RACE: A Robust Adaptive Caching Strategy for Buffer Cache

Yifeng Zhu, *Member*, *IEEE*, and Hong Jiang, *Member*, *IEEE*

Abstract—Although many block replacement algorithms for buffer caches have been proposed to address the well-known drawbacks of the LRU algorithm, they are not robust and cannot maintain a consistent performance improvement over all workloads. This paper proposes a novel and simple replacement scheme, called the Robust Adaptive buffer Cache management schemE (RACE), which differentiates the locality of I/O streams by actively detecting access patterns that are inherently exhibited in two correlated spaces, that is, the discrete block space of program contexts from which I/O requests are issued and the continuous block space within files to which I/O requests are addressed. This scheme combines the global I/O regularities of an application and the local I/O regularities of individual files that are accessed in that application to accurately estimate the locality strength, which is crucial in deciding which blocks are to be replaced upon a cache miss. Through comprehensive simulations on 10 real-application traces, RACE is shown to have higher hit ratios than LRU and all other state-of-the-art cache management schemes studied in this paper.

Index Terms—Operating systems, file systems management, memory management, replacement algorithms.

1 Introduction

This paper presents a novel approach for buffer cache management, called the Robust Adaptive Caching strategy for buffer cache management schemE (RACE), which is motivated by the limitations of existing solutions and the need to further improve the buffer cache hit rate, a significant factor affecting the performance of the supported file system given the relatively very high buffer cache miss penalties. RACE is shown to outperform all existing solutions significantly in most cases. In this section, we first describe the limitations of existing solutions and then present the main motivations for this work, followed by an outline of the major contributions of the paper.

1.1 The Limitations of LRU and Recent Solutions

Designing an effective block replacement algorithm is an important issue in improving file system performance. In most real systems, the replacement algorithm is based on the Least Recently Used (LRU) scheme [1], [2] or its clock-based approximation [3]: Upon a cache miss, the block whose last reference was the earliest among all cached blocks is replaced. LRU has the advantages of simple implementation and constant space and time complexity. Although it has been theoretically verified that LRU can absorb the maximum number of I/O references under a spectrum of workloads that can be represented by the independent reference model [4], in reality, LRU often suffers severely from two pathological cases:

- *Scan pollution.* After a long series of sequential accesses to one-time-use-only (cold) blocks, many frequently accessed blocks may be evicted from the cache immediately, leaving all of these cold blocks occupying the buffer cache for an unfavorable amount of time, thus resulting in a waste of the memory resources. A wise replacement strategy should consider the reference frequency of each block and, hence, can distinguish hot data from cold data.
- *Cyclic access to a large working set.* A large number of applications, especially those in the scientific computation domain, exhibit a looping access pattern. When the total size of repeatedly accessed data is larger than the cache size, LRU always evicts the blocks that will be revisited in the nearest future, resulting in perpetual cache misses. For example, when repeatedly accessing a file that has 100 blocks, an LRU cache with 99 blocks always evicts the very block that will be referenced next and leads to a zero-hit ratio. A clever strategy would observe this access with long-term locality and only generate cache misses for the references to the block that is least accessed.

To address the limitations of the LRU scheme, several novel and effective replacement algorithms [5], [6], [7], [8] have been proposed to avoid the two pathological cases described above by using advanced knowledge of the unusual I/O requests. Specifically, they exploit the patterns exhibited in I/O workloads, such as sequential scan and periodic loops, and apply specific replacement polices that can best utilize the cache under that reference pattern.

According to the level at which the reference patterns are observed, these algorithms can be divided into three categories: 1) At the application level, Detection Adaptive

- *Y. Zhu is with the Department of Electrical and Computer Engineering, University of Maine, 5708 Barrows Hall, Orono, ME 04469-5708. E-mail: zhu@eece.maine.edu.*
- *H. Jiang is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115. E-mail: jiang@cse.unl.edu.*

Manuscript received 22 May 2006; revised 11 Jan. 2007; accepted 28 June 2007; published online 24 July 2007.
Recommended for acceptance by A. Gonzalez.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0195-0506.
Digital Object Identifier no. 10.1109/TC.2007.70788.

Replacement (DEAR) [6] observes the patterns of references issued by a single application, assuming that the I/O patterns of each application are consistent, 2) at the file level, UBM [5], [9] examines the references to the same file with an assumption that a file is likely to be accessed with the same pattern in the future, and 3) at the program context level, PCC [7] and AMP [8] separate the I/O streams into substreams by program context and detect the patterns in each substream, assuming that a single program context tends to access files with the same pattern in the future.

