An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme

Jongmoo Choi† Sam H. Noh‡ Sang Lyul Min† Yookun Cho†

†*Department of Computer Engineering*
Seoul National University
Seoul 151-742, Korea
http://ssrnet.snu.ac.kr/~{choijm,cho}
http://archi.snu.ac.kr/~symin

‡*Department of Computer Engineering*
Hong-Ik University
Seoul 121-791, Korea
http://www.cs.hongik.ac.kr/~noh

Abstract

In this paper, we propose a new adaptive buffer management scheme called DEAR (DEtection based Adaptive Replacement) that automatically detects the block reference patterns of applications and applies different replacement policies to different applications based on the detected reference pattern. The proposed DEAR scheme uses a periodic process. Detection is made by associating block attribute values such as backward distance and frequency gathered at the $(i-1)$-th invocation with forward distances of blocks referenced between the $(i-1)$-th and i-th invocations. We implemented the DEAR scheme in FreeBSD 2.2.5 and measured its performance using several real applications. The results show that compared with the LRU buffer management scheme, the proposed scheme reduces the number of disk I/Os by up to 51% (with an average of 23%) and the response time by up to 35% (with an average of 12%) in the case of single application executions. For multiple applications, the proposed scheme reduces the number of disk I/Os by up to 20% (with an average of 12%) and the overall response time by up to 18% (with an average of 8%).

1 Introduction

The speed gap between processors and disks continues to increase as VLSI technologies advance at an enormous rate. This speed gap has resulted in disk I/O becoming a serious performance bottleneck for many computer systems [1, 2]. Hence, the role of the buffer cache located in main memory and managed by the operating system is becoming increasingly important. Judicious use of the buffer cache can improve the response time of individual applications and also the throughput of the system by reducing the number of disk I/Os. To this end, study of effective block replacement policies has been the focus of much research both in the systems and database areas [3, 4, 5, 6, 7, 8].

There also have been a number of studies on predicting future access for prefetching purposes [9, 10, 11]. They make use of past access history to predict files or blocks that are likely to be referenced in the near future. This prediction information is used to issue prefetch requests in order to hide disk I/O latency.

Recently, buffer management schemes based on user-level hints such as application-controlled file caching [12] and informed prefetching and caching [13] have been proposed. User-level hints in these schemes provide information about which blocks are good candidates for replacement, allowing different replacement policies to be applied to different applications.

However, to obtain user-level hints, users need to accurately understand the characteristics of block reference patterns of applications. This requires considerable effort from users limiting its applicability. For a sequential reference pattern, a simple heuristic method can be used to detect the pattern and the MRU replacement policy can be used to improve the buffer cache performance [14]. Such hints can also be obtained by the compiler for implicit I/Os to manage paged virtual memory [15]. As we will see later, our proposed approach is complementary to this approach since our approach is targeted for explicit I/Os.

In this paper, we propose a new buffer management scheme that we call DEAR (DEtection based Adaptive Replacement). Without any user intervention, the DEAR scheme detects the reference pattern of each application and classifies the pattern as sequential, looping, temporally-clustered, or probabilistic. After the detection, the scheme applies an appropriate replacement policy to the application. As the reference pattern of an application may change during its execution, the DEAR scheme periodically detects the reference pattern and applies a different replacement policy, if necessary.

We implemented the DEAR scheme in FreeBSD 2.2.5 and evaluated its performance with several real applications. The scheme is implemented at the kernel level without any modification to the system call interface, so the applications may run as-is. Performance measurements with real applications show that in the case of single application executions the DEAR scheme reduces the number of disk I/Os by up to 51% (with an average of 23%) and the response time by up to 35% (with an average of 12%), compared with the LRU buffer management scheme in FreeBSD. For multiple applications, the reduction in the number of disk I/Os is by up to 20% (with an average of 12%) while the reduction in the overall response time is by up to 18% (with an average of 8%).

We also compared the performance of the DEAR scheme with that of application-controlled file caching [12] through trace-driven simulations with the same set of application traces used in [12]. The results showed that the DEAR scheme without any use-level hints performs comparably to application-controlled file caching for the traces considered.

The rest of the paper is organized as follows. In Section 2, we explain the DEAR scheme in detail. Then, we describe the implementation of the DEAR scheme in FreeBSD in Section 3. In Section 4, we evaluate the performance of the DEAR scheme. Finally, we conclude with a summary and a discussion of future work in Section 5.

2 The DEAR Scheme

Recent research has shown that most applications show regular block reference patterns and that these patterns vary depending on the nature of the application. For example, a large class of scientific applications show a looping reference pattern where blocks are referenced repeatedly with regular intervals [16]. On the other hand, many database applications show a probabilistic reference pattern with different probabilities for index blocks and data blocks [17]. Unix applications tend to show either a sequential or a temporally-clustered reference pattern [12, 18]. Applications that deal with continuous media generally show a sequential or a looping reference pattern [19].

From these observations, we classify an application's reference pattern into one of the following: sequential, looping, temporally-clustered, or probabilistic reference pattern [20]. In the proposed DEAR scheme, the detection of an application's reference pattern is made by associating attributes of blocks with their forward distances, which are defined as the time intervals between the current time and the times of the next references. An attribute of a block can be anything that can be obtained from its past reference behavior including backward distance, frequency, inter-reference gap (IRG) [6], and k-th backward distance [4]. In this paper, we consider only two block attribute types: backward distance, which is the time interval between the current time and the time of the last reference[1], and frequency, which is the number of past references to the block.

The detection is performed by a monitoring process that is invoked periodically. At the time of its i-th invocation (we denote this time by m_i), the monitoring process calculates the forward distances (as seen from the standpoint of m_{i-1}) of the blocks referenced between m_{i-1} and m_i. From the block attribute values of those blocks, also as seen from the standpoint of m_{i-1}, the monitoring process builds two ordered lists using those blocks, one according to backward distance and the other according to frequency. Each ordered list is divided into a fixed number of sublists of equal size. Based on the relationship between the attribute value of each sublist and the average forward distance of blocks in the sublist, the block reference pattern of the application is deduced.

[1]In this paper, we assume that the (virtual) time is incremented at each block reference.

