Towards Application/File-level Characterization of Block References: A Case for Fine-Grained Buffer Management *

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

†School of Computer Science and 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

Two contributions are made in this paper. First, we show that system level characterization of file block references is inadequate for maximizing buffer cache performance. We show that a finer-grained characterization approach is needed. Though application level characterization methods have been proposed, this is the first attempt, to the best of our knowledge, to consider file level characterizations. We propose an Application/File-level Characterization (AFC) scheme where we detect on-line the reference characteristics at the application level and then at the file level, if necessary. The results of this characterization are used to employ appropriate replacement policies in the buffer cache to maximize performance. The second contribution is in proposing an efficient and fair buffer allocation scheme. Application or file level resource management is infeasible unless there exists an allocation scheme that is efficient and fair. We propose the ΔHIT allocation scheme that takes away a block from the application/file where the removal results in the smallest reduction in the number of expected buffer cache hits. Both the AFC and ΔHIT schemes are on-line schemes that detect and allocate as applications execute. Experiments using trace-driven simulations show that substantial performance improvements can be made. For single application executions the hit ratio increased an average of 13 percentage points compared to the LRU policy, with a maximum increase of 59 percentage points, while for multiple application executions, the increase is an average of 12 percentage points, with a maximum of 32 percentage points for the workloads considered.

*This work was supported by the Brain Korea 21 Project.

1. INTRODUCTION

Efficient use of the buffer cache is important in improving file system performance. Hence, characterization of cache block reference patterns and development of adequate replacement policies for such references have been studied intensively. A majority of the policies, ranging from simple and traditional policies such as the LRU and LFU to recent policies such as the LRU-K [13] or LRFU [11], have focussed their attention at optimizing the replacement for general system-wide reference patterns.

In this paper, we contend that finer-grained level of characterization of block references is needed to maximize file system performance. By finer-grained level, we refer to levels of characterization at the application level and/or individual file level. With more accurate characterizations, more adequate replacement policies may be employed leading to better overall system performance.

1.1 Motivation

Consider Figure 1, which shows the block references of a typical multiprogramming system. Specifically, three applications, namely, *cscope, cpp,* and *postgres2* (details of which will be discussed in Section 5) are executing concurrently, and the reference pattern is as observed from the file system level. The x-axis is the virtual time which ticks at each block access and the y-axis is the logical block number of that accessed at the given time. As can be observed, it is difficult to establish clear characteristics regarding the block reference pattern.

Now, consider Figure 2. This is exactly the same three applications mentioned above. However, the block references are observed at the application level. As we separate out the reference patterns of the three applications of Figure 1, we get the three graphs of Figure 2. The characteristics of these three reference patterns are much more distinguishable than those of Figure 1.

We can take this a step further and observe at even a finer level, that is, at the individual file level. Resulting reference patterns of files for the *postgres2* application (Figure 2(c)) are shown in Figure 3. Here one can clearly see that the same reference pattern is repeated over and over indicating

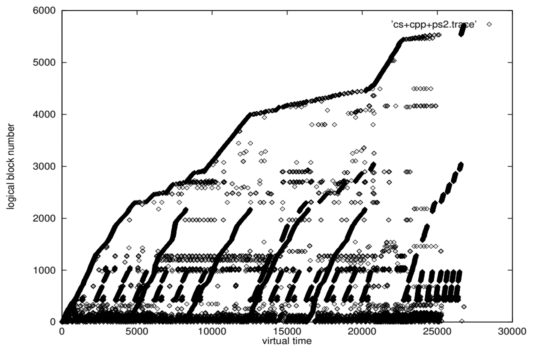
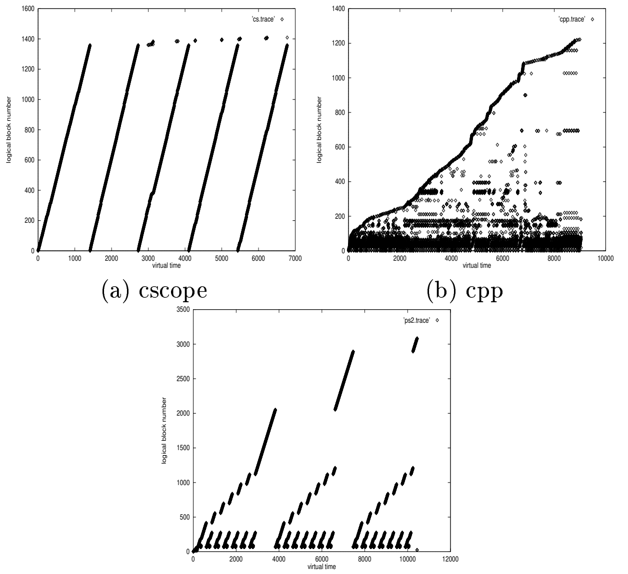


Figure 1: Block reference seen at the file system level.



(a) cscope (b) cpp

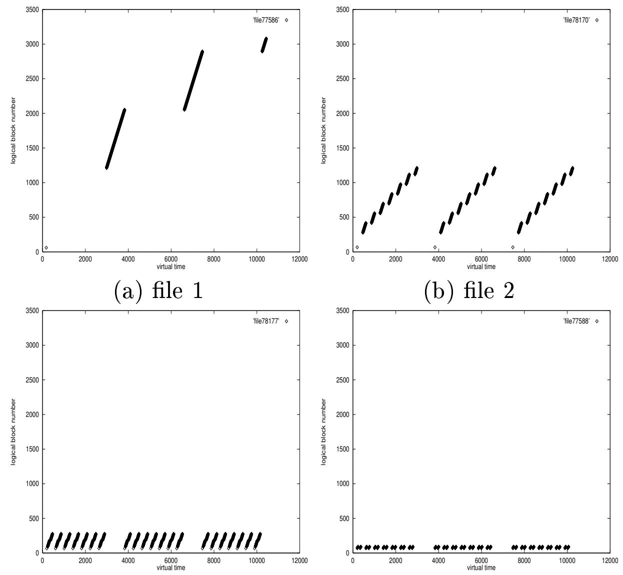
(c) postgres2

Figure 2: Block references seen at the application level.

that individual files are referenced repeatedly in a looping fashion.

