A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References[*]

Jong Min Kim[†] Jongmoo Choi[†] Jesung Kim[†] Sam H. Noh[‡]
Sang Lyul Min[†] Yookun Cho[†] Chong Sang Kim[†]

[†]*School of Computer Science and Engineering*
Seoul National University
Seoul 151-742, Korea

[‡]*Department of Computer Engineering*
Hong-Ik University
Seoul 121-791, Korea

Abstract

In traditional file system implementations, the Least Recently Used (LRU) block replacement scheme is widely used to manage the buffer cache due to its simplicity and adaptability. However, the LRU scheme exhibits performance degradations because it does not make use of reference regularities such as sequential and looping references. In this paper, we present a Unified Buffer Management (UBM) scheme that exploits these regularities and yet, is simple to deploy. The UBM scheme automatically detects sequential and looping references and stores the detected blocks in separate partitions of the buffer cache. These partitions are managed by appropriate replacement schemes based on their detected patterns. The allocation problem among the divided partitions is also tackled with the use of the notion of marginal gains. In both trace-driven simulation experiments and experimental studies using an actual implementation in the FreeBSD operating system, the performance gains obtained through the use of this scheme are substantial. The results show that the hit ratios improve by as much as 57.7% (with an average of 29.2%) and the elapsed times are reduced by as much as 67.2% (with an average of 28.7%) compared to the LRU scheme for the workloads we used.

1 Introduction

Efficient management of the buffer cache by using an effective block replacement scheme is important for improving file system performance when the size of the buffer cache is limited. To this end, various block replacement schemes have been studied [1, 2, 3, 4, 5, 6]. Yet, the Least Recently Used (LRU) block replacement scheme is still widely used due to its simplicity. While simple, it adapts very well to the changes of the workload, and has been shown to be effective when recently referenced blocks are likely to be re-referenced in the near future [7]. A main drawback of the LRU scheme, however, is that it cannot exploit regularities in block accesses such as sequential and looping references and thus, yields degraded performance [3, 8, 9]. In this paper, we present a Unified Buffer Management (UBM) scheme that exploits these regularities and yet, is simple to deploy. The performance gains are shown to be substantial. Trace-driven simulation experiments show that the hit ratios improve by as much as 57.7% (with an average of 29.2%) compared to the LRU scheme for the traces we considered. Experimental studies using an actual implementation of this scheme in the FreeBSD operating system show that the elapsed time is reduced by as much as 67.2% (with an average of 28.7%) compared to the LRU scheme for the applications we used.

1.1 Motivation

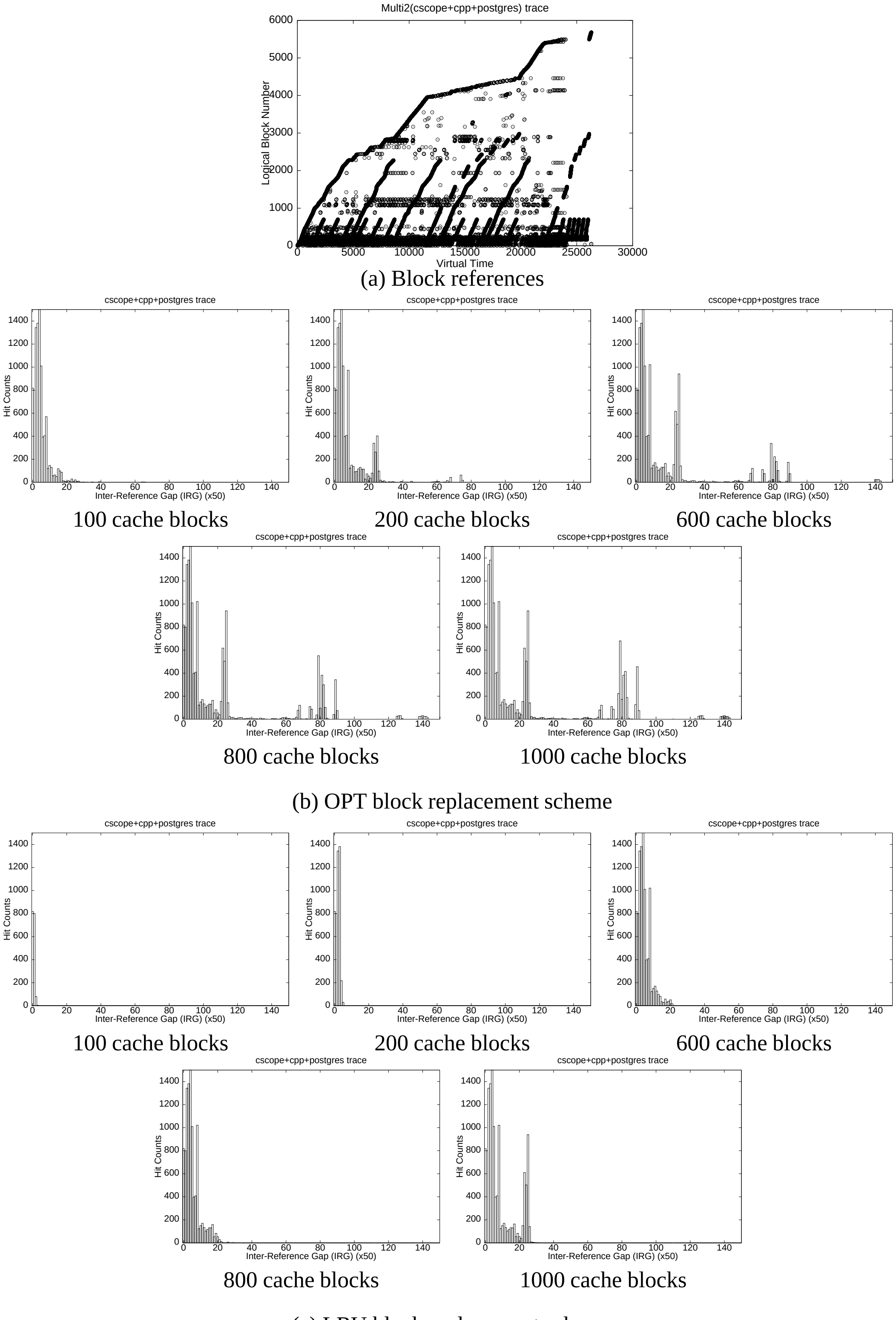
The graphs in Figure 1 show the motivation behind this study. First, Figure 1(a) shows the space-time graph of block references from three applications, namely, *cscope*, *cpp*, and *postgres* (details of which will be discussed in Section 4), executing concurrently. The

[*] This work was supported in part by the Ministry of Education under the BK21 program and by the Ministry of Science and Technology under the National Research Laboratory program.

[†] http://archi.snu.ac.kr/{jmkim,jskim,symin,cskim}

[†] http://ssrnet.snu.ac.kr/˜{choijm,cho}

[‡] http://www.cs.hongik.ac.kr/˜noh



(a) Block references

100 cache blocks

200 cache blocks

600 cache blocks

800 cache blocks

1000 cache blocks

(b) OPT block replacement scheme

100 cache blocks

200 cache blocks

600 cache blocks

800 cache blocks

1000 cache blocks

(c) LRU block replacement scheme

Figure 1: Caching behaviors of the OPT block replacement scheme and the LRU block replacement scheme.

x-axis is the virtual time which ticks at each block reference and the y-axis is the logical block number of the block referenced at the given time. From this graph, we can easily notice sequential and looping reference regularities throughout their execution.

Now consider the graphs in Figures 1(b) and 1(c). They show the Inter-Reference Gap (IRG) distributions of blocks that hit in the buffer cache for the off-line optimal (OPT) block replacement scheme and the LRU block replacement scheme, respectively, as the cache size increases from 100 blocks to 1000 blocks. The x-axis is the IRG and the y-axis is the total hit count with the given IRG.