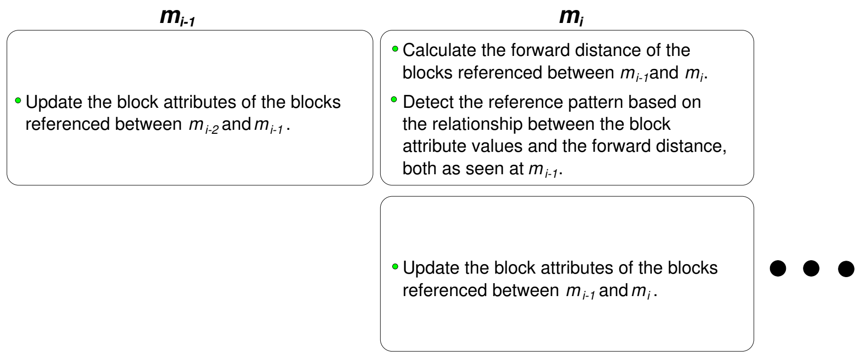


Figure 1: Detection process: two-stage pipeline with one-level look-behind.

After the detection, the block attributes of the blocks referenced between m_{i-1} and m_i are updated for the next detection at m_{i+1}. As shown in Figure 1, the detection process is essentially a two-stage pipeline with one-level look-behind since the detection at m_i is made based on the relationship between the block attribute values and the forward distance at m_{i-1}.

As an example, consider Figure 2. Assume that the period of the monitoring process (i.e., detection period) is 10 as measured in the number of block references made by the associated application. Also assume that between $m_{i-1} = 40$ and $m_i = 50$, blocks b_4, b_2, b_6, b_{12}, b_4, b_8, b_{11}, b_6, b_4, and b_6 were referenced in the given order (see Figure 2-(b)). Note that there are 10 block references since the detection period is 10. Finally, assume that at m_{i-1} the backward distance and frequency of the six distinct blocks b_4, b_2, b_6, b_{12}, b_8, b_{11} were 15, 12, 25, 4, 20, 9 and 6, 4, 5, 2, 1, 1, respectively (see Figure 2-(a)). Note that these distinct blocks have forward distances of 1, 2, 3, 4, 6, 7, respectively as seen at m_{i-1}. From the information about the block attribute values and the forward distance as seen at m_{i-1}, at m_i the DEAR scheme constructs two ordered lists, one according to backward distance and the other according to frequency (see Figure 2-(c)). Each list is divided into a number of sublists of equal size (3 sublists of size 2, in this example). Then various rules for detecting reference patterns, which are explained below, are applied to the two lists. In this particular example, blocks with higher frequency have smaller forward distance, which allows us to deduce that the block reference pattern of the given application follows a probabilistic reference pattern. The detection rules for all the reference patterns we consider are as follows:

Sequential Pattern: A sequential reference pattern has the property that all blocks are referenced one after the other and never referenced again. In this pattern, the average forward distance of all the sublists is ∞. Therefore, a reference pattern is sequential if $\mathbf{Avg\_fd}(sublist_1^{bd}) = \mathbf{Avg\_fd}(sublist_2^{bd}) = \cdots = \mathbf{Avg\_fd}(sublist_1^{fr}) = \mathbf{Avg\_fd}(sublist_2^{fr}) = \cdots = \infty$ where $sublist_i^{bd}$ and $sublist_i^{fr}$ are the i-th sublist for the backward distance and frequency block attribute types, respectively, and $\mathbf{Avg\_fd}(sublist)$ is the average forward distance of blocks in $sublist$.

Looping Pattern: A looping reference pattern has the property that blocks are referenced repeatedly with a regular interval. In this pattern, a block with a larger backward distance has a smaller forward distance. Therefore, a reference pattern is looping if the following relationship holds: if $i < j$ then $\mathbf{Avg\_fd}(sublist_i^{bd}) > \mathbf{Avg\_fd}(sublist_j^{bd})$.

Temporally-clustered Pattern: A temporally-clustered reference pattern has the property that a block referenced more recently will be referenced sooner in the future. Thus, a block with a smaller backward distance has a smaller forward distance. Therefore, a reference pattern is temporally-clustered if the following relationship holds: if $i < j$ then $\mathbf{Avg\_fd}(sublist_i^{bd}) < \mathbf{Avg\_fd}(sublist_j^{bd})$.

Probabilistic Pattern: A probabilistic reference pattern has a non-uniform block reference behavior that can be modeled by the Independent Reference Model (IRM) [21]. Each block b_i has a stationary probability p_i and all blocks are independently referenced with the associated probabilities. Under the stationary and independent condition, the expected forward

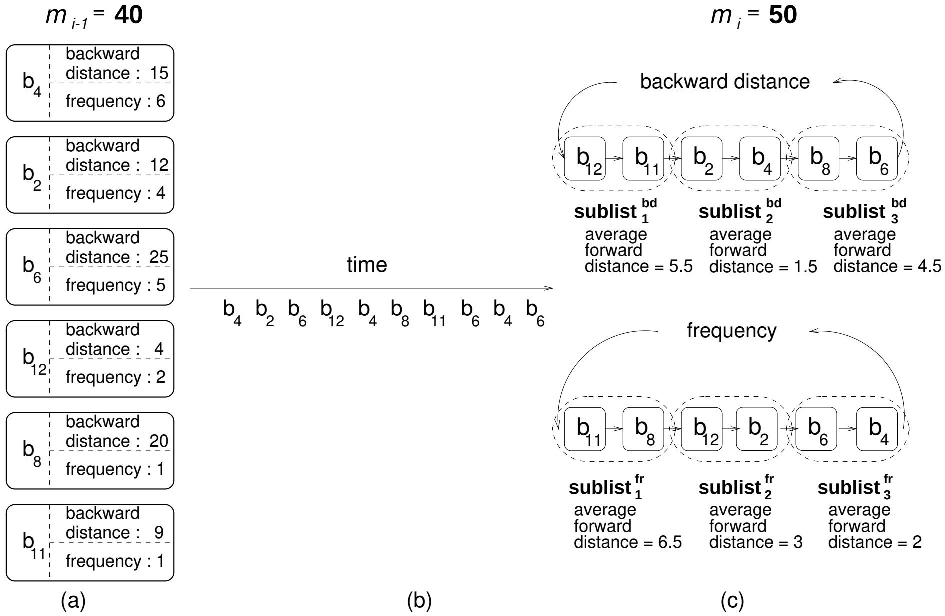


Figure 2: Example of block reference pattern detection.

distance of b_i is proportional to $1/p_i$. Thus, a block with a higher frequency has a smaller forward distance. Therefore, a reference pattern is probabilistic if the following relationship holds: if $i < j$ then $\mathbf{Avg\_fd}(sublist_i^{fr}) > \mathbf{Avg\_fd}(sublist_j^{fr})$.