As shown through these figures, the characteristics of reference patterns become more discernible as we explore at finer levels of reference. We need to exploit this to maximize the performance of the system. The first contribution of this paper is that we show that this exploitation is possible. We show that reference patterns can be monitored automatically and on-line, at levels needed to maximize performance. Trace-driven simulation results show that by using this technique performance benefits are considerable, allowing the buffer cache hit ratio to reach close to (and for particular cases reaching) the optimal for single application executions.

Another contribution of this paper is that we provide a fair and effective buffer allocation scheme. One problem in employing a replacement policy at such fine-grained levels is the issue of buffer allocation. In a general multiprogramming environment, multiple applications will vie for buffer space, and a fair and effective solution to this problem must be pro-



(a) file 1 (b) file 2

(c) file 3 (d) file 4

Figure 3: Block references of *postgres2* seen at the file level.

vided. To date, only limited research has been conducted on this matter (which we briefly discuss below). We propose the ΔHIT allocation scheme that takes away a block from the application/file where the removal results in the smallest reduction in the number of expected buffer cache hits. Results from trace-driven simulations show that the allocation scheme combined with exploiting the fine-grained characterization scheme can provide considerable performance benefits.

1.2 Related Works

Previous research related to this paper can be largely divided into two categories. One is that related to characterizing reference patterns and the other concerns buffer allocation.

There has been much research in characterizing or exploiting the characteristics of references for buffer management, both in the systems [5, 6, 10, 12, 18] and database arenas [8, 9]. For example, Glass and Cao investigated a means of automatically detecting and exploiting sequential reference patterns at the system level, which resulted in the SEQ replacement scheme [6]. Automatic detection by the compiler for implicit management of I/O at the virtual memory system has also been attempted [12]. Also, it was shown by Choi et al. that automatic detection of reference characteristics is feasible within the file system [5]. This attempt, however, only considers reference patterns at the individual application level, and does not consider references at the file level.

Research of exploiting characterization at the application level by providing user hints to the system has also been attempted by Cao et al. [3] and Patterson et al. [15]. Both of them provide buffer allocation strategies. We briefly discuss those strategies below.

Cao et al. [3] proposed an application-controlled file caching scheme that splits the responsibilities of allocation and replacement between the kernel and the user process. The kernel is responsible for allocating buffer blocks to processes, while each user process controls the replacement of blocks, based on user level hints, on its share of buffer blocks. The proposed allocation strategy called the LRU-SP (Swapping and Placeholders) essentially tries to replace the global LRU block among all the in-cache blocks. However, if the process holding that block chooses to replace a block other than that particular block based on user level hints, bookkeeping is done, so that the same process may not be victimized again as it still holds on to the global LRU block.

Patterson et al. [15] proposed an informed prefetching and caching scheme where they develop a framework for buffer management based on cost-benefit analysis. This framework essentially provides a means of balancing the benefits of caching and prefetching. Rather than considering the issue of per-application buffer allocation, buffer allocation is considered in terms of whether to prefetch a block or not, and when to do it, if so desired.

1.3 The Remainder of the Paper

The rest of the paper is organized as follows. In Section 2, we discuss the characteristics of block references that are relevant to this paper. In Section 3, we show how the notion of block attributes may be used to detect reference characteristics. We also show how this can be done on-line. The Application/File-level Characterization (AFC) scheme and the ΔHIT buffer allocation scheme are discussed in Section 4. In Section 5, we show the performance evaluation results. Finally, we conclude with a summary in Section 6.

2. BLOCK REFERENCE CHARACTERISTICS

Many studies in the past have shown that a majority of applications show some form of regularity in referencing file blocks. For example, a large class of scientific applications have a looping reference pattern where blocks are accessed repeatedly with regular intervals [14]. On the other hand, many database applications have a non-uniform reference pattern with different probabilities for index blocks and data blocks [8]. Unix applications tend to show either a sequential or a temporally-clustered reference pattern [2, 3], while Web clients and proxies tend to show non-uniform reference patterns with temporal locality [1]. Applications that deal with continuous media generally have a sequential or a looping reference pattern [17].

From these observations, we conclude that a reference pattern observable by the system at some particular time frame falls into one of the following four major types of patterns, namely, *sequential*, *looping*, *temporally-clustered*, and *probabilistic* reference patterns. In some situations, a reference pattern may fail to fall into any of the above four cases, and for these cases we simply categorize the pattern as an *undetected* pattern.

We now give definitions of the four reference patterns of interest. Given a set N $= \{b_1, b_2, \cdots, b_n\}$ of disk blocks, a *reference* of length w is a sequence $< r_1, r_2, \cdots, r_w >$ of blocks accessed in the given order, where $r_t = b_i$ $(1 \leq t \leq w, 1 \leq i \leq n)$ denotes that block b_i is accessed at a given instance t in the reference.

A sequential reference pattern is, then, a reference where all blocks are accessed one after another and never re-accessed. We define that a reference $< r_1, r_2, \cdots, r_w >$ is a sequential reference pattern if $r_i \neq r_j$ for all $1 \leq i \neq j \leq w$.

A looping reference pattern is a reference where all blocks are accessed repeatedly with a regular interval. Formally, a reference $< r_1, r_2, \cdots, r_w >$ is a looping reference pattern if for some $l \leq w$, $r_i \neq r_{i+j}$ for $1 \leq j < l$, and $r_{i+l} = r_i$ for $1 \leq i \leq w - l$. The subsequence $< r_1, ..., r_l >$ is called a *loop*, and the value l is called the *length* of the loop.