To best exploit the access patterns, the design space centers on investigating an automatic pattern detection technique that should satisfy the following requirements:

- **Accuracy.** Applications often have certain I/O access patterns. An accurate detection of access patterns serves as the basis for quantitatively identifying the locality of accessed blocks and tuning caching policies accordingly. A misclassification of an access stream may increase the number of disk accesses by evicting useful blocks and taking up memory space. In addition, the detection algorithm should be able to detect not only reference patterns presented explicitly in the consecutive address space, but also implicit patterns in a nonconsecutive way. For example, a stream of references to blocks with a set of random addresses may belong to a sequential pattern category.
- **Responsiveness.** Real applications within various phases of execution may exhibit different reference patterns. The cache replacement algorithm should adapt to different behaviors within one application and, thus, a good online detection algorithm is required to quickly reflect the transition of access patterns. A detection approach based on aggregate statistical measures of program behavior, as used by PCC and AMP, tends to have a large amount of inertia or reluctance and may not responsively detect local patterns, although it can correctly recognize global patterns.
- **Stability.** A detection-based caching system applies different replacement policies for different reference patterns. To achieve this goal, a buffer cache is divided into multiple partitions and blocks from different patterns are stored in their corresponding partitions. An unstable detection scheme swings a block from different patterns and moves it repeatedly among cache partitions accordingly. In an asynchronous environment with multiple threads, moving a block between the links of different partitions relies on locks to ensure consistency and correctness, which can be quite costly. A stable classification can eliminate the *lock contention* and reduce the overhead of cache maintenance.

The key in the design of an effective pattern detection scheme is to strike an optimal trade-off among the above three requirements. A scheme with better stability may sacrifice its classification accuracy and responsiveness and vice versa. As strongly suggested by the results obtained from the extensive simulations conducted in this study, none of the currently existing schemes is able to maintain a good balance among the three requirements:

1. Application-level detection [6] has good stability but suffers in accuracy and responsiveness since many applications access multiple files and exhibit a mixture of access patterns, as shown in [5] and later in this paper.
2. File-level detection [5], [9] has a smaller observation granularity than the application-based approach but has two main drawbacks that limit its classification accuracy. First, a training process needs to be performed for each new file and is thus likely to cause a misclassification for the references targeted at new files. Second, to reduce the running overhead, the access patterns presented in small files are ignored. Nevertheless, this approach tends to have good responsiveness and stability due to the fact that most files tend to have stable access patterns, although large database files may show mixed access patterns.
3. Program-context-level detection [7], [8] trains only for each program context and has a relatively shorter learning period than the file-based one. Although it can make correct classification for new files after training, it classifies the accesses to all files touched by a single program context into the same pattern category and thus limits the detection accuracy. In addition, it has problems in responsiveness and stability. It bases its decision on aggregate statistical information and is thus not sensitive to pattern changes. The stability problem is caused by the fact that, in real applications, as explained in Section 3, multiple program contexts may access the same set of files but exhibit different patterns if observed from the program-context point of view.

1.2 Contributions of This Paper

This paper makes three contributions. First, we collect the I/O traces for 10 real applications and investigate I/O access patterns in two correlated spaces: the program context space, from which I/O operations are issued, and the file space, to which I/O requests are addressed. Second, our comprehensive pattern study in real applications reveals pathological behavior related to existing state-of-the-art cache replacement algorithms, including a file-level detection method named UBM [5], [9] and two program-context-level detection methods named PCC [7] and AMP [8]. This observation motivates us to design a novel, robust, and adaptive cache replacement scheme, which is presented in this paper. Our new scheme, called RACE, which has a time complexity of O(1), can accurately detect access patterns exhibited in both the discrete block space accessed by a program context and the continuous block space within a specific file, which leads to more accurate estimations and more efficient utilizations of the strength of data locality. We show that our design can effectively combine the advantages of both file-based and program-context-based caching schemes and best satisfy the requirements of accuracy, responsiveness, and stability. Third, we conduct extensive trace-driven simulations by

using 10 different types of real-life workloads and show that RACE substantially improves the absolute hit ratio of LRU by as much as 56.9 percent, with an average of 15.5 percent. RACE outperforms UBM, PCC, and AMP in absolute hit ratios by as much as 22.5 percent, 42.7 percent, and 39.9 percent, with an average of 3.3 percent, 6.6 percent, and 6.9 percent, respectively. These gains in absolute hit ratios by RACE are likely to have significant performance implications in the applications' response times.

1.3 Outline of This Paper

The rest of this paper is organized as follows: Section 2 briefly reviews relevant studies in buffer cache management. Section 3 explains our RACE design in detail. Section 4 presents the trace-driven evaluation method and Section 5 evaluates the performance of RACE and other related algorithms and discusses the experimental results. Finally, Section 6 concludes this paper.

2 Related Work on Buffer Cache Replacement Strategies

A large number of replacement algorithms have been proposed in the last few decades. These algorithms can be classified into three categories: 1) replacement algorithms that incorporate longer reference histories than LRU, 2) replacement algorithms that rely on application hints, and 3) replacement algorithms that actively detect the I/O access patterns. The following sections describe the theoretically optimal replacement algorithm, followed by representative replacement algorithms in the above three categories.

2.1 Offline Optimal Replacement

Offline optimal policy (OPT) [10], [11] replaces the block whose next reference is farthest in the future. This policy is not realizable in actual computer systems since it requires complete knowledge of future block references. However, it provides a useful upper bound on the achievable hit ratio of all practical cache replacement policies.