Observe from the corresponding graphs of the two figures the difference with which the two replacement schemes behave. The main difference comes from how looping references (that is, blocks that are accessed repeatedly with a regular reference interval, which we refer to as the loop period) are treated. The OPT scheme retains the blocks in the increasing order of loop periods as the cache size increases since the scheme chooses a victim block according to the forward distance (i.e., difference between the time of the next reference in the future and the current time). From Figure 1(b), we can see that in the OPT scheme the hit counts of blocks with IRG between 70 and 90 increase gradually as the cache size increases. On the other hand, in the LRU scheme there are no buffer hits at this range of IRGs even when the buffer cache has 1000 blocks. This results from blocks at these IRGs being replaced either by blocks that are sequentially referenced (and thus never re-referenced) or by those with larger IRGs (and thus are replaced before being re-referenced). Although predictable, the regularities of sequential and looping references are not exploited by the LRU scheme, which leads to significantly degraded performance.

From this observation, we devise a new buffer management scheme called the Unified Buffer Management (UBM) scheme. The UBM scheme exploits regularities in reference patterns such as sequential and looping references. Evaluation of the UBM scheme using both trace-driven simulations and an actual implementation in the FreeBSD operating system shows that 1) the UBM scheme is very effective in detecting sequential and looping references, 2) the UBM scheme manages sequentially-referenced and looping-referenced blocks similarly to the OPT scheme, and 3) the UBM scheme shows substantial performance improvements.

1.2 The Remainder of the Paper

The remainder of this paper is organized as follows. In the next section, we review related work. In Section 3, we explain the UBM scheme in detail. In Section 4, we describe our experimental environments and compare the performance of the UBM scheme with those of previous schemes through trace-driven simulations. In Section 5, an implementation of the UBM scheme in the FreeBSD operating system is evaluated. Finally, we provide conclusions and directions for future research in Section 6.

2 Related Work

In this section, we place previous page/block replacement schemes into the following three groups and in turn, survey the schemes in each group.

- Replacement schemes based on frequency and/or recency factors.
- Replacement schemes based on user-level hints.
- Replacement schemes making use of regularities of references such as sequential references and looping references.

The FBR (Frequency-based Replacement) scheme by Robinson and Devarakonda [1], the LRU-K scheme by O'Neil *et al.* [2], the IRG (Inter-Reference Gap) scheme by Phalke and Gopinath [5], and the LRFU (Least Recently/Frequently Used) scheme by Lee *et al.* [6] fall into the first group. The FBR scheme chooses a victim block to be replaced based on the frequency factor differing from the Least Frequently Used (LFU) mainly in that it considers correlations among references. The LRU-K scheme bases its replacement decision on the blocks' kth-to-last reference, while the IRG scheme's decision is based on the inter-reference gap factor. The LRFU scheme considers both the recency and frequency factors of blocks. These schemes, however, show limited performance improvements because they do not consider regular references such as sequential and looping references.

Application-controlled file caching by Cao *et al.* [4] and informed prefetching and caching by Patterson *et al.* [10] are schemes based on user-level hints. These schemes choose a victim block to be replaced based on user-provided hints on application reference characteristics, 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 the applicability.

The third group of schemes considers regularities of references, and the 2Q scheme by Johnson and Shasha [3], the SEQ scheme by Glass and Cao [8], and the EELRU (Early Eviction LRU) scheme by Smaragdakis *et al.* [9] fall into this group. The 2Q scheme quickly removes from the buffer cache sequentially-referenced blocks and looping-referenced blocks with long loop periods. This is done by using a special buffer called the $A1in$ queue in which all missed blocks are initially placed and from which the blocks are replaced in the FIFO order after short residence. On the other hand, the scheme holds looping-referenced blocks with short loop periods in the main buffer cache by using a ghost buffer called the $A1out$ queue in which the addresses of blocks replaced from the $A1in$ queue are temporarily placed to discriminate between frequently referenced blocks and infrequently referenced ones. When a block is re-referenced while its address is in the $A1out$ queue, it is promoted to the main buffer cache. The 2Q scheme, however, has two drawbacks. One is that an additional miss has to occur for a block to be promoted to the main buffer cache from the $A1out$ queue. The other is that a careful tuning is required for two control parameters, that is, the size of the $A1in$ queue and the size of the $A1out$ queue, which may be sensitive to the types of workload.

The SEQ scheme detects long sequences of page faults and applies the Most Recently Used (MRU) scheme to those pages. However, in determining the victim page, it does not distinguish sequential and looping references. The EELRU scheme confirms the existence of looping references by examining aggregate recency distributions of referenced pages and changes the page eviction points using a simple on-line cost/benefit analysis. The EELRU scheme, however, does not distinguish between looping references with different loop periods.

3 The Unified Buffer Management Scheme

The Unified Buffer Management (UBM) scheme is composed of the following three main modules.

Detection This module automatically detects sequential and looping references. After the detection, block references are classified into sequential, looping, or other references.

Replacement This module applies different replacement schemes to the blocks belonging to the three reference patterns according to the properties of each pattern.

Allocation This module allocates the limited buffer cache space among the three partitions corresponding to sequential, looping, and other references.

In the following subsections, we give a detailed explanation of each of these modules.

3.1 Detection of Sequential and Looping References

The UBM scheme automatically detects sequential, looping, and other references according to the following rules:

Sequential references that are consecutive block references occurring only once.

Looping references that are sequential references occurring repeatedly with a regular interval.

Other references that are detected neither as sequential nor as looping references.

Figure 2 shows the classification process of the UBM scheme. Note that looping references are initially detected as sequential until they are re-referenced.

For on-line detection of sequential and looping references, information about references to blocks in each file is maintained in an abstract form. The elements needed are shown in Figure 3.

Information for each file is maintained as a 4-tuple consisting of a file descriptor ($fileID$), a start block number ($start$), an end block number (end), and a loop period ($period$). A reference is categorized as a sequential reference after a given number of consecutive references are made. For a sequential reference the loop period is ∞, while for a looping reference its value is the actual loop period. In real systems, the loop period fluctuates by various factors including the degree

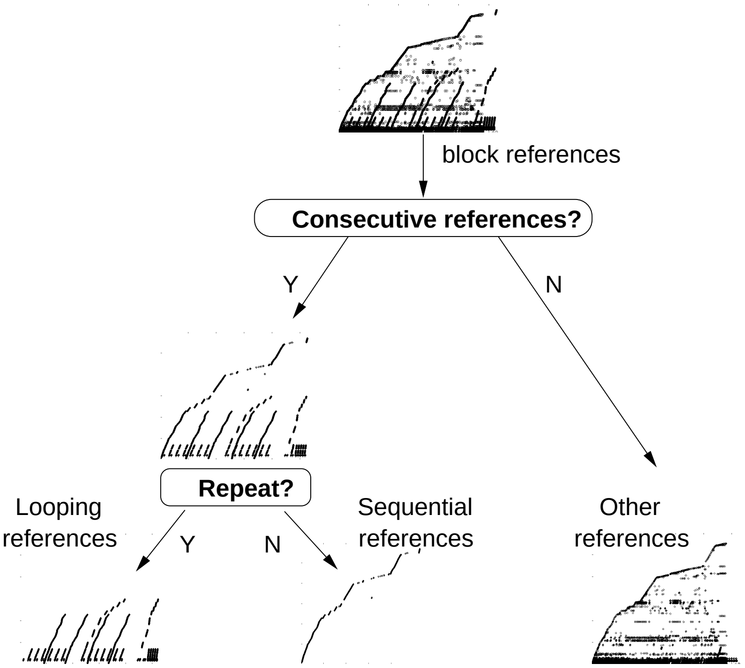


Figure 2: Classification process of the UBM scheme.

of multiprogramming and scheduling. Hence, this value is set as an exponential average of measured loop periods. Also, since a looping reference is initially detected as a sequential reference, its blocks are managed just like those belonging to a sequential reference until they are re-referenced. This may make them miss the first time they are re-referenced if there is not enough space in the cache (cf. Section 4.7).

The resulting table keeps information for sequences of consecutive block references that are detected up to the current time and is updated whenever a block reference occurs. In most UNIX file systems, sequences of consecutive block references are detected by using vnode numbers and consecutive block addresses.

Figure 4 shows an example of sequential and looping references, and the data structure that is used to maintain information for these references. In the figure, the file with fileID 3 is a sequential reference as it has ∞ as its loop period. Files with fileID 1 and 2 are looping references with loop periods of 80 and 40, respectively.