A temporally-clustered reference pattern is one where blocks accessed more recently are the ones more likely to be accessed in the future. This reference pattern can be characterized by the LRU stack model [7]. In this model, all blocks are ordered in the LRU stack by their last access time. When a block is accessed, it moves to the top position of the stack (position 1) and the blocks which were above it are pushed down one position (from position i to position i+1). Each position i of the LRU stack has its own reference probability a_i. When a reference $< r_1, r_2, \cdots, r_w >$ satisfies the condition $a_1 \geq a_2 \geq \cdots \geq a_n$, it is a temporally-clustered reference pattern.

A probabilistic reference pattern is a reference where each block has a stationary reference probability and all blocks are accessed independently with the associated probabilities. This reference pattern can be characterized by the Independent Reference Model (IRM) [7, 13]. In the model, each block b_i has its stationary and independent reference probability p_i. A reference $< r_1, r_2, \cdots, r_w >$ is a probabilistic reference if for all $1 \leq i, j \leq w$, r_i is non-uniformly distributed over the set of all blocks with stationary and independent reference probabilities.

3. DETECTING REFERENCE CHARACTERISTICS

In this section, we show how the reference characteristics, specifically in the form of the four reference patterns described in Section 2, are actually detected.

3.1 Using Block Attributes to Detect Reference Characteristics

An attribute of a block is information that can be obtained from the block's past reference behavior such as backward distance [7], frequency, inter-reference gap (IRG) [16], and k-th backward distance [13]. In this study, we only consider two block attributes: backward distance and frequency.

Definition 1. **Backward distance $\mathbf{bd_t(b_i)}$:** Given a reference $< r_1, r_2, \cdots, r_t, \cdots >$, the backward distance $bd_t(b_i)$ for block b_i at time t is the distance to the most recent reference to b_i up to time t:

$$\begin{aligned} bd_t(b_i) &= x, && \text{if } r_{t-x} = b_i \text{ and } r_j \neq b_i \text{ for } t - x < j \leq t \\ &= \infty, && \text{if } r_j \neq b_i \text{ for } 1 \leq j \leq t \text{ (that is, } b_i \text{ is not referenced prior to time } t). \end{aligned}$$

Definition 2. **Frequency $\mathbf{fr_t(b_i)}$:** Given a reference $< r_1, r_2, \cdots, r_t, \cdots >$, the frequency $fr_t(b_i)$ for block b_i at time

t is the number of references to b_i up to time t:

$$fr_t(b_i) = \sum_{j=1}^{t} \delta_j \text{ where } \delta_j = \begin{cases} 1 & \text{for } r_j = b_i \\ 0 & \text{for } r_j \neq b_i \end{cases} .$$

Detection of block reference patterns is made by associating block attributes with the forward distance of a block defined as follows.

Definition 3. **Forward distance $\mathbf{fd_t(b_i)}$:** Given a reference $< r_1, r_2, \cdots, r_t, \cdots >$, the forward distance $fd_t(b_i)$ for block b_i at time t is the distance to the first reference to b_i after time t:

$$\begin{aligned} fd_t(b_i) &= x, && \text{if } r_{t+x} = b_i \text{ and } r_j \neq b_i \text{ for } t \leq j < t+x \\ &= \infty, && \text{if } r_j \neq b_i \text{ for } j \geq t \text{ (that is, } b_i \text{ is not referenced after time } t). \end{aligned}$$

Based on the notion of block attributes and the forward distance, we derive rules for detecting the aforementioned block reference patterns.

Recall the definitions of the block reference patterns. In a sequential reference, all blocks are accessed one after another and never accessed again. Thus, the detection rule for the sequential reference pattern can be given as follows.

Detection Rule 1. A reference $< r_1, r_2, \cdots, r_w >$ is defined as a sequential reference when it satisfies the following condition: $fd_t(b_i) = \infty$ for all $1 \leq i \leq w$.

In a looping reference, all blocks are accessed repeatedly with a regular reference interval. Since all blocks have the same reference interval, the detection rule for the looping reference pattern can be given as follows.

Detection Rule 2. A reference $< r_1, r_2, \cdots, r_w >$ is defined as a looping reference when it satisfies the following condition: if $bd_t(b_i) \leq bd_t(b_j)$ then $fd_t(b_i) \geq fd_t(b_j)$ for all $1 \leq i, j \leq w$.

In a temporally-clustered reference, all blocks are accessed in accordance with the condition $a_1 \geq a_2 \geq \cdots \geq a_n$, where a_i is the reference probability associated with position i of the LRU stack. That is, a more recently referenced block has higher probability of being re-referenced. Thus, the detection rule for the temporally-clustered reference is given as follows.

Detection Rule 3. A reference $< r_1, r_2, \cdots, r_w >$ is defined as a temporally-clustered reference when it satisfies the following condition: if $bd_t(b_i) \leq bd_t(b_j)$ then $fd_t(b_i) \leq fd_t(b_j)$ for all $1 \leq i, j \leq w$.

In a probabilistic reference, each block b_i is accessed with its stationary and independent probability p_i. Under the stationary and independent condition, the expected forward distance of b_i is proportional to $1/p_i$. That is, the higher the probability of reference to a block, the sooner the block will be referenced in the future. Thus, the frequency of a block, $fr_t(b_i)$, has a positive correlation with probability p_i. From this observation, the detection rule for the probabilistic reference is given as follows.

Detection Rule 4. A reference $< r_1, r_2, \cdots, r_w >$ is defined as a probabilistic reference when it satisfies the following condition: if $fr_t(b_i) \leq fr_t(b_j)$ then $fd_t(b_i) \geq fd_t(b_j)$ for all $1 \leq i, j \leq w$.

3.2 On-line Detection of Block Reference Characteristics

The above detection rules require information about the future in the form of the forward distance of blocks. To make on-line detection of block reference patterns possible without the information about the future, we take an approximate approach by using a periodic detection method that detects block reference patterns via one-level look-behind.