2.2 Deeper-History-Based Replacement

To avoid the two pathological cases in LRU, as described in the previous section, many cache replacement strategies are proposed to incorporate the "frequency" information when making a replacement decision. A common characteristic of such strategies is that all of them keep longer history information than LRU. A chronological list of these algorithms by date of publication includes LRU-K [12], 2Q [13], LRFU [14], EELRU [15], MQ [16], Low LIRS [17], and ARC [18]. A brief overview of each algorithm is given below.

For every block x, LRU-K [12] dynamically records the Kth backward distance, which is defined as the number of references during the time period from the last Kth reference to x to the most recent reference to x. A block with the maximum Kth backward distance is dropped to make space for missed blocks. LRU-2 is found to best distinguish infrequently accessed (cold) blocks from frequently accessed (hot) blocks. The time complexity of LRU-2 is $O(\log_2 n)$, where n is the number of blocks in the buffer.

The 2Q scheme [13] is proposed to perform similarly to LRU-K but with considerably lower time complexity. It achieves quick removal of cold blocks from the buffer by using a FIFO queue $A1_{in}$, an LRU queue Am, and a "ghost" LRU queue $A1_{out}$ that holds no block contents except for block identifiers. A missed block is initially placed in $A1_{in}$. When a block is evicted from $A1_{in}$, this block's identifier is added to $A1_{out}$. If a block in $A1_{out}$ or $A1_{in}$ is rereferenced, this block is promoted to Am.

LRFU [14], [19] endeavors to replace a block that is both least recently and least frequently used. A weight $C(x)$ is associated with every block x and a block with the minimum weight is replaced:

$$C(x) = \begin{cases} 1 + 2^{-\lambda} C(x) & \text{if } x \text{ is referenced at time } t \\ 2^{-\lambda} C(x) & \text{otherwise,} \end{cases} \quad (1)$$

where λ, $0 \le \lambda \le 1$, is a tunable parameter and, initially, $C(x) = 0$. LRFU reduces to LRU when $\lambda = 1$ and to LFU when $\lambda = 0$. By controlling λ, LRFU represents a continuous spectrum of replacement strategies that subsume LRU and LFU. The time complexity of this algorithm ranges between $O(1)$ and $O(\log n)$, depending on the value of λ.

EELRU [15] builds a history queue that records the identifiers of recently evicted blocks and uses this queue to detect the recency of evicted blocks. Based on the recency distributions of referenced blocks, it dynamically changes its replacement strategies. Specifically, it performs LRU replacement by default but diverges from LRU and arbitrarily evicts some pages early to allow not-recently-touched blocks to remain longer when EELRU detects that many pages fetched recently have just been evicted.

MQ [16] uses $m+1$ LRU queues (typically $m = 8$), $Q_0, Q_1, \ldots, Q_{m-1}$, and Q_{out}, where Q_i contains blocks that have been referenced at least 2^i times but not more than 2^{i+1} times recently and Q_{out} contains the identifiers of blocks evicted from Q_0 in order to remember access frequencies. On a cache hit in Q_i, the frequency of the accessed block is incremented by 1 and this block is promoted to the most recently used position of the next level of queue if its frequency is equal to or larger than 2^{i+1}. MQ associates each block with a timer that is set to $currentTime + lifeTime$. $lifeTime$ is a tunable parameter that is dependent upon the buffer size and workload. It indicates the maximum amount of time that a block can be kept in each queue without any access. If the timer of the head block in Q_i expires, this block is demoted into Q_{i-1}. The time complexity of MQ is $O(1)$.

LIRS [17], [20] uses the distance between the last and second-to-the-last references to estimate the likelihood of the block being rereferenced. It categorizes a block with a large distance as a cold block and a block with a small distance as a hot block. A cold block is chosen to be replaced on a cache miss. LIRS uses two LRU queues with variable sizes to measure the distance and also provides a mechanism for allowing a cold block to compete with hot blocks if the access pattern changes and this cold block is frequently accessed recently. The time complexity of LIRS is $O(1)$. Clock-Pro [21] is an approximation of LIRS.

For a given cache size c, ARC [18], [22] uses two LRU lists, L_1 and L_2, and they collectively contain c physical pages and c identifiers of recently evicted pages. Although

all blocks in L_1 have been referenced only once recently, those in L_2 have been accessed at least twice. The cache space is allocated to the L_1 and L_2 lists adaptively according to their recent miss ratios. More cache space is allocated to a list if there are more misses in this list. The time complexity of ARC is $O(1)$. CAR [23] is a variant of ARC based on clock algorithms.

All of the above replacement algorithms base their cache replacement decisions on a combination of recency and reference frequency information of accessed blocks. However, they are not able to explicitly exploit the regularities exhibited in past behaviors or histories, such as looping or sequential references. Thus, their performance is confined due to their limited knowledge of I/O reference regularities [7].