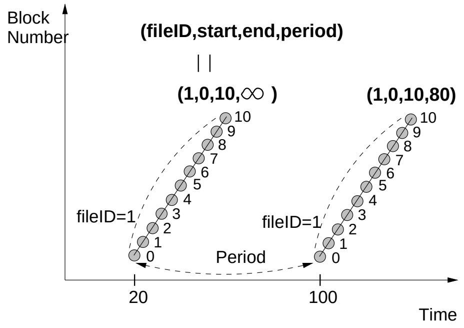
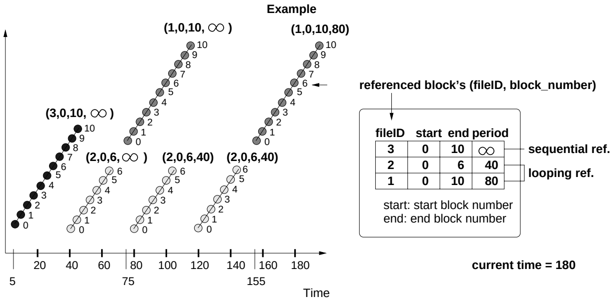


Figure 3: Elements for representing sequential and looping references.



fileID	start	end	period
3	0	10	∞
2	0	6	40
1	0	10	80

start: start block number
end: end block number

Figure 4: Example of detection: sequential and looping references.

3.2 Block Replacement Schemes

The detection mechanism results in files being categorized into three types, that is, sequential, looping, and other references. The buffer cache is divided into three

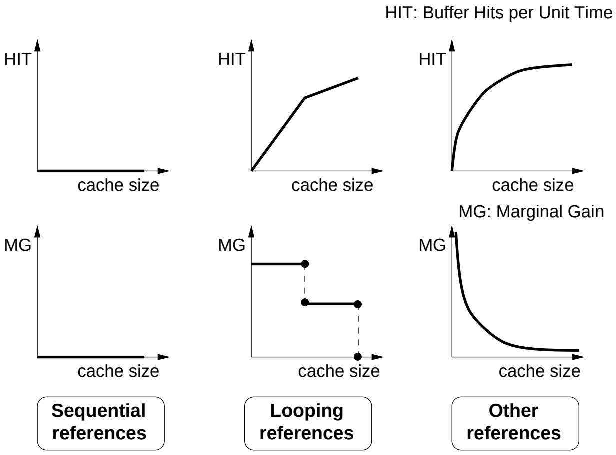


Figure 5: Typical curves of buffer hits per unit time and marginal gain values.

partitions to accommodate the three different types of references. Management of the partitions must be done according to the reference characteristics of blocks belonging to each partition.

For the partition that holds sequential references, it is a simple matter. Sequentially-referenced blocks are never re-referenced. Hence, the referenced blocks need not be retained and therefore, the MRU replacement policy is used.

For the partition that holds looping references, victim blocks to be replaced are chosen based on their loop periods because their re-reference probabilities depend on these periods. To do so, we use a period-based replacement scheme that replaces blocks in decreasing order of loop periods, and the MRU block replacement scheme among blocks with the same loop period.

Finally, within the partition that holds other references, victim blocks to be replaced can be chosen based on their recency, frequency, or a combination of the two factors. Hence, we may use any of previously proposed replacement schemes including the Least Recently Used (LRU), the Least Frequently Used (LFU), the LRU-K, and the Least Recently/Frequently Used (LRFU) as long as they have a model that approximates the hit ratio for a given buffer size to compute the marginal gain, which will be explained in the next subsection. In this paper, we assume the LRU replacement scheme.

3.3 Buffer Allocation Based on Marginal Gain

The buffer cache has now been divided into three partitions that are being managed separately. An important problem that should be addressed then is how to allocate the blocks in the cache among the three partitions. To this end, we use the notion of marginal gains, which has frequently been used in resource allocation strategies in various computer systems areas [10, 11, 12, 13].

Marginal gain is defined as $MG(n) \approx Hit(n) - Hit(n-1)$, which specifies the expected number of extra buffer hits per unit time that would be obtained by increasing the number of allocated buffers from (n-1) to n, where $Hit(n)$ is the expected number of buffer hits per unit time using n buffers. In the following, we explain how to predict the marginal gains of sequential, looping, and other references, as the buffer cache is partitioned to accommodate each of these types of references.

The expected number of buffer hits per unit time of sequential references when using n buffers is $Hit_{seq}(n) = 0$, and thus, the expected marginal gain is always $MG_{seq}(n) = 0$.

For looping references, the expected number of buffer hits per unit time and the expected marginal gain value are calculated as follows.

First, for a looping reference, $loop_i$, with loop length l_i and loop period p_i, the expected number of buffer hits per unit time when using n buffers is $Hit_{loop_i}(n) = min[l_i, n]/p_i$. Thus, if $n \leq l_i$, the expected marginal gain is $MG_{loop_i}(n) = n/p_i - (n-1)/p_i = 1/p_i$ and if $n > l_i$, $MG_{loop_i}(n) = l_i/p_i - l_i/p_i = 0$.

Now, assume that there are L looping references $\{loop_1, ..., loop_i, ..., loop_L\}$, where the loops here are arranged in the increasing order of loop periods. Let i_{max} be the maximum of i such that $m = \sum_{k=1}^{i} l_k < n$, where n is the number of buffers in the partition for looping references. If $i_{max} = L$, then all loops can be held in the buffer cache, and hence $Hit_{loop}(n) = \sum_{k=1}^{L} l_k/p_k$, and $MG_{loop}(n) = 0$. Consider now, the more complicated case where $i_{max} < L$. The expected number of buffer hits per unit time of these looping references when using n buffers is $Hit_{loop}(n) = \sum_{k=1}^{i_{max}} l_k/p_k + min[l_{i_{max}+1}, n - \sum_{k=1}^{i_{max}} l_k]/p_{i_{max}+1}$. (Recall that we are using the period-based replacement scheme to manage the partition for looping references. Hence, there can be no loops within this partition that has loop period longer than $p_{i_{max}+1}$.) Hence, the expected marginal gain is $MG_{loop}(n) = 1/p_{i_{max}+1}$.

Finally, for the partition that holds the other references and which is managed by the LRU replacement scheme, the expected number of buffer hits per unit time and the expected marginal gain value can be calculated from the buffer hit ratio using ghost buffers [11, 10] and/or the Belady's lifetime function [14]. Ghost buffers, sometimes called $dataless$ buffers, are used to estimate the number of buffer hits per unit time for cache sizes larger than the current size when the cache is managed by the LRU replacement scheme. A ghost buffer does not contain any data blocks but maintains control information needed to count cache hits. The prediction of the buffer hit ratio using only ghost buffers is impractical due to the overhead of measuring the hit counts of all LRU stack positions individually. In the UBM scheme, we use an approximation method suggested by Choi *et al.* [13]. The proposed method utilizes the Belady's lifetime function, which is well-known to approximate the buffer hit ratio of references that follow the LRU model. Specifically, the hit ratio with n buffers is given by Belady's lifetime function as

$$hit_{other}(n) = h_1 + h_2 + h_3 + ... + h_n \approx 1 - c * n^{-k}$$

where c and k are control parameters. Specific c and k values can be calculated on-line by using measured buffer hit ratios at pre-set cache sizes. Ghost buffers are used to determine the hit ratios at these pre-set cache sizes. The overhead of using ghost buffers in this case is minimal as accurate LRU stack positions of referenced blocks need not be located. For example, to calculate the values of the control parameters, c and k, buffer hit ratios at a minimum of two cache sizes, say, p and q, where $p \neq q$ are required. Using these values and the equation of the lifetime function, we can calculate the values of c and k. Then, the expected number of buffer hits per unit time is given by $Hit_{other}(n) = hit_{other}(n) \times \frac{n_{other}}{n_{total}}$ where n_{other} and n_{total} are the number of other references and the number of total references, respectively, during the observed period. Finally, the expected marginal gain is simply $MG_{other}(n) = Hit_{other}(n) - Hit_{other}(n-1)$.