In the DEAR scheme, different replacement policies are used for different applications depending on the detected reference pattern. For the sequential and looping reference patterns, the MRU replacement policy is used where the block with the smallest backward distance is always selected for replacement. For the temporally-clustered reference pattern, the LRU replacement policy, which replaces the block with the largest backward distance, is used. Finally, for the probabilistic reference pattern, the LFU replacement policy that replaces the block with the lowest reference frequency is used.

3 Implementation of the DEAR Scheme in FreeBSD

Figure 3 shows the overall structure of the buffer cache manager for the DEAR scheme as implemented in FreeBSD 2.2.5. The DEAR scheme applies different replacement policies for different applications. This requires a split of the buffer cache management module into two parts, one for block allocation and the other for block replacement. The module responsible for block allocation is the System Cache Manager (SCM). There is one SCM in the system. The module responsible for block replacement is the Application Cache Manager (ACM). There is one ACM for each application. This organization is similar to that proposed for application-controlled file caching [12]. Both of the modules are located in the VFS (Virtual File System) layer and collaborate with each other for buffer allocation and block replacement.

An ACM is allocated to each process when the process is forked. When a block is referenced from the process, the associated ACM is called by the *bread()* or *bwrite()* procedure in the SCM (1) to locate the information about the referenced block using a hash table, (2) to update the block attribute that is changed by the current reference, (3) to place the block into a linked list that maintains the blocks referenced in the current detection period, and (4) to adjust the replacement order according to the application-specific replacement policy. To maintain the replacement order, the current implementation uses the linked list data structure for the LRU and MRU replacement policies and the heap data structure for the LFU replacement policy.

After the steps (1)-(4) are performed, a check is made to see whether the current detection period

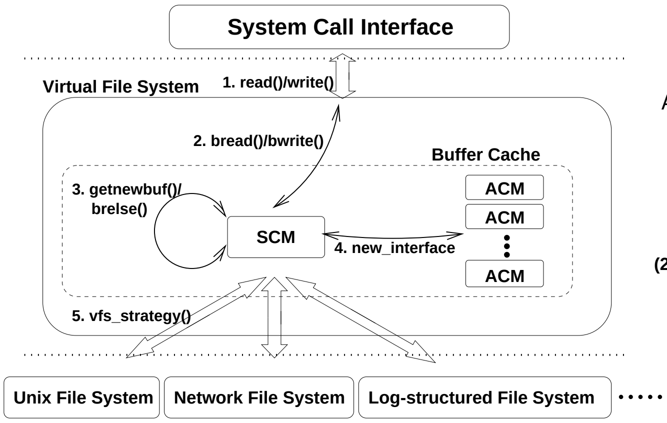


Figure 3: Overall structure of the DEAR scheme in FreeBSD 2.2.5.

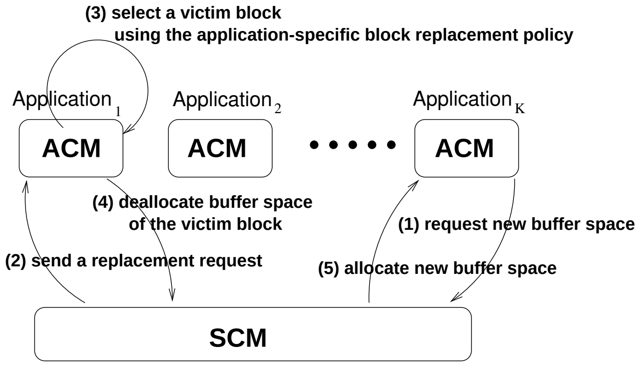


Figure 4: Interaction between ACM and SCM.

is over. If so, the monitoring process explained in the previous section is invoked to detect the application's reference pattern. The detected reference pattern dictates the replacement policy of the ACM. If none of the detection conditions previously explained is satisfied, the default LRU replacement policy is used.

The structure of information maintained for each block by the ACM is `<vnode #, block #, backward distance, frequency, forward distance, hp, bp, fp, cp>`. The pointer `hp` is used to place the block into the hash table that is used to locate the information about the currently referenced block. The pointers `bp` and `fp` are used to place the block into the ordered lists for the backward distance and frequency block attribute types, respectively, which are constructed when the monitoring process is invoked. Finally, the pointer `cp` is used to place the block into the list of blocks referenced in the current detection period. This data structure is the main space overhead of the DEAR scheme.

The main time overhead of the DEAR scheme is that needed to order the blocks according to each block attribute value, which has an $O(n \log n)$ time complexity where n is the number of distinct blocks referenced in the detection period. This operation is invoked once at the end of each detection period for each block attribute type. Other time overheads include those needed to calculate the forward distance, backward distance, and frequency of blocks at the end of each detection period, which has a time complexity of $O(n)$ where n is the number of distinct blocks referenced in the detection period.

The ACM and SCM interact with each other as depicted in Figure 4. When an application misses in the buffer cache, the ACM for the application makes a request to the SCM for additional buffer space (step (1) in Figure 4). If the SCM does not have any free buffer space, it sends a replacement request to one of the ACMs (step (2)). This operation is performed in the *getnewbuf()* procedure in the SCM, and the selected ACM is the one associated with an application whose current reference pattern is sequential. If there is no such application, the SCM simply chooses the ACM of the application with the global LRU block. The selected ACM decides the victim block to be replaced using its current replacement policy (step (3)) and deallocates its space to the SCM (step (4)). The SCM allocates this space to the ACM that requested the space (step (5)).

4 Performance Evaluation

In this section, we present the results of the performance evaluation of the DEAR scheme. We first describe the experimental setup. Then, we give the results of reference pattern detection followed by the performance measurement results for both single applications and multiple applications. We also give results from sensitivity analysis for different cache sizes and detection periods. Finally, we compare the performance of the DEAR scheme with that of application-controlled file caching [12] through trace-driven simulations with the same set of application traces used in [12].

Table 1: Characteristics of the applications.