2.3 Application-Controlled Replacement

Application-informed caching management schemes are proposed in ACFS [24] and TIP [25] and they rely on programmers to insert useful hints to inform operating systems of future access patterns. ACFS uses a two-level cache scheme, where a global cache management policy decides which application should give up a cache block upon a miss and the local policy decides intelligently which block of that application should be evicted by applying application-specific knowledge. TIP partitions the cache into three logical domains for hinted-prefetching blocks, hinted-caching blocks, and unhinted-caching blocks, respectively. Based on the estimated cost benefits, TIP dynamically allocates file buffers among those three domains. To free the programmer from the onerous burden, Profet [26] exploits a compiler-based technique to automatically insert crucial hints to facilitate I/O prefetching. However, this technique cannot achieve a satisfactory performance level if the I/O access pattern is only known at runtime. Artificial intelligence tools [27] are proposed to learn these I/O patterns at execution time and thus obtain the hints dynamically.

2.4 Active Pattern-Detection-Based Replacement

Depending on the level at which patterns are detected, the pattern-detection-based replacement can be classified into four categories:

1. block-level patterns,
2. application-level patterns,
3. file-level patterns, and
4. program-context-level patterns.

An example of the block-level pattern detection policy is SEQ [28], which detects the long sequences of page cache misses and applies the Most Recently Used (MRU) [29] policy to such sequences to avoid scan pollution.

At the application level, DEAR [6] periodically classifies the reference patterns of each individual application into four categories: sequential, looping, temporally clustered, and probabilistic. DEAR uses MRU as the replacement policy to manage the cache partitions for looping and sequential patterns, LRU for the partition of the temporally clustered pattern, and LFU for the partition of the probabilistic pattern. The time complexity of DEAR is $O(n \log n)$, where n is the number of distinct blocks referenced in the detection period.

At the file level, the UBM [5], [9] scheme separates the I/O references according to their target files and automatically classifies the access pattern of each individual file into one of these three categories: *sequential references*, *looping references*, and *other references*. It divides the buffer cache into three partitions, one each for blocks belonging to each pattern category, and then uses different replacement policies on different partitions. For blocks in the sequentially referenced partition, the MRU replacement policy is used since those blocks are never revisited. For blocks in the periodically referenced partition, the block with the longest period is replaced first and the MRU block replacement is used among blocks with the same period. For blocks that belong to neither the sequential partition nor the looping partition, a conventional algorithm such as LRU is used.

At the program context level, the PCC [7] algorithm exploits virtual program counters (PCs) that are exhibited in the application's binary execution codes to classify the program signatures into the same three categories as UBM and then uses the same replacement policies for these categories, respectively. Although UMB classifies the I/O access patterns based on files, PCC classifies the patterns based on the virtual PCs of the I/O instructions in the program code. The AMP caching scheme [8] inherits the design of PCC but proposes a new pattern detection algorithm. It defines an experiential mathematical expression to measure *recency* and classifies PCs according to the comparison between the average recency and a static threshold.

3 The Design of a Robust Adaptive Caching Replacement (RACE) Algorithm

This section presents the design of the RACE caching algorithm in detail. We first introduce the recently developed program-context-based technology in buffer caching and then analyze its limitations that in part motivate our RACE design, which is followed by the presentation of the details of our RACE algorithm.

3.1 PC-Based Technology in Caching Replacement

The temporal locality of I/O references in all kinds of program executions has been extensively demonstrated and is a well-known program behavior. Cache performance can be enhanced by taking full advantage of the temporal locality. Based on this principle, many cache management algorithms, including PCC [7] and AMP [8], exploit the temporal locality by using the history information of program behavior to estimate the reuse distance of cache blocks. These studies successfully link the past I/O behavior to their future reoccurrences by borrowing a computer architectural concept: *program counters*, which indicate the location of the instructions in memory. It is found that a particular instruction which is identified by its PC usually performs a very unique task and seldom changes its behavior. Thus, these studies assume that there is a considerably high probability of the access pattern of a PC remaining unchanged in the near future.

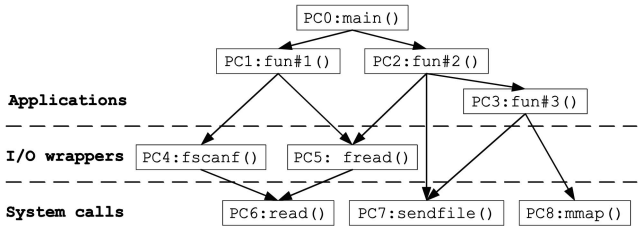


Fig. 1. An example call graph of some application.

Fig. 1 presents a call graph to further illustrate the key idea behind the studies of PCC and AMP. A call graph represents the runtime calling relationships among a program's functions or procedures in which a node corresponds to a function and an arc represents a call. An I/O instruction which is issued by some function in the application layer may be interpreted in the I/O wrapper layer that hides the I/O complexity, provides flexible interfaces, and eventually invokes system calls to access data. To uniquely identify the program context from which an I/O operation is invoked, a PC *signature* is defined as the sum of the PCs of all functions along the I/O call path. PC signatures can be obtained by traversing the function stack frames backward from the system calls *main()*. For simplicity, program signatures are denoted as PCs in the rest of this paper.