Figure 5 shows typical curves of both buffer hits per unit time and marginal gain values of sequential, looping, and other references as the number of allocated buffers increases. In the UBM scheme, since the marginal gain of sequential references, $MG_{seq}(n)$, is always zero, the buffer manager does not allocate more than one buffer to the corresponding partition, except when buffers are not fully utilized. That is, only when there are free buffers at the initial stage of allocation, more than one buffer may be allocated to this partition. Thus, in general, buffer allocation is determined between the partitions that hold the looping-referenced blocks and the other-referenced blocks. The UBM scheme tries to maximize the expected number of total buffer hits by dynamically controlling the allocation so that the marginal gain value of looping references, $MG_{loop}(n)$, and the marginal gain value of other references, $MG_{other}(C - n)$, where C is the cache size, converge to the same value.

3.4 Interaction Among the Three Modules

Figure 6 shows the overall interactions between the three modules of the UBM scheme. Whenever a block reference occurs in the buffer cache, the detection module updates and/or classifies the reference into one of the sequential, looping, or other reference types (step (1) in Figure 6). In this example, assume that a miss has occurred and the reference is classified as an other reference. As a miss has occurred the buffer allocation module is called to get additional buffer space for the referenced block (step (2)). The buffer allocation module would normally compare the marginal gain values of looping and other references and choose the one with a smaller marginal gain value and send a replacement request to the corresponding cache partition as shown in step (3). However, if there is space allocated to a sequential reference, such space is always deallocated first. The cache management module of the selected cache partition decides a victim block to be replaced using its replacement scheme (step (4)) and deallocates the buffer space of the victim block to the allocation module (step (5)). The allocation module forwards this space to the cache partition for other-referenced blocks (step (6)). Finally, the referenced block is fetched from disk into the buffer space (step (7)).

4 Performance Evaluation

In this section and the next, we discuss the performance of the UBM scheme. This section concentrates on the simulation study, while the next section focuses on the implementation study.

In this section, the performance of the UBM scheme is compared with those of the LRU, 2Q, SEQ, EELRU, and OPT schemes through trace-driven simulations[1]. We also compare the performance of the UBM scheme with that of application-controlled file caching through trace-driven simulations with the same multiple application trace used in [4]. We did not compare the performance of the UBM and those of schemes based on recency and/or frequency factors such as FBR, LRU-K,

[1] Though the SEQ and EELRU schemes were originally proposed as page replacement schemes, they can also be used as block replacement schemes.

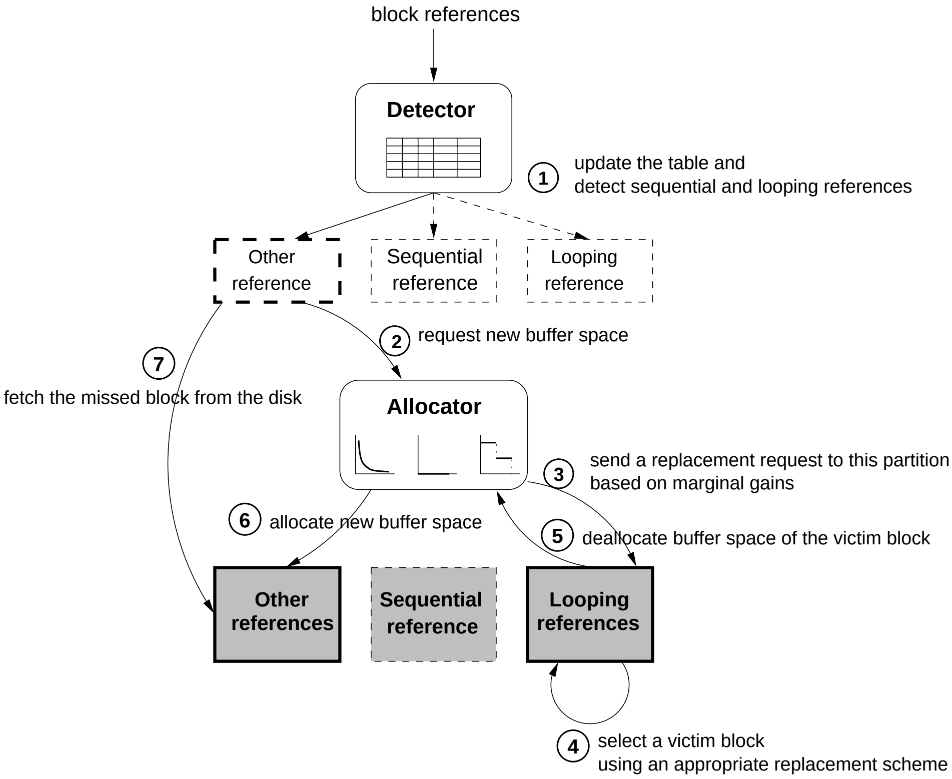

block references
Detector
1 update the table and detect sequential and looping references
Other reference
Sequential reference
Looping reference
2 request new buffer space
7
fetch the missed block from the disk
Allocator
3 send a replacement request to this partition based on marginal gains
6 allocate new buffer space
5 deallocate buffer space of the victim block
Other references
Sequential reference
Looping references
4 select a victim block using an appropriate replacement scheme

Figure 6: Overall structures of the UBM scheme.

Table 1: Characteristics of the traces used.

Trace	Applications executed concurrently	# of references	# of unique blocks
Multi1	cscope, cpp	15858	2606
Multi2	cscope, cpp, postgres	26311	5684
Multi3	cpp, gnuplot, glimpse, postgres	30241	7453

and LRFU since the benefits from the two factors are largely orthogonal and any of the latter schemes can be used to manage other references in the UBM scheme. We first describe the experimental setup and then present the performance results.

4.1 Experimental Setup

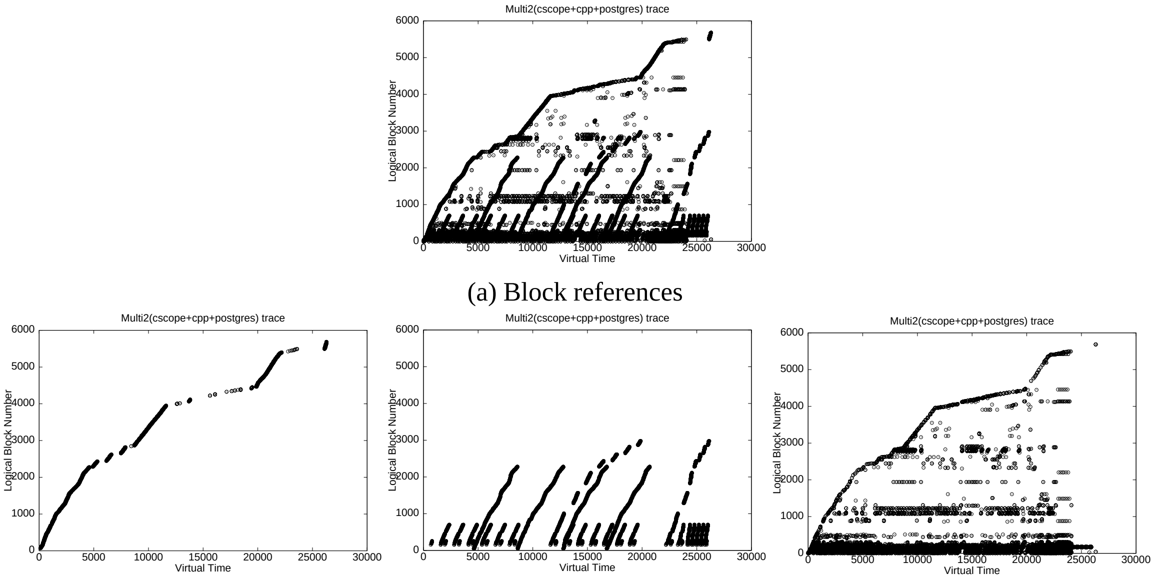
Traces used in our simulations were obtained by concurrently executing diverse applications on the FreeBSD operating system running on an Intel Pentium PC. The characteristics of the applications are described below.

cpp Cpp is the GNU C compiler pre-processor. The total size of C sources used as input is roughly 11MB. During execution, observed block references are sequential and other references.

cscope Cscope is an interactive C source examination tool. The total size of C sources used as input is roughly 9MB. It exhibits looping references with an identical loop period and other references.

glimpse Glimpse is a text information retrieval utility. The total size of text files used as input is roughly 50MB. Its block reference characteristic is diverse - it shows sequential references, looping references with different loop periods, and other references.

gnuplot Gnuplot is an interactive plotting program. The size of raw data used as input is 8MB. Looping references with an identical loop period and



(a) Block references

Sequential references

Looping references

Other references

(b) Classified references

Figure 7: Reference classification results (Trace: $Multi2$).

other references were observed during execution.

postgres Postgres is a relational database system from the University of California at Berkeley. We used join queries among four relations, namely, twothoustup, twentythoustup, hundredthoustup, and twohundredthoustup, which were made from a scaled-up Wisconsin benchmark. The sizes of each relation are approximately 150KB, 1.5MB, 7.5MB, and 15MB. It exhibits sequential references, looping references with different loop periods, and other references.

mpeg_play Mpeg_play is an MPEG player from the University of California at Berkeley. The size of the MPEG video file used as input is 5MB. Sequential references dominate in this application.