Application	Description	Input data (MB)
cscope	C examination tool	C code (9)
glimpse	information retrieval tool	text files (50)
sort	UNIX sort utility	text files (4.5)
link	UNIX link editor	object files (2.5)
cpp	C preprocessor	C code (11)
gnuplot	GNU plotting utility	numeric data (8)
postgres1	relational DB system	two relations
postgres2	relational DB system	four relations

4.1 Experimental Setup

The experiments were conducted with FreeBSD 2.2.5 on a 166MHZ Intel Pentium PC with 64MB RAM and a 2.1GB Quantum Fireball hard disk. The applications we used are described below and are summarized in Table 1.

cscope Cscope is an interactive C-source examination tool. It creates an index file named *cscope.out* from C sources and answers interactive queries like searching C symbols or finding specific functions or identifiers. We used cscope on kernel sources of roughly 9MB in size and executed queries that search for five literals.

glimpse Glimpse is a text information retrieval utility. It builds indexes for words and allows fast searching. Text files of roughly 50MB in size were indexed resulting in about 5MB of indexes. A search was done for lines that contain the keywords *multithread, realtime, DSM, continuous media,* and *diskspace.*

sort Sort is a utility that sorts lines of text files. A 4.5MB text file was used as input, and this file was sorted numerically using the first field as the key.

link Link is the UNIX link-editor. We used this application to build the FreeBSD kernel from about 2.5MB of object files.

cpp Cpp is the GNU C-compatible compiler preprocessor. The kernel source was used as input with the size of header files and C-source files of about 1MB and 10MB, respectively.

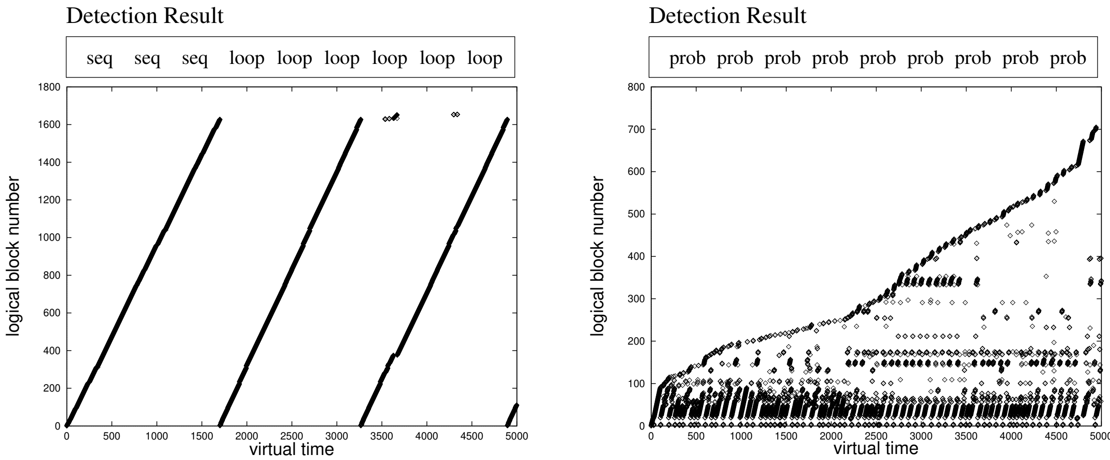
gnuplot Gnuplot is a command-line driven interactive plotting program. Using 8MB raw data, we plotted three-dimensional plots four times with different points of view.

postgres1 and postgres2 Postgres is a relational database system from the University of California at Berkeley. PostgresSQL version 6.2 and relations from a scaled-up Wisconsin benchmark such as thoustup and tenthoustup were used. Postgres1 is a join between the hundredthoustup and twohundredthoustup relations while postgres2 is a join among four relations, namely, fivehundredup, twothoustup, twentythoustup, and twohundredthoustup. The sizes of fivehundredup, twothoustup, twentythoustup, hundredthoustup, and twohundredthoustup are approximately 50KB, 150KB, 1.5MB, 7.5MB, and 15MB, respectively.

4.2 Detection Results

Figure 5 shows the results of the detection by the DEAR scheme for the cscope and cpp applications. In each graph, the x-axis is the virtual time and the y-axis is the logical block numbers of those referenced at the given time. The detection results are given at the top of the graph assuming a detection period of 500 references. For cscope, the DEAR scheme initially detects a sequential reference pattern but changes its detection to a looping reference pattern after the sequentially referenced blocks are re-accessed. This results from cscope always reading the file *cscope.out* sequentially whenever it receives a query about the C source. For cpp, the DEAR scheme detects a probabilistic reference pattern throughout the execution since as we can see from the graph, some blocks are more frequently accessed than others. This reference pattern results from the characteristic of cpp that header files are more frequently referenced than C files.

Figure 6 shows the detection results of the other applications. Although the result shows that the DEAR scheme performs reasonably well for the other applications, it also reveals the limitation of the current DEAR scheme, notably for the sort and postgres2 applications. They have either parallel or nested reference streams, which indicates a need for the proposed DEAR scheme to address more general reference patterns with arbitrary control structures such as parallel, sequence, and nested.



(a) cscope (b) cpp

Figure 5: Block reference patterns and detection results for cscope and cpp.

4.3 Performance Measurements: Single Applications

We compared the performance of each application under the DEAR scheme with not only that under the LRU scheme in FreeBSD but also with those under the LFU and MRU schemes. For this purpose, we implemented the DEAR scheme as well as the LFU and MRU schemes in FreeBSD. We measured both the number of disk I/Os and the response time of each application for a 6MB buffer cache with block size set to 8KB. For the DEAR scheme, we set the length of the detection period to 500 and the number of sublists in the ordered lists to 5 for both the backward distance and frequency block attribute types. The performance of the DEAR scheme for different cache sizes, different detection periods, and different numbers of sublists in the ordered lists is discussed in Section 4.5.

Figure 7 shows the number of disk I/Os and the response time of the four schemes. The values reported here are the average of three separate executions and before each execution, the system was rebooted. From the results we observe the following:

- The DEAR scheme performs almost as good as the best of the other three schemes for all the applications we considered. Also, when compared with the LRU scheme in FreeBSD, the number of disk I/Os is reduced by up to 51% (for the cscope application) with an average of 23% and the response time by up to 35% (also for the cscope application) with an average of 12%.

- For the link application, there is no performance difference among the four schemes. This is because the input data to the link application is small (2.5MB), and thus all the blocks reside in the buffer cache after they are initially loaded.