PCC and AMP separate the I/O references into substreams according to their PCs and then classify PCs into appropriate I/O reference pattern categories. A PC is assumed to exhibit the same I/O reference pattern in the future as that into which it has been classified and the target data blocks referenced by the current PC will be managed by the corresponding policy. Unfortunately, such an approach has three significant disadvantages:

1. The first iteration of each new PC in a global looping pattern will be misclassified as *sequential*. This is intrinsically caused by the inability of PC-based schemes to detect the phenomenon of *pattern sharing* among multiple PCs. Pattern sharing in real applications is not rare. For example, it is highly likely that a subroutine is called by multiple parent subroutines but shares the same I/O access pattern. Recursive functions are another example since they generate a set of different program signatures but share the same reference patterns. Multithreading can also lead to pattern sharing. Figs. 2 and 3 show the traces of *gnuplot* and *BLAST*, whose detailed descriptions are presented in Section 4. (Throughout this paper, the terms "block number" and "block address" are used interchangeably.) In the *gnuplot* trace, a sequence of plotting commands is issued in the order of *plot*, *plot*, *replot*, *plot3d*, *replot* to two large data files. Although the two *replot* functions access a data file with the same pattern as their preceding *plot* and *plot3d* functions, they follow a slightly different I/O path and thus have different PCs with each *plot*. Although these plotting functions access the data repeatedly, a pure PC-based scheme such as PCC or AMP will erroneously classify these references as *sequential*. In the trace of *BLAST* (also described in Section 4), as shown in Fig. 3, the program forks three threads and searches through the database files simultaneously. Although the database files are accessed repeatedly, PC-based detectors will not be able to classify the patterns correctly due to their inability to retain the "global picture."
2. Pattern conflicts reduce detection accuracy and increase management overhead. Fig. 4 presents traces of the top-six PCs with the highest numbers of references, collected from the *gcc* trace described in Section 4. Although PC1 and PC2 exhibit a looping pattern, PC5 and PC6 show a *sequential* pattern and PC3 and PC4 show a *sequential* pattern with a small degree of repetition. Most referenced blocks thus are classified as *sequential* if they are initiated by PC3, PC4, PC5, and PC6 and as looping if by PC1 and PC2. Since the references of these PCs are interwoven with one another, blocks need to be continuously moved between the *sequential* partition and the *looping* partition. Such moves require *locks* to ensure consistency and correctness and can cause a significant maintenance overhead.
3. PC-based schemes cannot accurately distinguish *locality strengths*. Locality strength in the PC-based approach is used for determining which block is replaced. It is measured by the looping period, where a longer looping period represents a weaker locality. PCC uses a single period to measure the locality of all the blocks accessed by a particular PC on a cache miss and evicts the block accessed by the

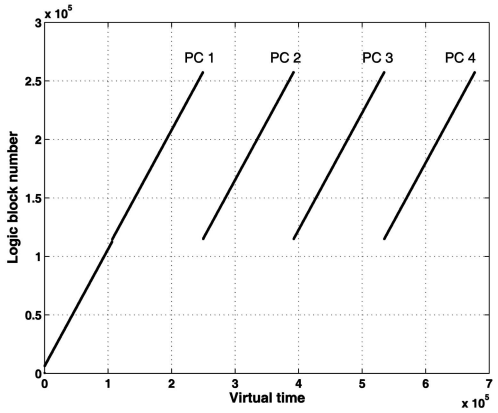


Fig. 2. Block references of *gnuplot*.

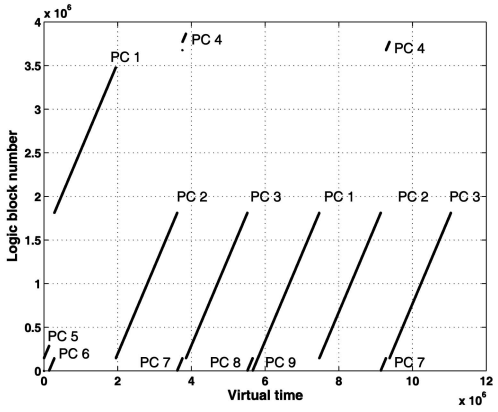


Fig. 3. Block references of *BLAST*.