We used three multiple application traces in our experiments. They are denoted by $Multi1$, $Multi2$, and $Multi3$, and their characteristics are shown in Table 1.

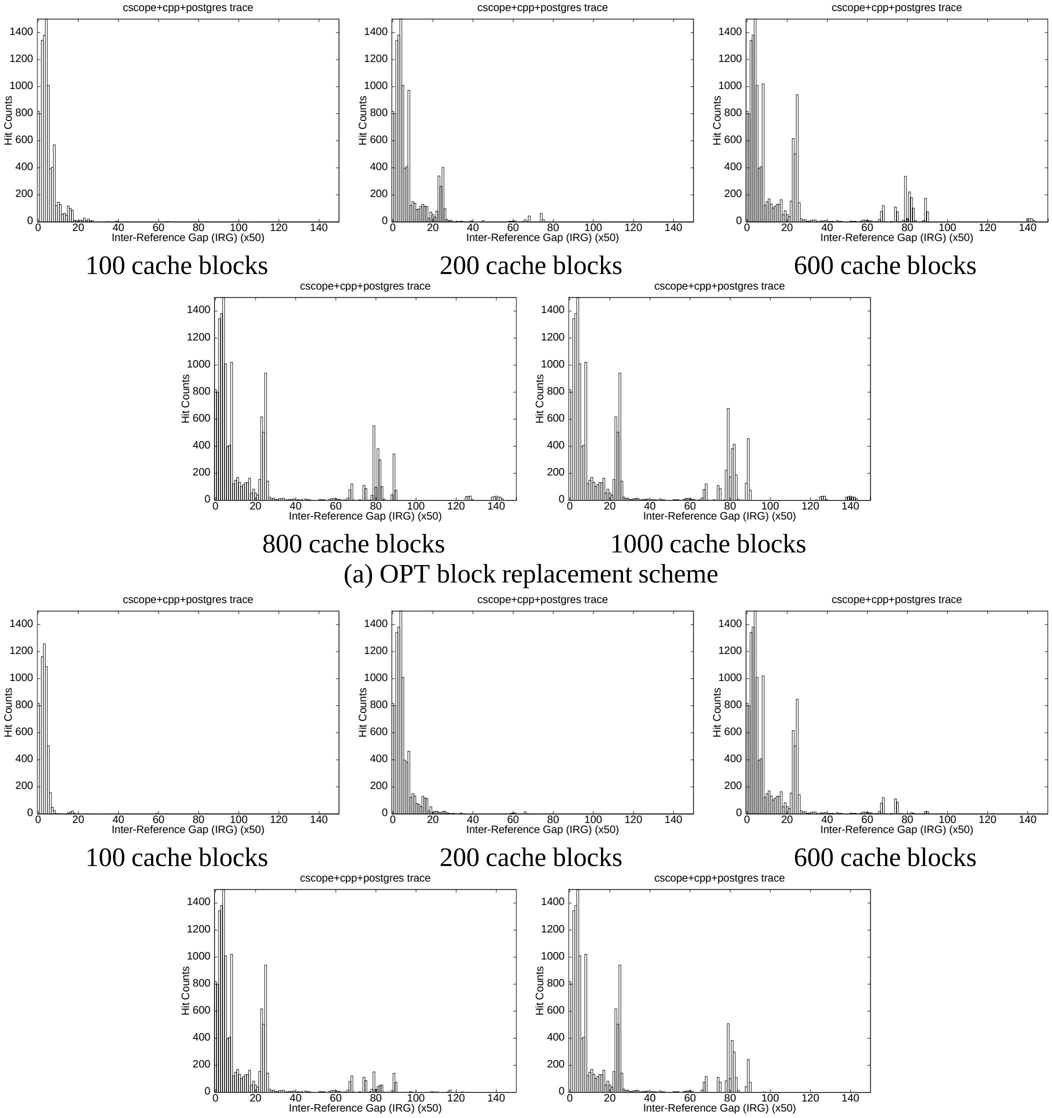
We built simulators for the LRU, 2Q, SEQ, EELRU, and OPT schemes as well as the UBM scheme. Unlike the UBM scheme that does not have any parameters that need to be tuned, the 2Q, SEQ, and EELRU schemes have one or more parameters whose settings may affect the performance. For example, in the 2Q scheme the parameters are the sizes of the $A1in$ and $A1out$ queues. The parameters of the SEQ scheme are threshold values used to choose victim blocks among consecutively missed blocks. Finally, for the EELRU scheme, the early and late eviction points from which a block is replaced, have to be set. In our experiments, we used the values suggested by the authors of each of the schemes.

4.2 Detection Results

Figure 7 shows the classifications resulting from the detection module for the $Multi2$ trace. The x-axis is the virtual time and the y-axis is the logical block number. The figures are space-time graphs with Figure 7(a) showing block references for the whole trace and Figure 7(b) showing how the detection module classified the sequential, looping, and other references from the original references. The results indicate that the UBM scheme accurately classifies the three reference patterns.

4.3 IRG Distribution Comparison of the UBM Scheme with the OPT Scheme

Figure 8 shows the caching behaviors of the OPT and UBM schemes for the $Multi2$ trace. Compare these results with those shown in Figures 1(b) and 1(c), which use the same trace. Recall that these graphs show the



100 cache blocks 200 cache blocks 600 cache blocks

800 cache blocks 1000 cache blocks

(a) OPT block replacement scheme

100 cache blocks 200 cache blocks 600 cache blocks

800 cache blocks 1000 cache blocks

(b) Unified buffer management scheme

Figure 8: Caching behaviors of the OPT and UBM schemes.

hit counts in the buffer cache using IRG distributions of referenced blocks. We note that the UBM scheme is very closely mimicking the behavior of the OPT scheme.

4.4 Performance Comparison of the UBM Scheme with Other Schemes

Figure 9 shows the buffer hit ratios of the UBM scheme and other replacement schemes as a function of cache size with the block size set to 8KB. For most cases, the UBM scheme shows the best performance. Further analysis for each of the schemes is as follows.

The SEQ scheme shows fairly stable performance for all cache sizes. The reason behind its good performance is that it quickly replaces sequentially-referenced blocks that miss in the buffer cache. However, since the scheme does not consider looping references, it shows worse performance than the UBM scheme.

The 2Q scheme shows better performance than the LRU scheme for most cases because it quickly replaces sequentially-referenced blocks and looping-referenced blocks with long loop periods. However, when the cache size is large (caches with 1800 or more blocks for the $Multi1$ trace, 2800 or more blocks for the $Multi2$ trace, and 3600 or more blocks for the $Multi3$ trace), the scheme shows worse performance than the LRU scheme. There are two reasons behind this. First, since the scheme replaces all newly referenced blocks after holding it in the buffer cache for a short time, whenever these blocks are re-referenced, additional misses occur. The ratio of these additional misses to total misses increases as the cache size increases resulting in a significant impact on the performance of the buffer cache. Second, the scheme does not distinguish between looping-referenced blocks with different loop periods. The performance of the 2Q scheme does not gradually increase with the cache size, but rather surges beyond some cache size and then holds steady. Also the 2Q scheme exhibits rather anomalous behavior for the $Multi2$ and $Multi3$ traces. When the cache sizes are about 2200 blocks and 2800 blocks for $Multi2$ and $Multi3$, respectively, the buffer hit ratio of the scheme decreases as the cache size increases. A careful inspection of results reveals that when the cache size increases the $A1out$ queue size increases accordingly and this results in a situation where blocks that should not be promoted are promoted to the main buffer cache leading to such an anomaly.