- Postgres1 and postgres2 do not show as much improvement in the response time as that in the number of disk I/Os when using the DEAR scheme. This is because of the constant synchronization between the client (the psql utility that provides the user interface) and the server (the postgres process that performs the query processing and database management). For the gnuplot application, much time was spent for user mode computation and thus reduction in the number of disk I/Os also has a limited impact on the response time.

- Except for the above three applications, the ratio between the reduction in the number of disk I/Os and that in the response time is consistent. This indicates that the DEAR scheme incurs little extra overhead compared to those in the other schemes.

The last point is more evident in Figure 8 where the response time is divided into three components: I/O stall time, system time, and user time. For the LRU scheme of FreeBSD, the system time consists of VFS processing time, buffer cache manage-

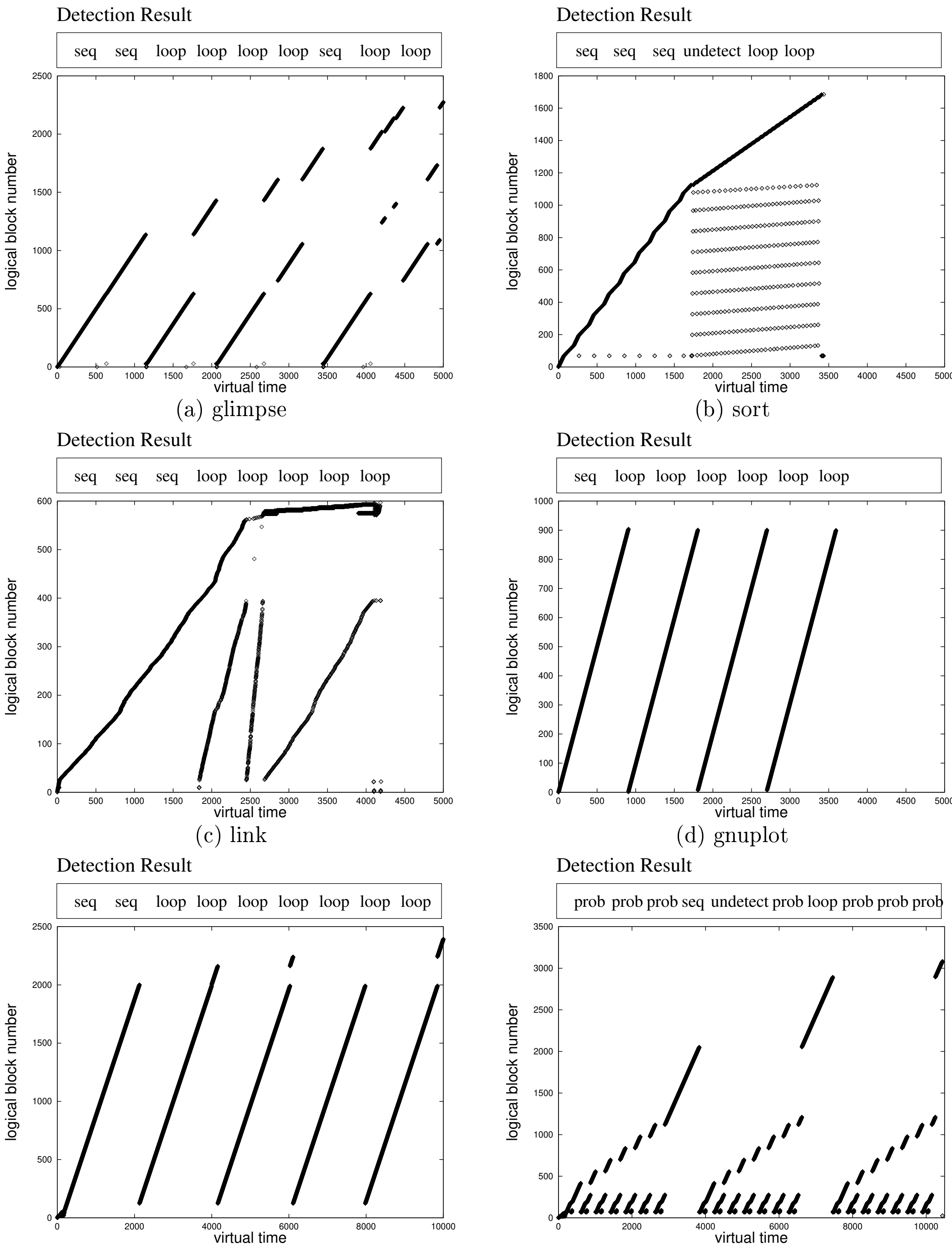
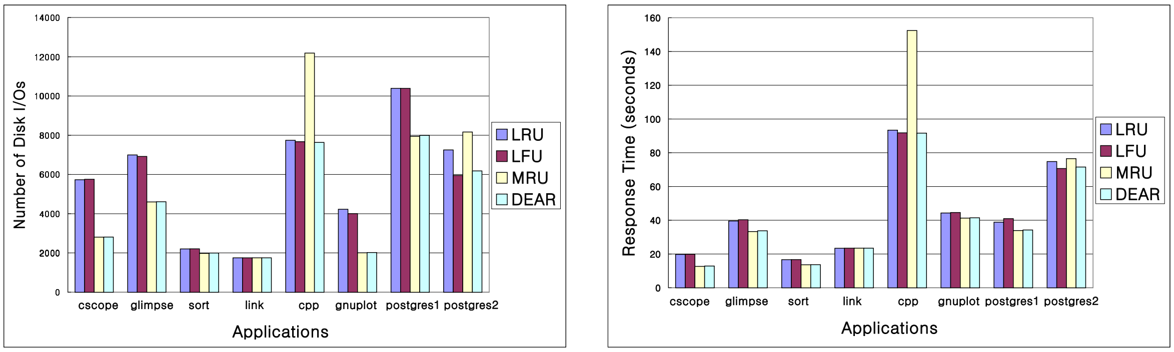


Figure 6: Block reference patterns and detection results for the other applications.



(a) Number of Disk I/Os

(b) Response Time

Figure 7: Single application performance.