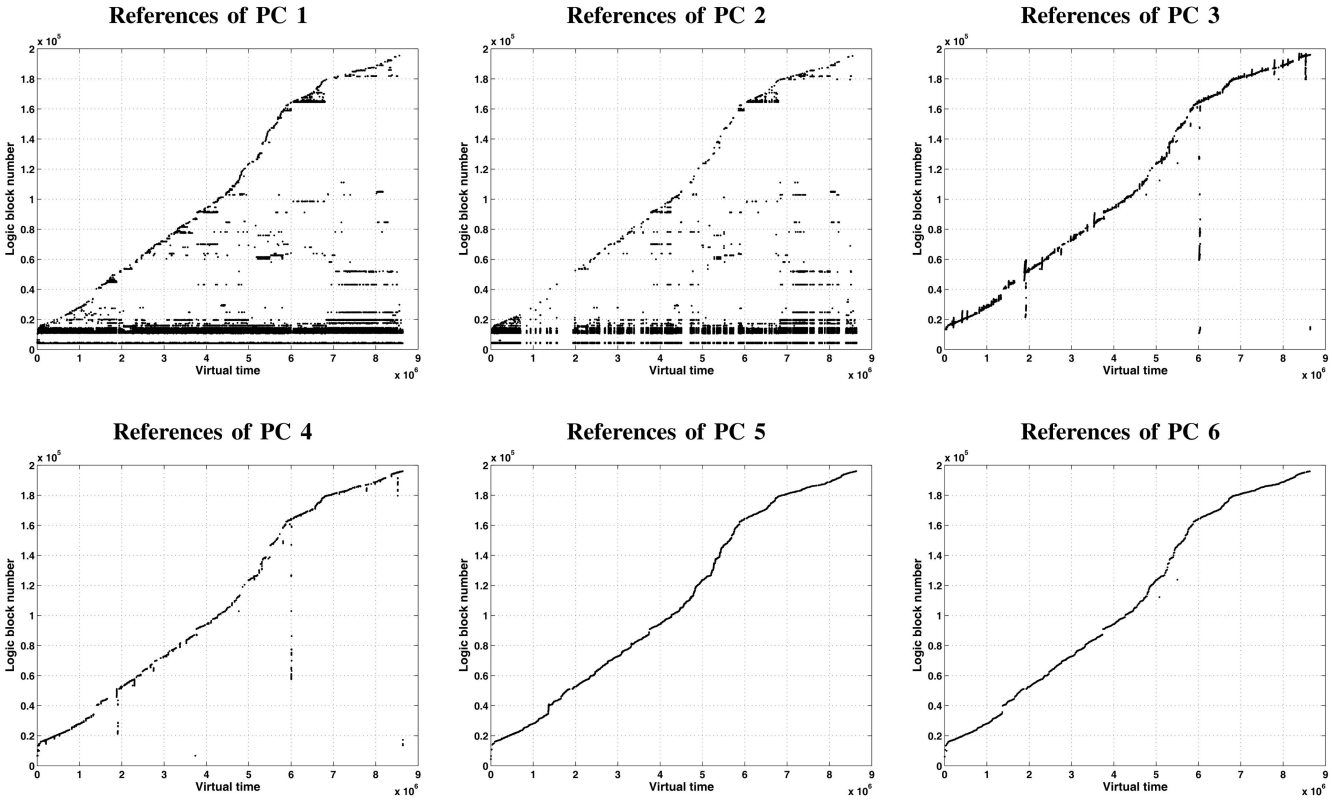


Fig. 4. Traces of six PCs with the highest numbers of references in *gcc*. The PCs, in order of decreasing number of references, are from 1 to 6.

PC that has the longest looping period. Although this period is averaged exponentially[1] by weighing recent periods more heavily than older ones, clearly a single looping period will not accurately measure the locality strength when a large amount of data is accessed. This can be easily observed from the traces of PC1 and PC2 in Fig. 4. Although PC1 and PC2 show a looping pattern, there is a significant portion of blocks whose actual looping periods are much longer than the average looping period.

3.2 The Design of RACE

Our RACE scheme is built upon the assumption that *future access patterns have a strong correlation with both the program context identified by program signatures and the past access behavior of currently requested data*. Although UBM only associates its prediction with the data's past access behavior, PCC and AMP only consider the relationship between future patterns and the program context in which the current I/O operation is generated. Our assumption is more appropriate for real workloads, as demonstrated by our comprehensive experimental study presented in Section 5.

Our RACE scheme automatically detects an access pattern as belonging to one of the following types:

- *Sequential references*. All blocks are referenced one after another and are never revisited again.
- *Looping references*. All blocks are referenced repeatedly with a regular interval.
- *Other references*. All references that are not sequential or looping.

Fig. 5 presents the overall structure of the RACE caching scheme. RACE uses two important data structures: a *file hash table* and a *PC hash table*. The *file hash table* records the sequences of consecutive block references and is updated for each block reference. The sequence is identified by the file description (*inode*), the starting and ending block numbers, the last access time of the first block, and their

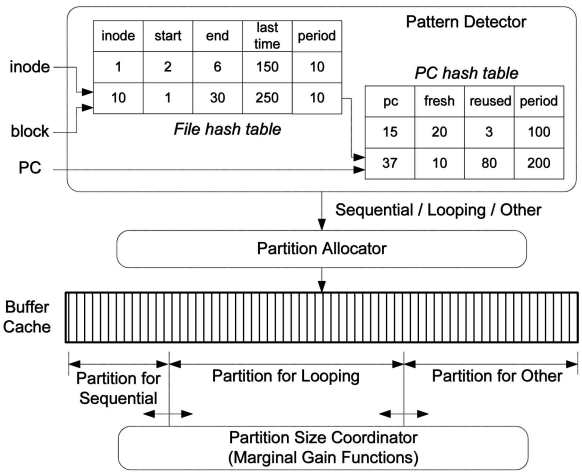


Fig. 5. The key structure of the RACE scheme. The Partition allocator and the Partition size coordinator take the results of pattern detector to adaptively fine-tune the size of each cache partition. If a sequence is found in the file hash table, then the period stored in the file hash table is used for updating the period field in the PC hash table.