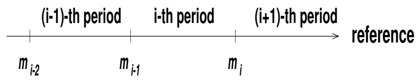


Figure 4: Periodic on-line detection of block reference patterns.

In this method, the time line is divided into periods, as shown in Figure 4. When the i-th period is over (we denote this time as m_i), we determine the forward distance $fd_{m_{i-1}}(b_j)$ for every block b_j that was accessed during the i-th period, from the standpoint of time m_{i-1}. Also, we gather the backward distance $bd_{m_{i-1}}(b_j)$ and the frequency $fr_{m_{i-1}}(b_j)$ for the blocks, again from the standpoint of time m_{i-1}. Then, by applying the detection rules to these values, we can detect the block reference patterns. This periodic detection is one-level look-behind since the detection at m_i is made based on the inter-relationship between block attribute values and the forward distance values, both seen from the standpoint of time m_{i-1}.

The periodic detection method is effective, since a large class of applications have a regular block reference pattern throughout their execution time. Furthermore, the method enables us to dynamically recognize changes in the reference patterns when an application changes its reference behavior during execution.

3.3 Block Replacement Policy

The benefit of reference characterizations can be reaped only by employing the appropriate replacement policy for each pattern. For sequential and looping reference patterns, we apply the Most Recently Used (MRU) replacement policy, where the block with the smallest backward distance is always selected for replacement. This is because the MRU is known to be the optimal policy for such reference patterns [9]. For the temporally-clustered reference pattern, we apply the LRU replacement policy, which replaces the block with the largest backward distance. Finally, for the probabilistic reference pattern, we apply the LFU replacement policy, which replaces the block with the lowest frequency. These choices are, again, due to their optimality under such reference patterns [7]. When a pattern cannot be detected we categorize it as undetected, and in this case, we apply the default LRU replacement policy.

4. APPLICATION/FILE-LEVEL CHARACTERIZATION AND ALLOCATION

Up to this point, we have discussed the mechanism for detecting the reference pattern. In the first part of this section, we pose the question of at what level of reference such a mechanism should be applied. The second part of this section discusses the buffer allocation scheme used at the application/file level.

4.1 Application/File-level Characterization

The question is this: At what level do we go about detecting the reference characteristics? In essence, we detect reference characteristics at two levels. First, at the application level and then, if necessary, at the file level. This is in contrast to traditional systems where reference characteristics are observed at a much coarser level, that is, at the file system level.

To elaborate, in real systems, multiple applications execute concurrently, making block reference requests independently. Thus, references observed at the system level are actually a mixed form of the unique reference patterns of individual applications. The scheme that we propose unravels this by managing each application separately and detecting the reference patterns on an application-by-application basis.

Even from an application's point of view, the references may be a mixture of references to multiple files that exhibit widely different reference characteristics. This may hinder the identification of the unique reference characteristics of the application. If this is the case, then we try to characterize the reference at an even finer granularity, that is, at the file level.

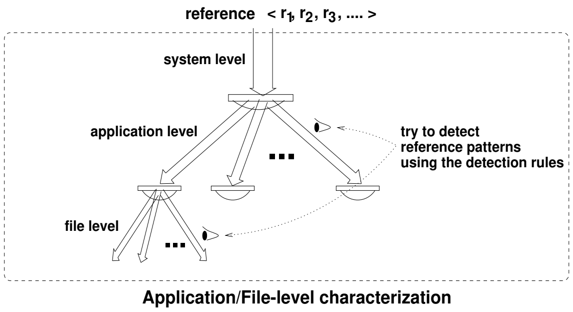


Figure 5: Application/File-level characterization.

Figure 5 depicts the Application/File-level Characterization (AFC) scheme. Traditionally, reference characteristics have been observed at the system level. In the AFC scheme, we first observe at the application level, and then, if necessary, observe at the file level. File level observation is triggered when a reference pattern cannot be detected at the application level or if the detected reference pattern changes from one to another.

Figures 1, 2, and 3 that we presented in Section 1, put together, are an example of the detection process and its result. Figure 1 is the system workload that is seen at the system level, whose characteristics are not clearly discernible as three independent applications are being multiprogrammed. The three separated out and independently characterized are shown in the three figures of Figure 2. The *cscope* application is detected as a looping reference pattern since a single file is accessed repeatedly. (We return to detailed discussions of individual applications later in Section 5.) The *cpp* application is detected as a probabilistic reference pattern all throughout its execution. A more noteworthy application is the *postgres2* application as shown in Figure 2(c). It turns out that reference patterns cannot be determined at various stages of the application. Hence, the detection mechanism tries a finer-grained file level detection, and the results are obvious as shown in Figure 3. Of the four files that *postgres2* accesses file 1 is detected as being referenced sequentially, while files 2, 3, and 4 are detected as being referenced in a looping pattern with loop length being roughly 960, 190, and 20, respectively.

4.2 Buffer Allocation Scheme

One problem that we face with fine-grained management of resources such as at the application or file level is that of allocation. How much of the available blocks should each of the applications[1] be allocated? The solution to this problem must be fair as well as effective.

In this section, we propose the ΔHIT allocation scheme where the allocation is based on optimizing the performance of the overall system. In essence, the scheme takes away a block from the application where the removal results in the smallest reduction in the number of expected buffer cache hits.

Let us now elaborate. When free blocks are available in the system, the blocks are allocated to applications as requested. When a block reference misses in the buffer cache and the system does not have any free buffer blocks left, a block has to be taken away from one of the applications. The victim application is chosen as follows. The system computes $\Delta HIT(B)$ for each application, where $\Delta HIT(B) = HIT(B) - HIT(B-1)$, where B is the number of blocks allocated to the application. Then, the application with the smallest $\alpha \times \Delta HIT(B)$ value is chosen as the victim, where α is the application's activity rate, defined as the number of block accesses to the pattern per unit time. What the ΔHIT scheme does is that it satisfies the block miss by taking away a block from the application that will be least hurt, that is, performance least affected, by the reduction.