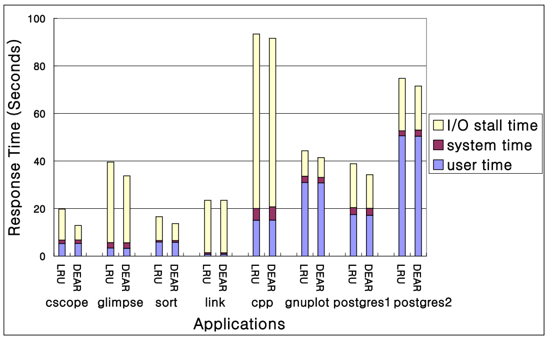


Figure 8: Decomposition of response time.

ment time, disk driver processing time, disk interrupt handling time and, data copy time from buffer cache to user space. On top of those, the DEAR scheme requires additional processing time such as those for sorting blocks according to block attribute values and maintaining block attribute values and forward distances. From Figure 8, we can notice that the system times of the two schemes are comparable meaning that the DEAR scheme incurs little additional overheads.

4.4 Performance Measurements: Multiple Applications

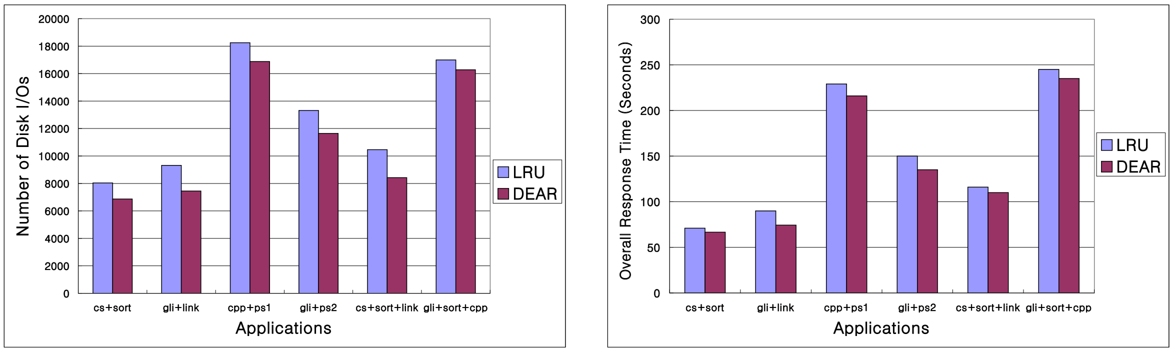
In real systems, multiple applications execute concurrently competing for limited buffer space. To test the DEAR scheme in such an environment, we ran several combinations of two or more of the applications with a buffer cache of 6MB and measured the total number of disk I/Os and the overall response time for both the DEAR scheme and the LRU scheme in FreeBSD. Again, we set the length of the detection period to 500 and the number of sublists in the ordered lists to 5.

The results in Figure 9 show that the number of disk I/Os is reduced by up to 20% (for the cscope+sort+link case) with an average of 12% and the overall response time by up to 18% (for the glimpse+link case) with an average of 8%.

In the multiple application case, there are two possible benefits from using the proposed DEAR scheme. The first is from applying different replacement policies to different applications based on their detected reference patterns. The second is from giving preference to blocks that belong to an application with the sequential reference pattern when a replacement is needed. To quantify these benefits, we performed an experiment where even the LRU replacement policy gives preference to blocks belonging to an application with the sequential reference pattern, which we call the LRU-SEQ replacement policy.

Table 2: Performance comparison between the LRU-SEQ and the DEAR schemes.

Scheme	Response Time (seconds)			
	cs+sort	gli+link	cs+wc	gli+wc
LRU	70.96	89.87	81.27	89.97
LRU-SEQ	70.72	87.24	71.75	86.88
DEAR	66.61	74.29	62.88	82.36



(a) Number of Disk I/Os

(b) Overall response time

Figure 9: Multiple application performance.

Table 2 shows the results of the LRU-SEQ scheme for the 6MB buffer cache size. In the case of cscope+sort and glimpse+link, there is little difference between the LRU and the LRU-SEQ schemes, since the reference pattern of the four component applications is not sequential in the steady state. Wc is a utility that displays the numbers of lines, words, and characters in a file. Its steady state reference pattern is sequential. Replacing sort and link with wc, produces a significant difference in the response time between the LRU and the LRU-SEQ schemes. This results from the LRU-SEQ scheme allocating more buffer space to cscope (or glimpse) by replacing blocks of the wc application earlier than the usual LRU order. Still, there is a substantial difference in the response time between the LRU-SEQ scheme and the DEAR scheme indicating that the benefit from applying different replacement polices tailored for different applications is significant.

4.5 Sensitivity Analysis

4.5.1 Cache Size

Tables 3 and 4 compare the performance of the DEAR scheme against the LRU scheme for various buffer cache sizes for the single and multiple application cases, respectively. The results from the single application case show that as long as the total number of distinct blocks accessed by an application is greater than the number of blocks in the buffer cache, there is a substantial difference in the response time between the DEAR and the LRU schemes. However, when the number of distinct blocks of an application is smaller than the number of blocks in the buffer cache, all the blocks are cached in the buffer cache and the two schemes show similar performance. The latter behavior is most visible for the link application that has the smallest number of distinct blocks (about 310 blocks). For link, the DEAR and the LRU schemes provide similar response times.

Table 3: Single application performance for various buffer cache sizes.

Application	Scheme	Response Time (seconds)			
		2MB	4MB	6MB	8MB
cscope	DEAR	16.99	14.90	12.87	11.17
	LRU	19.79	19.79	19.77	19.77
glimpse	DEAR	39.12	35.68	33.73	32.87
	LRU	40.70	39.72	39.55	37.49
sort	DEAR	18.16	15.54	13.60	12.10
	LRU	18.50	17.45	16.59	14.68
link	DEAR	28.19	23.38	23.38	23.38
	LRU	29.65	23.35	23.35	23.35
cpp	DEAR	132.94	94.42	91.61	91.36
	LRU	159.59	97.94	93.39	91.82
gnuplot	DEAR	43.54	42.26	41.39	41.19
	LRU	44.30	44.30	44.30	44.30
postgres1	DEAR	38.37	36.16	34.22	32.17
	LRU	39.72	38.91	38.82	38.76
postgres2	DEAR	74.57	72.51	71.15	68.45
	LRU	82.93	74.93	74.75	73.93

For the multiple application case, the case where the total number of distinct blocks accessed by the component applications is smaller than the number

Table 5: The effect of the detection period on the performance of the DEAR scheme for the single application case.