1. The exponential average S of a time series $u(t)$ is defined as $S(t+1) = \alpha \cdot S(t) + (1-\alpha) \cdot u(t+1)$, where $0 < \alpha < 1$.

looping period. The *virtual access time* is defined on the reference sequence, where a reference represents a time unit. The looping period is exponentially averaged over the virtual time. The *PC hash table* records how many *unique* blocks each PC has accessed (*fresh*) and how many references (*reused*) each PC has issued to access blocks that have been visited previously. Although PCC also uses two counters, our RACE scheme is significantly different from PCC in that 1) PCC's counters do not accurately reflect the statistical status of each PC process, resulting in a misclassification of access patterns, as discussed later in this section, and 2) PCC only considers the correlations between the last PC and the current PC that accesses the same data block. In fact, many PCs exist in one application and it is likely that two or more PCs access the same data blocks.

The detailed pattern detection algorithm is given in Algorithm 1. The main process can be divided into three steps. First, the file hash table is updated for each block reference. RACE checks whether the accessed block is contained in any sequence in the file hash table. If found, RACE updates both the last access time and the sequence's average access period. When a block is not included in any sequence of the file hash table, RACE then tries to extend an existing sequence if the current block address is the next block of that sequence; otherwise, RACE assumes that the current request starts a new sequence. Second, RACE updates the PC hash table by changing the *fresh* and *reused* counters. For each revisited block, *fresh* and *reused* of the corresponding PC are decreased and increased, respectively. On the other hand, for a block that has not been visited recently, the *fresh* counter is incremented. The last step is to predict access patterns based on the searching results on the file and PC hash tables. If the file table reports that the currently requested block has been visited before, a "looping" pattern is returned. The looping period will identify how often this file has been accessed. If the file table cannot provide any history information of the current block, RACE relies on the PC hash table to make predictions. A PC with its *reused* counter larger than its *fresh* counter is considered to show a "looping" pattern. On the other hand, a PC is classified as "sequential" if the PC has referenced a certain amount of one-time-use-only blocks and as "others" if there is no strongly supportive evidence to make a prediction. By using the hashing data structure to index the file and PC tables, which is also used in LRU to facilitate the search of a block in the LRU stack, RACE can be implemented with a time complexity of $O(1)$.

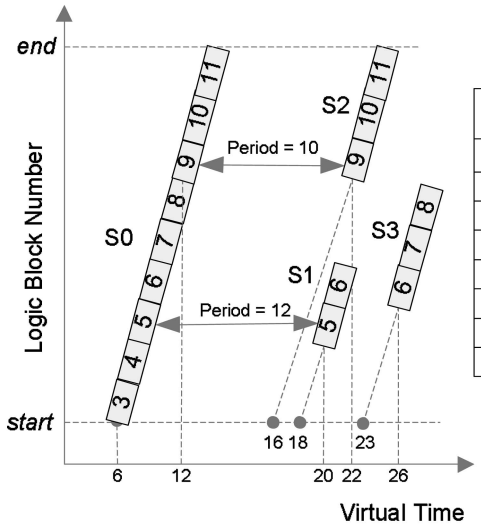
Algorithm 1: Pseudocode for the RACE pattern detection algorithm.

1: **RACE(**$inode$, $block$, pc, $curTime$)
2: {$\mathbb{F}$: File hash table; $\mathbb{P}$: PC hash table}
3: **if** $PC \notin \mathbb{P}$ **then** Insert (pc, 0, 0, ∞) into $\mathbb{P}$;
4: **if** $\exists f_1 \in \mathbb{F}$, $f_1.inode = inode$ and $f_1.start \leq block \leq f_1.end$ **then**
5: $lastTime = curTime - (block - f_1.start)$; {infer "ghost" reference time of the first block}
6: $f_1.period = \alpha \cdot f_1.period + (1 - \alpha) \cdot (lastTime - f_1.lastTime)$; {exponential average}
7: $\mathbb{P}[pc].reused$++; $\mathbb{P}[pc].fresh$–;
8: $\mathbb{P}[pc].period = \beta \cdot f_1.period + (1 - \beta) \cdot \mathbb{P}[pc].period$; {exponential average}
9: {update last reference time of the first block}
10: **if** access direction reversed **then** $f_1.lastTime = lastTime$;
11: **return**("looping", $f.period$);
12: **else if** $\exists f_2 \in \mathbb{F}$, $f_2.inode = inode$ and $f_2.end = block - 1$; **then**
13: $f_2.end = block$; {extend existing sequence}
14: $\mathbb{P}[pc].fresh$++;
15: **else**
16: $f.inode = inode$; $f.start = f.end = block$; $f.lastTime = curTime$; $f.period = \infty$;
17: Insert f into $\mathbb{F}$; {Insert a new sequence}
18: $\mathbb{P}[pc].fresh$++;
19: **end if**
20: **if** $\mathbb{P}[pc].reused \geq \mathbb{P}[pc].fresh$ **then return**("looping", $\mathbb{P}[pc].period$);
21: **if** $\mathbb{P}[pc].fresh >$ threshold **then return**("sequential");
22: **return**("other");