The EELRU scheme shows similar or better performance than the LRU scheme as the cache size increases. However, since the scheme chooses a victim block to be replaced based on aggregate recency distributions of referenced blocks, it does not replace quickly enough the sequentially-referenced blocks and looping-referenced blocks that have long loop periods. Further, like the 2Q scheme, it does not distinguish between looping-referenced blocks with different loop periods. Hence, it does not fair well compared to the UBM scheme.

The LRU scheme shows the worst performance for most cases because it does not give any consideration to the regularities of sequential and looping references.

Finally, the UBM scheme replaces sequentially-referenced blocks quickly and holds the looping-referenced blocks in increasing order of loop periods based on the notion of marginal gains as the cache size increases. Consequently, the UBM scheme improves the buffer hit ratios by up to 57.7% (for the $Multi1$ trace with 1400 blocks) compared to the LRU scheme with an average increase of 29.2%.

4.5 Results of Dynamic Buffer Allocation

Figure 10(a) shows the distribution of the buffers allocated to the partitions that hold sequential, looping, and other references as a function of (virtual) time when the buffer cache size is 1000 blocks. Until time 2000, the

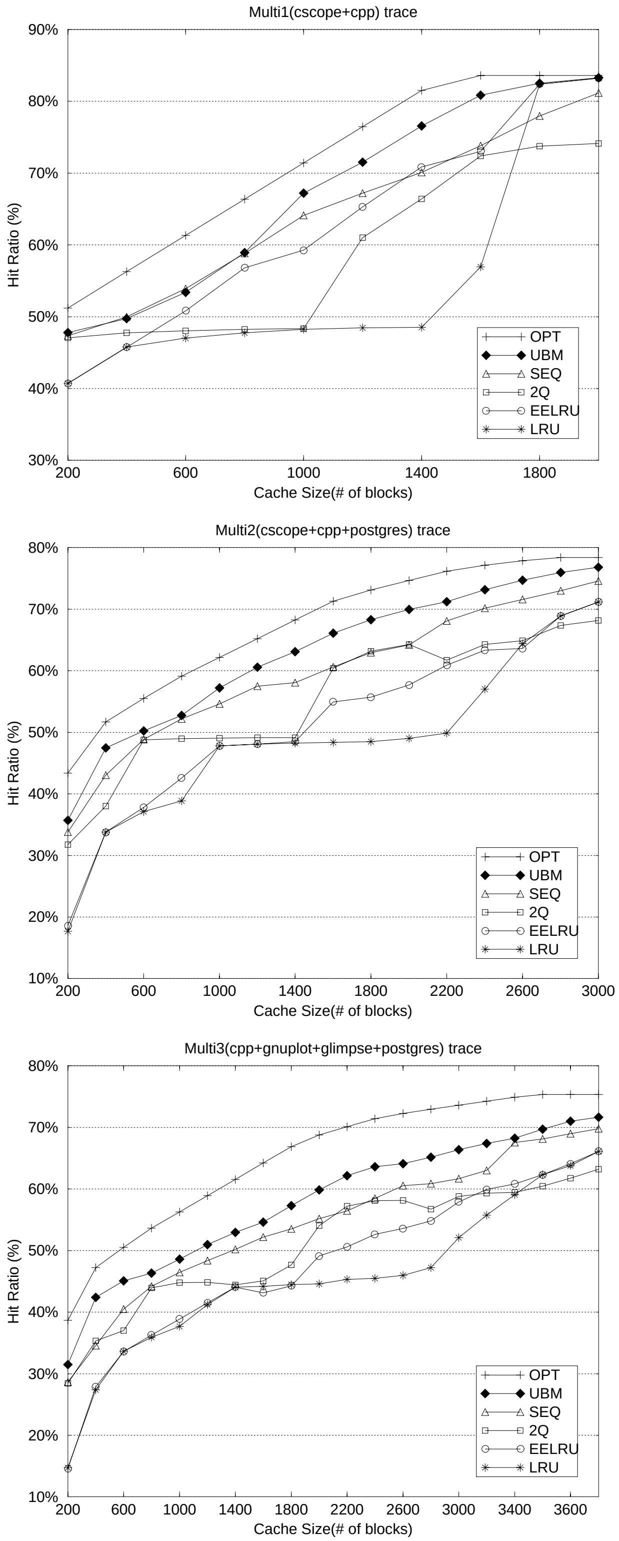
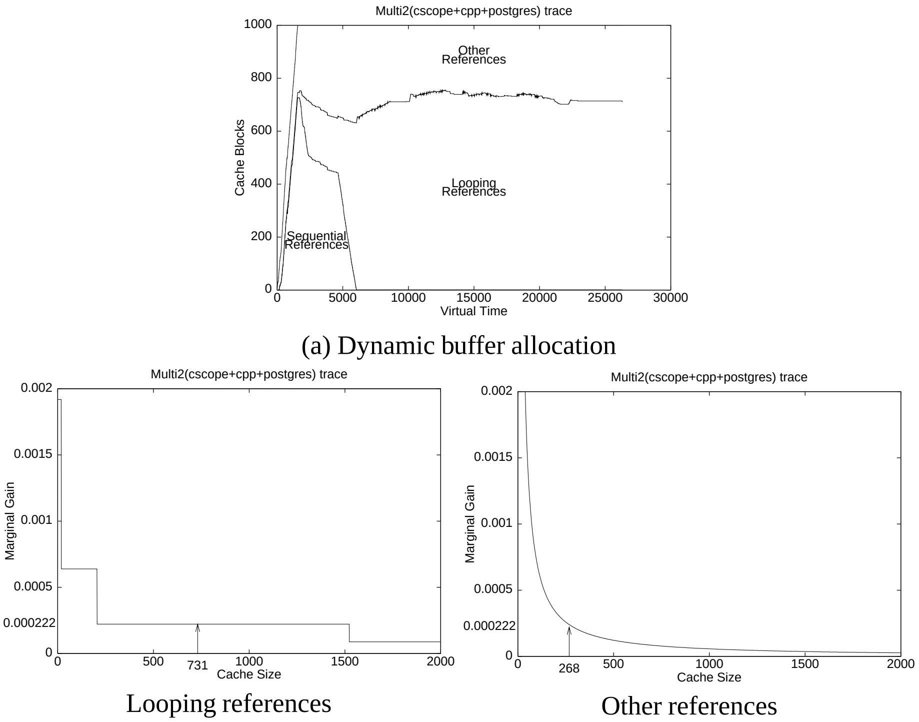


Figure 9: Performance comparison of the UBM scheme with other schemes.



(a) Dynamic buffer allocation

Looping references

Other references

(b) Expected marginal gain values (time: 20000)

Figure 10: Results of dynamic buffer allocation (cache size: 1000 blocks, trace: $Multi2$).

buffer cache is not fully utilized. Hence, buffers are allocated to any reference that requests it, resulting in the partition that holds sequentially-referenced blocks being allocated a considerable number of blocks. After time 2000, all the buffers have been consumed, and hence, the number of buffers allocated to the partition for sequentially-referenced blocks decreases, while allocations to the partitions for looping and other references start to increase. As a result, at around time 6000, only one buffer is allocated to the partition for sequentially-referenced blocks. From about time 10000, the allocations to the partitions for the looping and other references converge to a steady-state value.

Figure 10(b) shows marginal gains as a function of allocated buffers of looping and other references that are calculated at time 20000 of Figure 10(a). Since there are several looping references with different loop periods in the $Multi2$ trace, from the left figure, we can see that the expected marginal gain values of the looping references, $MG_{loop}(n)$, decrease step-wise as the number of allocated buffers, n, increases. The figure on the right shows the expected marginal gain values of the other references, $MG_{other}(n)$, that decrease gradually. The UBM scheme dynamically controls the number of allocated buffers to each partition so that the marginal gain values of the two partitions converge to the same value. At time 20000, the two marginal gain values converge to 0.000222 and the number of allocated buffers to the partitions for the looping and other references is 731 blocks and 268 blocks, respectively.