Scheme	Detection Period	Response Time (seconds)						
		cscope	glimpse	sort	cpp	gnuplot	postgres1	postgres2
DEAR	100	12.85	33.70	13.72	98.81	40.92	34.62	76.56
	250	12.79	33.68	13.30	91.54	40.93	34.13	72.30
	500	12.87	33.73	13.60	91.61	41.39	34.22	71.15
	1000	13.52	36.26	13.88	91.78	41.66	34.53	72.41
	2000	15.20	36.45	15.77	91.99	42.36	34.84	72.53
LRU	N/A	19.77	39.55	16.59	93.39	44.30	38.82	74.75

Table 4: Multiple application performance for various buffer cache sizes.

Applications	Scheme	Response Time (seconds)			
		4MB	6MB	8MB	10MB
cs+sort	DEAR	70.4	66.6	62.9	53.5
	LRU	71.5	70.9	69.9	67.3
gli+link	DEAR	79.1	74.2	71.6	70.1
	LRU	94.5	89.8	79.1	77.9
cpp+ps1	DEAR	222.2	216.7	209.8	202.4
	LRU	236.5	229.9	226.6	226.1
gli+ps2	DEAR	145.6	139.8	132.5	128.3
	LRU	165.5	155.2	146.7	138.8
cs+sort+link	DEAR	116.1	112.7	106.7	101.8
	LRU	121.3	118.0	112.8	105.3
gli+sort+cpp	DEAR	245.9	235.5	215.8	207.2
	LRU	246.3	245.3	225.9	222.9

of blocks in the buffer cache does not occur and the DEAR scheme shows consistently better performance than the LRU scheme.

4.5.2 Detection Period and the number of Sublists

Determining the length of the detection period is an important design issue that requires a trade-off. If the detection period is too long, the scheme will not be adaptive to possible changes of the reference pattern within a detection period. On the other hand, if the period is too short, the scheme would incur too much overhead to be practical. Moreover, if the period is too short, a short burst of references may mislead the detection. For example, a probabilistic reference pattern may be mistaken for a looping reference pattern when a small number of blocks are repeatedly accessed over two detection periods while satisfying the detection condition for a looping reference pattern.

The above trade-off relationship is evident in Table 5 that gives the response time of all but the link application as the detection period varies from 100 to 2000. We exclude the link application since as we mentioned earlier all of its blocks fit into the buffer cache. Thus different detection periods do not make any difference. For most applications, the best performance was obtained when the detection period is either 250 or 500. The results also show that even with detection periods that are considerably smaller or larger than these optimal values, the DEAR scheme performs better than the LRU scheme in FreeBSD. The exceptions are with the cpp and postgres2 applications when the detection period is 100. In the two cases, the performance degradation is considerably larger than the others at the detection period of 100. A careful inspection of the results revealed that when the detection period is 100 the DEAR scheme mistakenly detects both applications to have a looping reference pattern when in reality it was part of a probabilistic reference pattern. The multiple application case shows a similar effect of the detection period on the performance as we can see in Table 6.

The number of sublists used in the detection process can also affect the detection results of the DEAR scheme. Table 7 gives the detection results of the DEAR scheme as the number of sublists increases from three to seven. From the results, we can notice that the number of sublists hardly affects the detection results although there is a slight increase in the number of undetected cases as the number of sublists increases due to a more strict detection rule. Remember that to detect a reference pattern the associated detection rule should be held for *all* the sublists.

Table 6: The effect of the detection period on the performance of the DEAR scheme for the multiple application case.

Scheme	Detection Period	Response Time (seconds)					
		cs+sort	gli+link	cpp+ps1	gli+ps2	cs+sort+link	gli+sort+cpp
DEAR	100	66.68	73.54	236.29	144.67	108.86	251.54
	250	65.84	73.41	216.62	136.86	108.62	230.61
	500	66.61	74.29	216.73	139.88	112.73	235.56
	1000	67.34	74.99	216.91	139.24	116.30	238.70
	2000	68.70	81.69	219.38	139.34	116.84	241.41
LRU	N/A	70.96	89.87	229.99	155.27	118.03	245.34

Table 7: The effect of the number of sublists on the detection results of the DEAR scheme.

Application	Detection Results		
	Number of sublists = 3	Number of sublists = 5	Number of sublists = 7
cscope	seq[3],loop[8]	seq[3],loop[8]	seq[3],loop[8]
glimpse	seq[4],loop[8]	seq[4],loop[8]	seq[3],loop[9]
sort	seq[3],loop[3]	seq[3],loop[2],undetect[1]	seq[3],loop[2],undetect[1]
link	seq[3],loop[5]	seq[3],loop[5]	seq[3],loop[5]
cpp	prob[18]	prob[18]	prob[13],undetect[5]
gnuplot	seq[1],loop[6]	seq[1],loop[6]	seq[1],loop[6]
postgres1	seq[5],loop[16]	seq[5],loop[16]	seq[5],loop[16]
postgres2	prob[13],loop[5],seq[2],undetect[1]	prob[12],loop[4],seq[2],undetect[3]	prob[11],loop[3],seq[2],undetect[5]

4.6 Comparison with Application-controlled File Caching

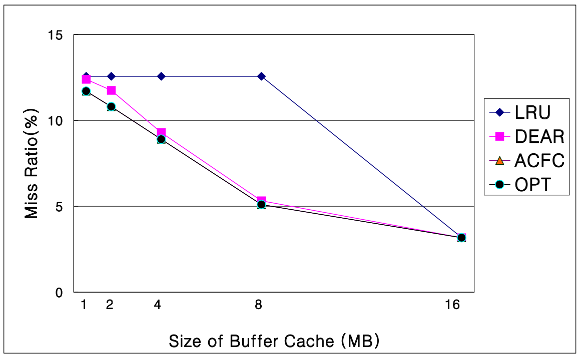
To compare the performance of the DEAR scheme with that of application-controlled file caching (ACFC) [12], we performed trace-driven simulations with the same set of three application traces used in [12]. The characteristics of the three applications and their traces can be found in [12]. Figure 10 shows the miss ratio of the three applications for the LRU, ACFC, DEAR, and OPT (off-line optimal) schemes when cache size increases from 1MB to 16MB. The results for the LRU, ACFC, and OPT schemes were borrowed from [12] and those for the DEAR scheme were obtained by simulating the DEAR scheme with detection period equal to 500 and the number of sublists in the ordered list equal to 5 for both backward distance and frequency block attribute types. The results show that the miss ratio of the DEAR scheme is comparable to that of the ACFC scheme, which utilizes user-level hints to guide the replacement decisions. The small difference between the two schemes results from the misses that occur before the DEAR scheme has a chance to detect the reference pattern.