Now the problem is how to compute $\Delta HIT(B)$. This is done by making use of the fact that an application is categorized into one of the four reference patterns (in fact, five if we consider the undetected pattern). For each of these patterns, the expected buffer cache hit ratios can be calculated as follows:

Sequential Reference Pattern The expected buffer hit ratio of the sequential reference pattern is $HIT_{seq}(B) = 0$, where B is the number of buffer blocks allocated to the pattern, since a block referenced is never again referenced.

Looping Reference Pattern For the looping reference pattern, we apply the MRU policy. Hence, with w the length of the reference and l the length of the loop, the expected buffer hit ratio is $HIT_{loop}(B) = (min[l,B] * (w/l))/w = min[l,B]/l$, assuming that w is a multiple of l.

Temporally-clustered Reference Pattern As we apply the LRU policy the expected buffer hit ratio of this pattern is $HIT_{tc}(B) = \sum_{i=1}^{B} a_i$, where a_i is the reference probability of position i in the LRU stack. It is possible to measure a_i by observing where buffer hits occur in a ghost LRU stack buffer [15, 18], or by

[1] For simplicity, in this section, we shall use the term "application" when we actually mean "application and/or file".

using an analytical buffer model [4]. The ghost LRU stack buffer is simply a list of dataless block headers where all blocks are ordered by their backward distance. Specifically, a_i is measured by the hit counts at stack position i divided by the reference length w. To reduce the measurement overhead, the hit counts are observed not by individual stack positions but by segments (disjoint intervals of stack positions), done similarly in [15].

Probabilistic Reference Pattern As we apply the LFU policy for the probabilistic reference pattern, the expected buffer hit ratio is $HIT_{prob}(B) = \sum_{i=1}^{B} p_i$, where p_i is the reference probability of block b_i (assuming $p_i \geq p_j$, if $i \leq j$). Similarly to the temporally-clustered reference pattern, p_i can be measured by observing where buffer hits occur in the ghost LFU buffer or through an analytical buffer model [4]. In this case, blocks in the ghost LFU buffer are ordered by their frequency.

The final consideration is for references that are categorized as undetected. Since we apply the LRU replacement policy to this reference, we simply employ the same method used for the temporally-clustered pattern.

Note that as with the AFC scheme, the ΔHIT buffer allocation scheme determines the buffer allocation on-line.

5. PERFORMANCE EVALUATION

We evaluate the performance of the proposed scheme using trace-driven simulation with traces from seven real applications. We first describe the experimental setup along with the characteristics of the applications used in the experiments. Then, we give the detection results of the applications. Finally, we present the performance results for both single application and multiple application executions.

5.1 Details of the Simulation

To evaluate the performance of the proposed scheme, we used traces from the following seven applications: *cscope*, *glimpse*, *cpp*, *link*, *sort*, *mpeg_play*, and *postgres*. The characteristics of these applications are described below.

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 for C symbols or finding specific functions or identifiers. We traced cscope by running cscope on kernel sources of roughly 9MB in size and executing queries that search for five different 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.*

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

link Link is the UNIX link-editor. We traced this application while building the FreeBSD kernel from about 2.5MB of object files.

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.

mpeg_play Mpeg_play is an MPEG player written in C and obtained from the University of California at Berkeley. A 45MB MPEG video file was used as the input data.

postgres1 and postgres2 Postgres is a relational database system from the University of California at Berkeley. The version 6.2 release of the PostgresSQL (formally known as Postgres95) and relations from a scaled-up Wisconsin benchmark such as thoustup and tenthoustup were used. We used two join queries: postgres1 is a join between the hundredthoustup and twohundredthoustup relations, while postgres2 is a join among four relations, namely, twothoustup, twentythoustup, hundredthoustup, and twohundredthoustup. The sizes of twothoustup, twentythoustup, hundredthoustup, and twohundredthoustup are approximately 150KB, 1.5MB, 7.5MB, and 15MB, respectively.

We built a simulator for the proposed scheme, which consists of the reference pattern detection module, the buffer cache module, and the buffer block allocation module. The reference pattern detection module employs the AFC scheme to detect, on-line, the reference patterns. Detection is done periodically, with the detection period set to 500. (Experimental results show that the detection period length has insignificant influence on the performance when it is between 250 and 1000.) The detection results are conveyed to the buffer cache module that manages the buffer cache for each application/file employing appropriate replacement policies. The buffer block allocation module makes on-line decisions using the ΔHIT allocation scheme when a miss occurs in the buffer and no blocks are available thereby adjusting the distribution of blocks allotted to applications/files.

5.2 Detection Results

Table 1 shows a summary of the detected block reference patterns observed at the application level. The results are shown at each detection point when the detection period is set to 500. For the *cpp* and *mpeg_play* applications, we detect a consistent reference pattern throughout the detection periods. Hence, detection at finer granularity is not considered. For all other applications, either when we find an undetected period or when the detected reference pattern changes, detection at the file level is triggered.

Results of finer-grain detection for the *postgres2* application were already presented in Figure 3 (and described in Section 4). *Postgres2* used a mixture of files that were individually accessed sequentially or in a looping fashion. Similar results were found for the *sort* application. The reference patterns detected for *sort* at the application and file levels are shown in Figure 6. This peculiar looking reference pattern comes from the fact that the UNIX sort utility uses merge sort, first partitioning the input file into chunks that are written into temporary files. These files are sorted, and then merged together resulting in the output file. In the figure, files 1 and 11, which are referenced sequentially, are the input and output files, respectively, while the other nine files, accessed in loops, are the temporary files. It was the combined reference patterns of these numerous files that were

Table 1: **Detection results (S: sequential, L: looping, T: temporally-clustered, P: probabilistic, U: undetected).**

Application	Number of references	Detected reference patterns at the i-th detection period (detection period is 500)																			
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
cscope	6,782	S	S	L	L	L	L	L	L	L	L	L	L	L							
glimpse	6,047	S	S	L	U	L	L	U	U	U	L	U	L								
cpp	9,051	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P		
link	4,194	S	S	S	L	L	T	U	U												
sort	3,438	S	S	S	U	L	L														
mpeg_play	5,544	S	S	S	S	S	S	S	S	S	S	S									
postgres1	10,473	S	S	S	S	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L
postgres2	10,449	T	U	P	P	P	U	S	U	P	U	U	L	U	S	U	P	U	U	P	L

observed at the application level, making the reference pattern difficult to decipher.