4.6 Performance Comparison of the UBM Scheme with Application-controlled File Caching

The application-controlled file caching (ACFC) scheme [4] is a user-hint based approach to buffer cache management, which is in contrast to the UBM scheme that requires no such hints. To compare the performance of these two schemes, we used the ULTRIX multiple application ($postgres + cscope+$*linking the kernel*) trace in [4].

Figure 11 shows the buffer hit ratios of the UBM scheme, two ACFC schemes, namely, *ACFC(HEURISTIC)* and *ACFC(RMIN)*, and the LRU, 2Q, SEQ, EELRU, and OPT schemes when the cache size increases from 4M to 16M. The *ACFC(HEURISTIC)* scheme uses user-level hints for each application, while the *ACFC(RMIN)* scheme uses

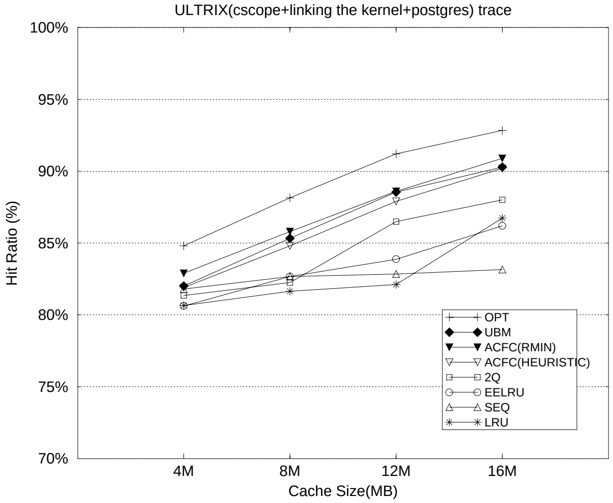


Figure 11: Performance comparison of the UBM scheme with the application-controlled file caching scheme.

the optimal off-line strategy for each application. The results for the two ACFC schemes were borrowed from [4] while the results for all the other schemes were obtained from simulations. The results show that the hit ratios of the UBM scheme, which does not make use of any external hints, are comparable to those of the *ACFC(RMIN)* scheme and higher than those of the *ACFC(HEURISTIC)* scheme.

4.7 Warm/Cold Caching Effects

All experiments so far were done with cold caches. To evaluate the performance of the UBM scheme at steady-state, we performed additional simulations with warmed-up caches. In the experiments, we run initially a long-run workload ($sdet\_benchmark$[2]) through the buffer cache and after the cache was warmed up, we run both the long-run workload and the target workload ($cscope + cpp + postgres$) concurrently. The cache statistics were collected only after the cache was warmed up.

Experimental results that show the warm/cold caching effects of the UBM scheme are given in Figure 12 and Table 2. The graphs of Figure 12 show that when the cache size is small, the performance improvements by the UBM scheme with warmed-up caches over the LRU scheme are similar to those with cold caches. As the cache size increases, however, the performance increase by the UBM scheme with warmed-up caches is reduced when compared with the performance increase with cold caches since sequential references are not allowed to be cached at all. Specifically, when the cache is warmed up, blocks belonging to sequential references are not allowed to be cached. If those blocks are re-referenced with a regular interval in the future (i.e., if they turn into looping references), the UBM scheme has to reread them from the disks. In cold caches, however, many of them are reread from the cache partition that holds sequentially-referenced blocks because when the cache is cold, more than one block can be allocated to the partition for the sequentially referenced blocks.

The resulting performance degradation, however, is not significant as we can see from Table 2 that summarizes the performance improvements by the UBM scheme for both cold caches and warmed-up caches - the difference in the average performance improvement between cold caches and warmed-up caches is less than 1%. Although the overall performance degradation is negligible, the additional misses may adversely affect the performance perceived by the user due to increased start-up time. As future work, we plan to explore an allocation scheme where blocks are allocated even for sequential references based on the probability that a sequential reference will turn into a looping reference.

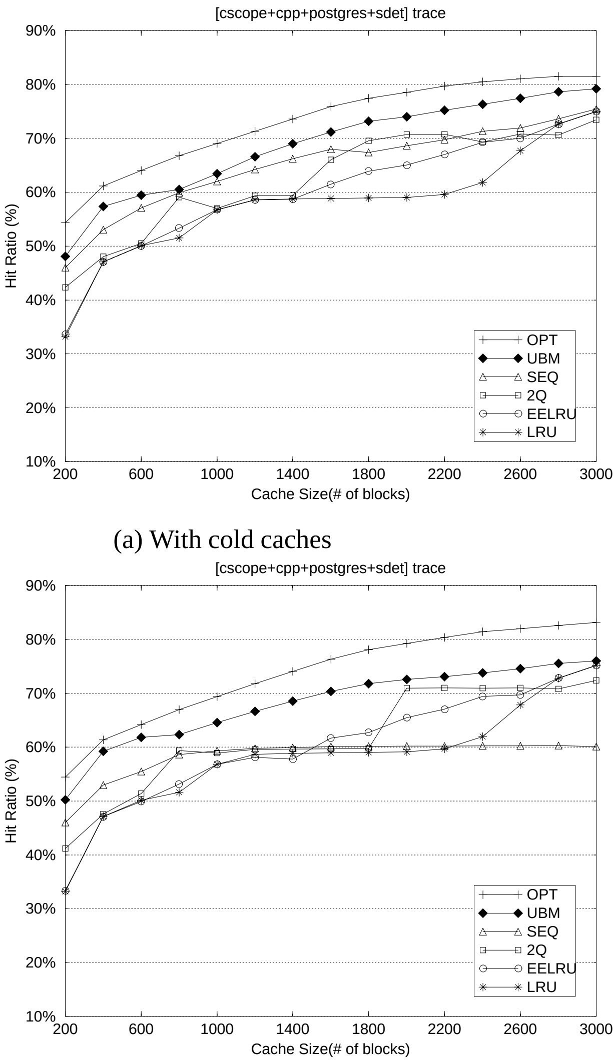
5 Implementation of UBM in the FreeBSD Operating System

The UBM scheme was integrated into the FreeBSD operating system running on a 133MHz Intel Pentium PC with 128MB RAM and a 1.6GB Quantum Fireball hard disk. For the experiments, we used five real applications (cpp, $cscope$, $glimpse$, $postgres$, and $mpeg\_play$) that were explained in Section 4. We ran several combinations of three or more of these applications concurrently and measured the elapsed time of each application under the UBM and SEQ schemes as well as under the built-in LRU scheme when the cache sizes are 8MB, 12MB, and 16MB with the block size set to 8KB.

5.1 Performance Measurements

Figure 13 shows the elapsed time of each individual application under the UBM, SEQ and LRU schemes. As expected, the UBM scheme shows better performance than the LRU scheme. In the figure, since the $postgres$ and $cscope$ applications access large files repeatedly with a regular interval, they show better improvement

[2] Sdet is the SPEC SDET benchmark that simulates a multiprogramming environment.



(a) With cold caches

(b) With warmed-up caches

Figure 12: The cold/warm caching effects of the UBM scheme in trace-driven simulations (Trace: $cscope + cpp + postgres + sdet$).

in the elapsed time than other applications. On the other hand, the $mpeg\_play$ application does not show as much improvement because it accesses a large video file sequentially.

Overall, the UBM scheme reduces the elapsed time by up to 67.2% (the elapsed time of the $postgres$ application for the $cpp + postgres + cscope + mpeg\_play$ case with 16MB cache size) compared to the LRU scheme with an average improvement of 28.7%. We note that improvements by the UBM scheme on the elapsed time are comparable to those on the buffer hit ratios we observed in the previous section.

