectively. He is currently an assitant professor in the School of Computer, Communications, and Commerce, Hanyang University, Korea. Previously, he was a senior engineer at Samsung Electronics Company, Korea, in 1998, and an assistant professor in the School of Telecommunication and Computer Engineering, Cheju National University, Korea, from 1999 to 2001. His research interests include operating systems, high performance I/O systems, flash memory software, and embedded real-time systems. He is a member of the IEEE and the IEEE Computer Society.

Jongmoo Choi received the BS degree in oceanography from Seoul National University, Korea, in 1993 and the MS and PhD degrees in computer engineering from Seoul National University in 1995 and 2001, respectively. Currently, he is with the Ubiquix Company, Korea, where he is participating in developing a real-time embedded micro-kernel for PDAs. His research interests include micro-kernels, file systems, I/O systems, flash memory, and data mining. He is a member of the IEEE and the IEEE Computer Society.

Jong-Hun Kim received the MS degree in computer engineering from Seoul National University, Korea, in 1997. He was a PhD student in Seoul National University and passed away suddenly in 1998. He was a member of the IEEE and the IEEE Computer Society.

Sam H. Noh received the BS degree in computer engineering from Seoul National University, Korea, in 1986, and the PhD degree from the University of Maryland at College Park in 1993. He held a visiting faculty position at George Washington University from 1993 to 1994 before joining Hong-Ik University in Seoul Korea, where he is now an associate professor in the School of Information and Computer Engineering. His current research interests include parallel and distributed systems, I/O issues in operating systems, and real-time systems. Dr. Noh is a member of the IEEE, the IEEE Computer Society, and the ACM.

Sang Lyul Min received the BS and MS degrees in computer engineering, both from Seoul National University, Seoul, Korea, in 1983 and 1985, respectively. In 1985, he was awarded a Fullbright scholarship to pursue further graduate studies at the University of Washington. He received the MS and PhD degrees in computer science from the University of Washington, Seattle, in 1988 and 1989, respectively. He is currently a professor in the School of Computer Science and Engineering, Seoul National University, Seoul, Korea. Previously, he was an assistant professor in the Department of Computer Engineering, Pusan National University, Pusan, Korea, from 1989 to 1992 and a visiting scientist at the IBM T.J. Watson Research Center, Yorktown Heights, New York, from 1989 to 1990. He has served on a number of program committees of technical conferences and workshops, including the IEEE Real-Time Systems Symposium (RTSS), the IEEE Real-Time Technology and Applications Symposium (RTAS), the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), and the International Conference on Distributed Computing Systems (ICDCS). He is also a member of the editorial board of the *IEEE Transactions on Computers*. His research interests include computer architecture, real-time computing, parallel processing, and computer performance evaluation. Dr. Min is a member of the IEEE and the IEEE Computer Society.

Yookun Cho received the BE degree from Seoul National University, Seoul, Korea, in 1971 and the PhD degree in computer science from the University of Minnesota at Minneapolis in 1978. He has been with the School of Computer Science and Engineering, Seoul National University since 1979, where he is currently a professor. He was a visiting assistant professor at the University of Minnesota during 1985 and a director of Educational and Research Computing Center at Seoul National University from 1993 to 1995. He is currently the president of the Korea Information Science Society. He was a member of the program committee of the IPPS/SPDP'98 in 1997 and the International Conference on High Performance Computing from 1995 to 1997. His research interests include operating systems, algorithms, system security, and fault-tolerant computing systems. He is a member of the IEEE and the IEEE Computer Society.

Chong Sang Kim received the PhD degree in electronic engineering from Seoul National University. He is currently a professor in the School of Computer Science and Engineering at Seoul National University, where he has been since 1977. His research interests are in computer architecture and computer networks. He is a senior member of the IEEE.