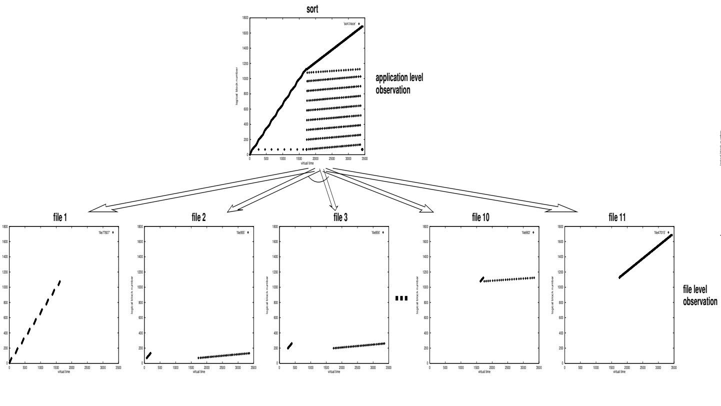


Figure 6: Finer-grain detection results for sort.

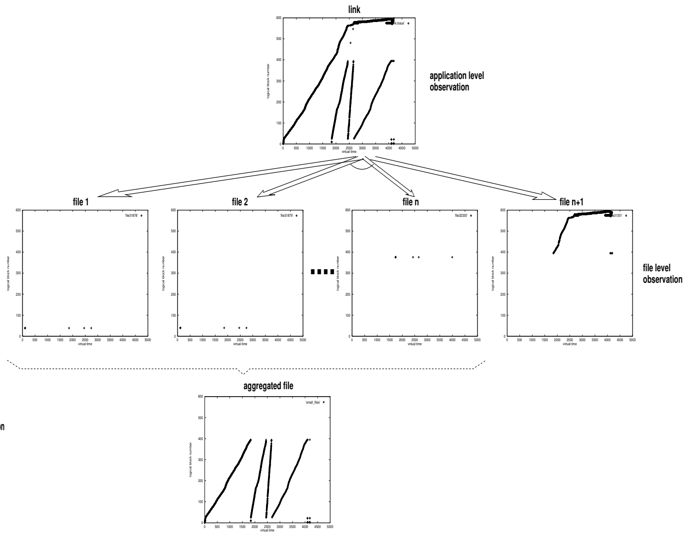


Figure 7: Finer-grain detection results for link.

Let us now consider the *link* application. Figure 7 shows finer-grain detection results of *link*, the top-most one being that observed at the application level and that at the second level observed at the file level. Recall that *link* is a UNIX link-editor utility, which makes an executable file from object files. Files 1 to n, in Figure 7 correspond to the object files, and file $n+1$ corresponds to the output file, that is, the executable file. Applications such as *link* touches numerous small files (in this case, n is 297). If we manage every single small file separately for detection this may incur considerable overhead. Hence, to reduce the handling overhead of small file accesses, we aggregate small files whose size is smaller than s and treat them as a single file. The result of this aggregation is shown at the bottom of Figure 7. In our experiments, we set s to 32KB, which is just 4 buffer blocks in typical UNIX systems. As a result, detection and management of file references for *link* happen on only two files; the file $n+1$ and the aggregated file. The two files are detected as being referenced sequentially and in a looping pattern, respectively.

Aggregation is used for the *glimpse* application as well. A total of 779 files are touched by *glimpse* of which 771 files are less than s, hence aggregated to one file. This file is detected as being sequentially referenced. The remaining eight files are detected as being referenced either sequentially or in a looping pattern.

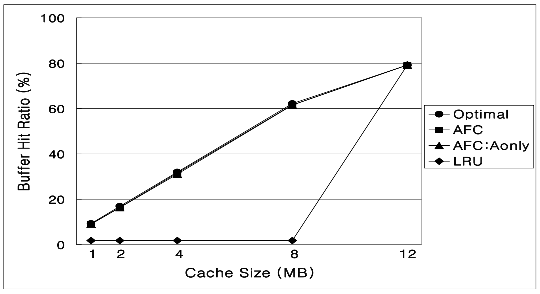
5.3 Effect of Application/File-based Characterization

A comparison of the performance of single application executions using the Application/File-level Characterization (AFC) scheme and that which simply uses the LRU policy is made. We measured the buffer hit ratio of each application for a variety of buffer cache sizes ranging from 0.25MB to 12MB with the block size set to 8KB.

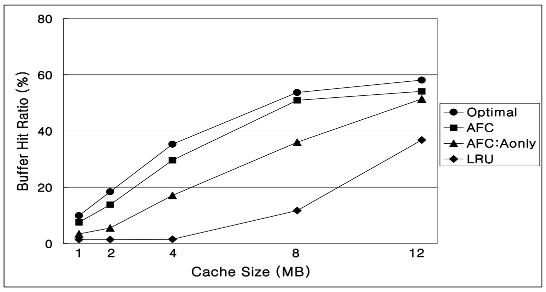
The results are shown in Figure 8. Note that the scales vary for each of the figures. We show two results for AFC. One, denoted as **AFC:Aonly**, detects reference patterns at the application level only. The other, denoted as **AFC**, uses both application and file levels of detection. Results of the off-line optimal replacement policy are also shown for comparison purposes.

The findings of these experiments can be summarized as follows.