There are two main benefits from using the UBM scheme. The first is from detecting looping references,

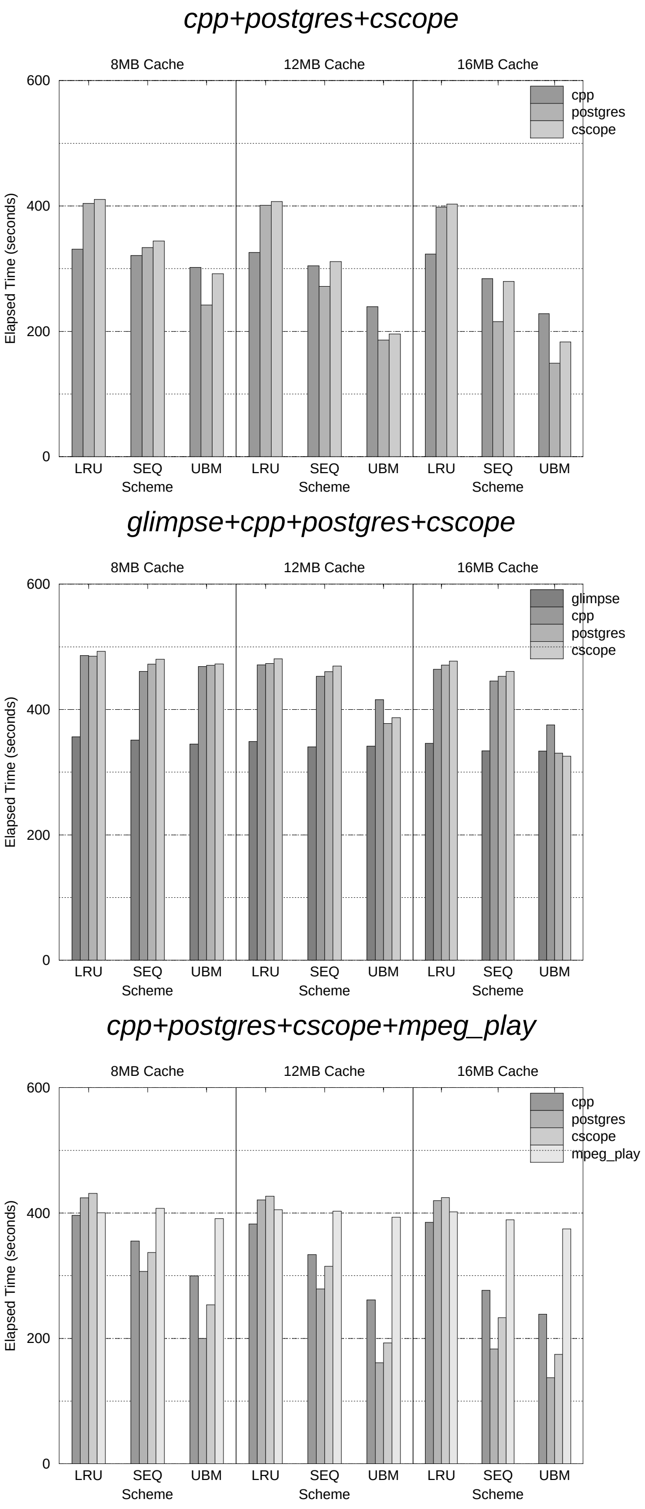


Figure 13: Performance of the UBM scheme integrated into the FreeBSD.

Table 2: Comparison of performance improvements of the UBM scheme compared to th e LRU scheme (Trace: $cscope + cpp + postgres + sdet$).

Improvements with cold caches	Improvements with warmed-up caches
avg. 19.6% (max. 26.2%)	avg. 18.9% (max. 22.6%)

managing them by a period-based replacement policy, and allocating buffer space to them based on marginal gains. The second is from giving preference to blocks belonging to sequential references when a replacement is needed. To quantify these benefits, we compared the UBM scheme with the SEQ scheme. The results in Figure 13 show that there is still a substantial difference in the elapsed time between the UBM scheme and the SEQ scheme indicating that the benefit from carefully handling looping references is significant.

5.2 Effects of Run-Time Overhead of the UBM Scheme

To measure the run-time overhead of the UBM scheme, we executed a CPU-bound application (cpu_bound) that executes an ADD instruction repeatedly, along with the other applications.

Figure 14 shows the run-time overhead of the UBM scheme for two combinations of applications, namely, $cpu\_bound+cpp+postgres+cscope$ and $cpu\_bound+glimpse+cpp+postgres+cscope$ when the cache size is 12MB. The elapsed time of the cpu_bound application increases slightly by around 5% when using the UBM scheme compared with that when using the LRU scheme. A major source of this overhead comes from the operations to manipulate the ghost buffers and to calculate the marginal gain values. Currently, we integrated the UBM scheme into the FreeBSD in a straightforward manner without any optimization, and hence we expect there is still much room for further optimizing the performance.

The UBM scheme also has the space overhead of maintaining ghost buffers in the kernel memory. The maximum size of the LRU stack to be maintained including ghost buffers is limited by the total number of buffers in the file system. Therefore, the space overhead due to ghost buffers is proportional to the difference between the total number of buffers in the system and the number of allocated buffers for other references. In the current implementation, each ghost buffer requires 13 bytes.

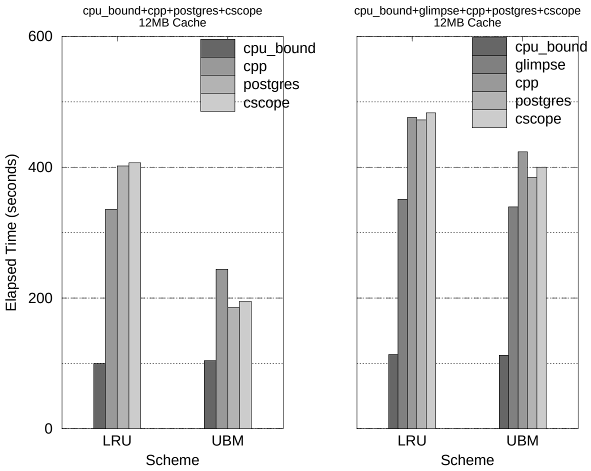


Figure 14: Run-time overhead of the UBM scheme.

6 Conclusions and Future Work

This paper starts from the observation that the widely used LRU replacement scheme does not make use of regularities present in the reference patterns of applications, leading to degraded performance. The Unified Buffer Management (UBM) scheme is proposed to resolve this problem. The UBM scheme automatically detects sequential and looping references and stores the detected blocks in separate partitions of the buffer cache. These partitions are managed by appropriate replacement schemes based on the properties of their detected patterns. The allocation problem among the partitions is also tackled with the use of the notion of marginal gains.

To evaluate the performance of the UBM scheme, experiments were conducted using both trace-driven simulations with multiple application traces and an implementation of the scheme in the FreeBSD operating system. Both simulation and implementation results show that 1) the UBM scheme accurately detects almost all the sequential and looping references, 2) the UBM scheme manages sequential and looping-referenced blocks similarly to the OPT scheme, and 3) the UBM scheme shows substantial performance improvements increasing the buffer hit ratio by up to 57.7% (with an average increase of 29.2%) and reducing, in an actual implementation in the FreeBSD operating system, the elapsed time by up to 67.2% (with an average of 28.7%) compared to the LRU scheme, for the workloads we considered.

As future research, we are attempting to apply to other references the Least Recently/Frequently Used (LRFU) scheme based on both recency and frequency factors rather than the LRU scheme, which is based on the recency factor only. Also, as automatic detection of sequential and looping references is possible, we are investigating the possibility of further enhancing performance through prefetching techniques that exploit these regularities as was attempted in [10] for informed prefetching and caching. Finally, we plan to extend the techniques presented in this paper to systems that integrate virtual memory and file cache management.

Acknowledgements

The authors are grateful to the anonymous reviewers for their constructive comments and to F. Douglis, our “shepherd”, for his guidance and help during the revision process. The authors also want to thank P. Cao for providing the ULTRIX trace used in the experiments.

References

[1] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, 1990.

[2] 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.

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

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

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

[6] D. Lee, J. Choi, 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 on Measurement and Modeling of Computer Systems*, pages 134–143, 1999.

[7] E. G. Coffman, JR. and P. J. Denning. *Operating Systems Theory*. Prentice-Hall International Editions, 1973.

[8] G. Glass and P. Cao. Adaptive Page Replacement Based on Memory Reference Behavior. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–126, 1997.

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

[10] 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.

[11] D. Thiebaut, H. S. Stone, and J. L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.

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

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

[14] J. R. Spirn. *Program Behavior: Models and Measurements*. New York: Elsevier-North Holland, 1977.