5 Conclusions and Future Work

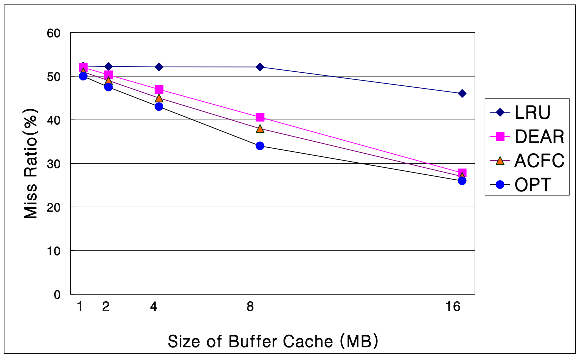
In this paper, we proposed a new buffer management scheme called DEAR (DEtection based Adaptive Replacement) that automatically detects the block reference pattern of applications as sequential, looping, temporally-clustered, or probabilistic without any user intervention. Based on the detected reference pattern, the proposed DEAR scheme applies an appropriate replacement policy to each application.

We implemented the DEAR scheme in FreeBSD 2.2.5 and measured its performance using several real applications. The results showed that compared with the buffer management scheme in FreeBSD the proposed scheme reduces the number of disk I/Os by up to 51% (with an average of 23%) and the response time by up to 35% (with an average of 12%) in the case of single application executions. For multiple applications, the reduction in the number of disk I/Os is by up to 20% (with an average of 12%) while the reduction in the overall response time is by up to 18% (with an average of 8%).

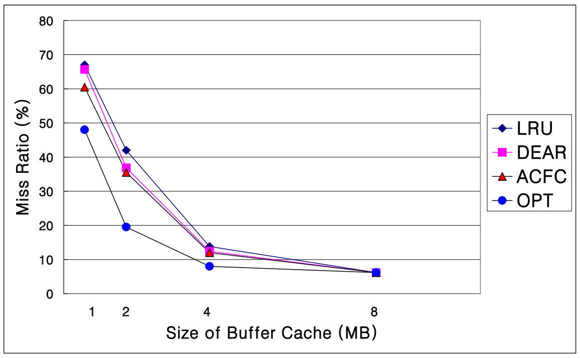
We also compared the performance of the DEAR scheme with that of application-controlled file caching [12] through trace-driven simulations with



(a) cscope



(b) linking kernel



(c) Postgres

Figure 10: Comparison with application-controlled file caching.

the same set of application traces used in [12]. The results showed that the DEAR scheme performs comparably to application-controlled file caching for the traces considered.

As we noted in Section 4.2, some applications have block reference behavior that cannot be characterized by a single reference pattern. One direction for future research is to extend the current DEAR scheme so that it can detect more complex reference patterns with parallel, sequence, and nested structures as well as to develop appropriate replacement policies for them. Another direction for future research is to study more advanced buffer allocation strategies for the DEAR scheme than the simple strategy explained in Section 3. A good buffer allocation strategy for the DEAR scheme should reward more to applications with larger reductions in the number of disk I/Os while preventing any one application from monopolizing the buffer space. Other directions for future research include applying the detection capability of the DEAR scheme to prefetching and considering block attribute types other than backward distance and frequency.

All the source code for the DEAR scheme and the applications run for the experiments can be found at `http://ssrnet.snu.ac.kr/~choijm/cache.html`.

Acknowledgments

We would like to thank Pei Cao, Edward W. Felten, and Kai Li for providing us with the traces we used in the experiments. We also would like to thank Yoonho Park, our shepherd, for his help in improving this paper and Donghee Lee for many helpful discussions. Finally, we would like to thank anonymous reviewers for their constructive comments and suggestions. This work was supported in part by the Korea Research Foundation for the program year of 1998.

References

[1] A. J. Smith, "Disk cache-miss ratio analysis and design considerations," *ACM Transactions on Computer Systems*, vol. 3, pp. 161–203, August 1985.

[2] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, pp. 145–182, June 1994.

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

[4] 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.

[5] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the Existence of a Spectrum of Policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies." To appear in the Proceedings of the ACM SIGMETRICS'99, 1999.

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

[7] A. Dan and D. Sitaram, "A Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads," in *Proceedings of Multimedia Computing and Networking(MMCN) Conference*, pp. 344–351, 1996.

[8] C. Faloutsos, R. Ng, and T. Sellis, "Flexible and Adaptable Buffer Management Techniques for Database Management Systems," *IEEE Transactions on Computers*, vol. 44, pp. 546–560, April 1995.

[9] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching via Data Compression," in *Proceedings of the 1993 ACM SIGMOD Conference*, pp. 257–266, 1993.

[10] J. Griffioen and R. Appleton, "Reducing File System Latency using a Predictive Approach," in *Proceedings of the 1994 Summer USENIX Conference*, pp. 197–207, 1994.

[11] T. M. Kroeger and D. D. E. Long, "Predicting File System Action from Prior Events," in *Proceedings of the 1996 Annual USENIX Technical Conference*, pp. 319–328, 1996.

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

[13] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," in *Proceedings of the 15th Symposium on Operating System Principles*, pp. 1–16, 1995.

[14] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," in *Proceedings of the 1997 ACM SIGMETRICS Conference*, pp. 115–126, 1997.

[15] T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 3–17, 1996.

[16] B. K. Pasquale and G. C. Polyzos, "A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload," in *Proceedings of Supercomputing '93*, pp. 388–397, 1993.

[17] A. Dan, P. S. Yu, and J.-Y. Chung, "Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability," *VLDB Journal*, vol. 4, pp. 127–154, January 1995.

[18] 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 Symposium on Operating System Principles*, pp. 198–212, 1991.

[19] P. J. Shenoy, P. Goyal, S. S. Rao, and H. M. Vin, "Design and Implementation of Symphony: An Integrated Multimedia File System," in *Proceedings of ACM/SPIE Multimedia Computing and Networking(MMCN) Conference*, pp. 124–138, 1998.

[20] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "An Adaptive Block Management Scheme Using On-Line Detection of Block Reference Patterns," in *Proceeding of the Fourth IEEE International Workshop on Multi-Media Database Management Systems*, pp. 172–179, 1998.

[21] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.