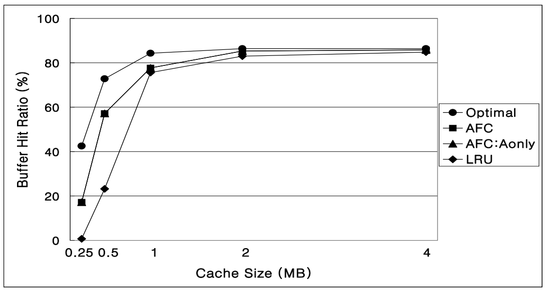
- Overall, AFC does far better than the LRU, performing very close to the optimal policy for *glimpse*, *postgres1*, and *postgres2* and matching the performance of the optimal policy for *cscope* and *mpeg_play*.
- Compared with the LRU scheme, AFC increases the buffer hit ratio by up to 59 percentage points (for the *cscope* application with 8MB buffer cache size) with



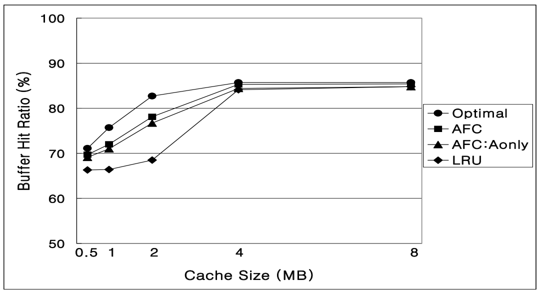
(a) cscope



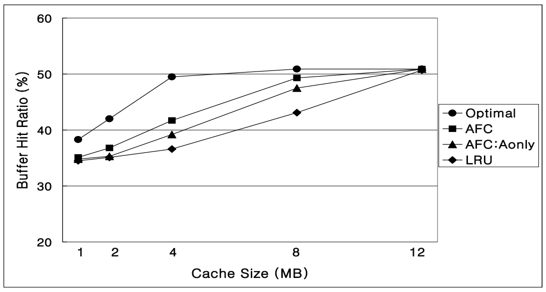
(b) glimpse



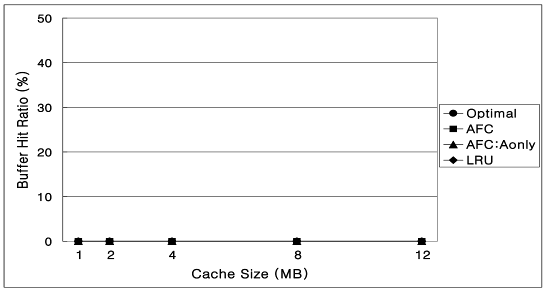
(c) cpp



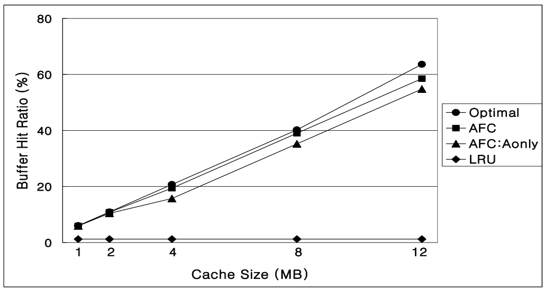
(d) link



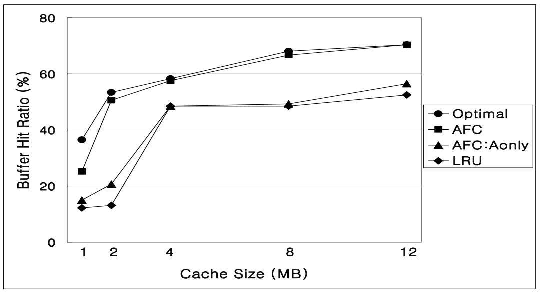
(e) sort



(f) mpeg_play

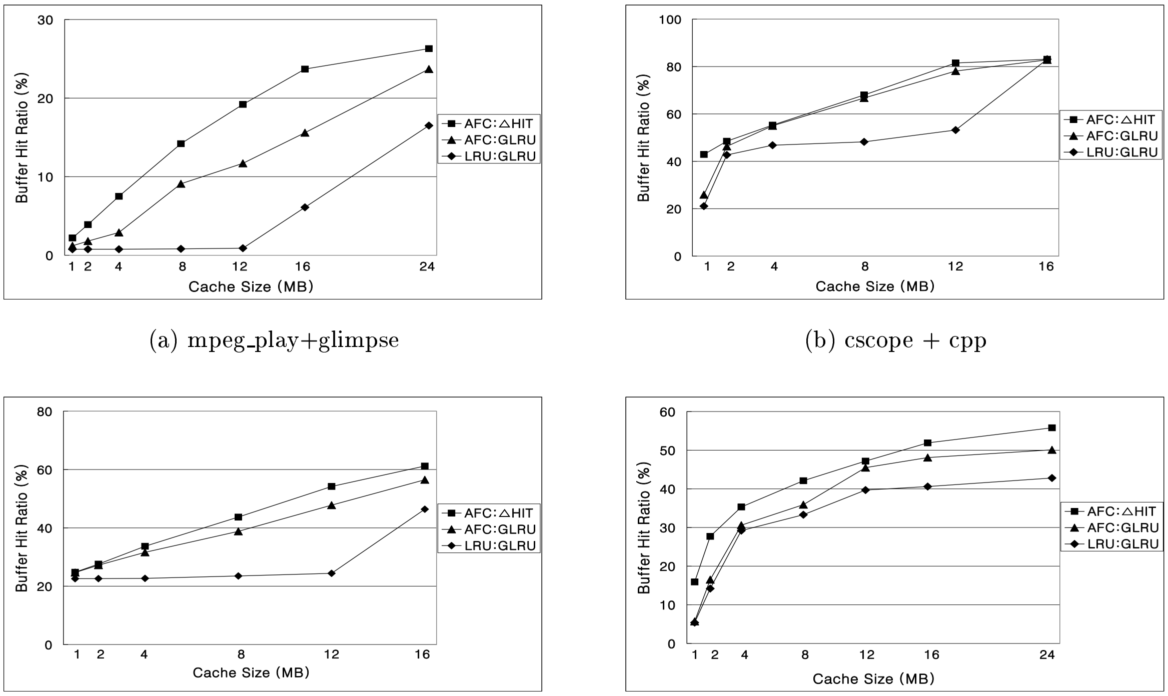


(g) postgres 1



(h) postgres 2

Figure 8: Single application performance.



(a) mpeg_play+glimpse

(b) cscope + cpp

(c) link+sort+postgres1

(d) cscope+cpp+mpeg_play+postgres2

Figure 9: Multiple application performance.

an average of 13 percentage points.

- For *cpp* and *mpeg_play*, the performance of **AFC** and that of **AFC:Aonly** are the same as detection does not progress to the file level. The *cscope* application triggers file level detection as there is a change in the detected reference pattern. However, little gain is obtained through this action since *cscope* mainly references a single file, namely, *cscope.out*.
- The results show that file level detection can make significant improvements in the performance. Specifically, *glimpse* and *postgres2* reap considerable benefits from this approach while *link*, *sort*, and *postgres1* improve slightly.

5.4 Effect of Fine-Grained Buffer Allocation

Figure 9 shows the performance of the AFC scheme compared to the LRU policy when multiple applications are executed simultaneously. The traces used in these experiments were obtained by actually executing the applications.

Improvements in performance can come from two sources. One is from the AFC scheme where fine-grained characterization allows the optimal replacement policy to be employed. The second source comes from the ΔHIT buffer allocation. Results shown in Figure 9 distinguish these two sources.

The line denoted as **LRU:GLRU** is the performance of the LRU policy with a buffer allocation scheme that replaces the global LRU block. This buffer allocation scheme corresponds to the LRU-SP (Swapping and Placeholders) strategy [3] when no hint is provided to the applications. The line denoted as **AFC:GLRU** is the performance of the AFC scheme with a buffer allocation scheme similar to LRU-SP. It is similar in that the global LRU block is selected and that swapping occurs, but is not LRU-SP because the placeholder mechanism is not used.

The performance difference between the **LRU:GLRU** and the **AFC:GLRU** comes from the AFC scheme only. Note that for all the experiments conducted the improvement brought by the AFC scheme is quite high.

The performance improvement brought in by the ΔHIT allocation scheme is shown in the performance difference between **AFC:GLRU** and **AFC:ΔHIT**. **AFC:ΔHIT** is the performance obtained when using both the AFC and ΔHIT schemes. Note that there is another notable jump in improvement when using the ΔHIT allocation scheme.

Overall, the maximum increase in the hit ratio of **AFC:ΔHIT** is 32 percentage points (for the *link+sort+postgres1* case with 12MB buffer cache size) compared to **LRU:GLRU**, with an average increase of 12 percentage points.

6. CONCLUSIONS

Two contributions are made in this paper. First, we showed that system level characterization of references is inadequate for optimal buffer cache performance and that a finer-grained characterization approach is needed. Though application level characterizations, either provided by the user or by a system module, have been proposed, this is the first attempt, to the best of our knowledge, to consider file level characterizations. We proposed the Application/File-level

Characterization (AFC) scheme where we detect on-line the reference characteristics at the application level and then at the file level, if necessary.

The second contribution is in proposing an efficient and fair buffer allocation scheme. Application or file level resource management is infeasible unless there exists an allocation scheme that is efficient and fair. We proposed the ΔHIT allocation scheme that takes away a block from the application/file that will be least hurt, that is, performance least affected, by the reduction.

Both the AFC and ΔHIT schemes are on-line schemes that detect and allocate while applications are executing. Experiments using trace-driven simulations show that substantial performance improvements can be made. For single application executions the hit ratio increased an average of 13 percentage points compared to the LRU policy, with a maximum increase of 59 percentage points, while for multiple application executions, the hit ratio increase was slightly less.

An important feature of the proposed schemes is their adaptability. As the spectrum of applications that make use of computer systems get broader and wider, we will see a large variance in their reference behavior. For example, in multimedia file systems, applications involve storing and retrieving multiple data types such as text, audio, and video, and applications for audio and video data have different block reference patterns compared with applications for textual data. This large variance implies that no single buffer or resource management scheme will be suitable for all applications and that adaptability is indispensable for efficient resource management. The results presented here are only the first couple of steps in this research direction. We are attempting to take this approach to other components of the computer system as well.

7. REFERENCES

[1] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization : The Search for Invariants. In *Proceedings of the 1996 ACM SIGMETRICS Conference*, pages 126–137, 1996.

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

[3] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application Controlled File Caching. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 165–178, 1994.

[4] J. Choi, S. Cho, S. H. Noh, S. L. Min, and Y. Cho. Analytical Prediction of Buffer Hit Ratios. *IEE Electronics Letters*, 36(1):10–11, January 2000.

[5] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme. In *Proceedings of the 1999 Annual USENIX Conference*, pages 239–252, 1999.

[6] G. Class and P. Cao. Adaptive Page Replacement Based on Memory Reference Behavior. In *Proceedings of the 1997 ACM SIGMETRICS Conference*, pages 115–126, 1997.

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

[8] A. Dan, P. S. Yu, and J.-Y. Chung. Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability. *VLDB Journal*, 4(1):127–154, January 1995.

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

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

[11] 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. In *Proceedings of the 1999 ACM SIGMETRICS Conference*, pages 134–143, 1999.

[12] 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*, pages 3–17, 1996.

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

[14] 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*, pages 388–397, 1993.

[15] 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*, pages 1–16, 1995.

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

[17] 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*, pages 124–138, 1998.

[18] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *Proceedings of the 1999 ACM SIGMETRICS Conference*, pages 122–133, 1999.