How to efficiently calculate the average access time for each access sequence is a challenging issue. It is not realistic to record the last access time of every accessed block due to prohibitively high overhead. We choose to only record the last access time of the very first block of a sequence to reduce the overhead. The real implementation is slightly different from the abstract procedure given in Algorithm 1 and Fig. 6 shows an example that illustrates the basic process. In a repeated I/O stream such as $S1$, $S2$, and $S3$, the access time of each block reference is virtually projected back to the time of the *start* block. Then, the current access period is the time difference between the projected time and the recorded last access time of the start block. The access period of a sequence is exponentially averaged over the access periods of all block references in that sequence. After the second reversing point (a decrement in access addresses) such as the time instant 16 in this example, the last access time recorded in the file hash table is then updated for all subsequent reversing points. Occasionally, the access periods may be erroneously calculated. For example, the access period of references to blocks 7 and 8 is incorrectly reported as 5. This approach, however, does not compromise the accuracy significantly, as indicated by our simulation results presented in Section 5.

The number of repeatedly access blocks, rather than the number of repeatedly accessed files, is used for identifying the loop patterns for each PC. This is because we want to give cold files, which are less frequently accessed, less weight in the classifying process. Previous studies have shown that a small fraction of files absorb most of the I/O activities in a file system [30], [31], [32], [33]. We believe that this commonly existing file access locality justifies our choice. Otherwise, biased pattern predictions would be generated if we place the same weight on all files and do not consider how many blocks have been accessed from individual files.

By observing the patterns both at the program context level and the file level and by exploiting the detection mechanism in both the continuous block address space within files and the discrete block address space in program contexts, RACE can more accurately detect access patterns.



Current Time	Current Blk Ref	Current Period	Last Access Time
20	5	18-6=12	6
21	6	18-6=12	6
22	9	16-6=10	6
23	10	16-6=10	6
24	11	16-6=10	6
26	6	23-18=5	18
27	7	23-18=5	18
28	8	23-18=5	18

Fig. 6. An example to illustrate the calculation of average access period.

An example, which is shown in Fig. 7, is used for illustrating and comparing the classification results of RACE, UBM, PCC, and AMP, in which all false classification results are underscored:

RACE. File A is initially classified as *other*. After file A is visited, the $fresh$ and $reused$ counters of PC1 are set to 2 and 0, respectively. After the first block of file B is accessed, the pattern of PC1 immediately changes to be *sequential* since the $fresh$ count becomes larger than the threshold. Thus, during the first iteration of accesses to files A and B, RACE incorrectly classifies the first three blocks as *other* and then the next three blocks as *sequential*. However, after the first iteration, RACE can correctly identify the access patterns. During the second and third iterations, the sequences for both files A and B are observed in the file hash table and are correctly classified as *looping*. Although file C is visited for the first time, it is still correctly classified as *looping*. This is because the $fresh$ and $reused$ counters of PC1 are 0 and 6, respectively, before file C is accessed. After that, all references are made by PC2 and they are classified as *looping* since the file hash table have access records of files B and C.

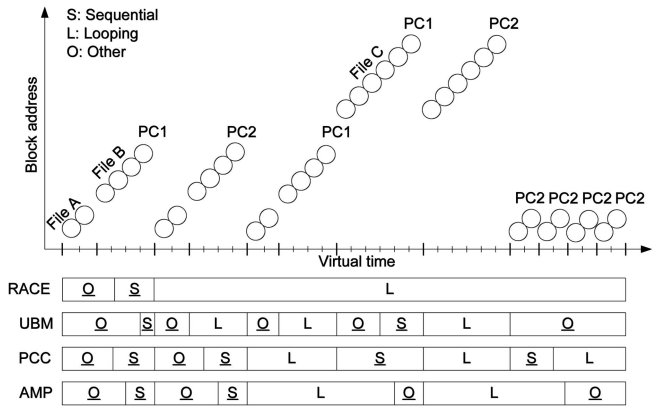


Fig. 7. An example of reference patterns. The sequentiality thresholds for UBM, PCC, and RACE are 3. The sequentiality threshold, looping threshold, and exponential average parameter for AMP are 0.4, 0.01, and 4, respectively. All incorrect classification results are underscored.

UBM. Since the total number of blocks in file A is less than the threshold in UBM, all references to file A are incorrectly classified as *other*. The initial references to the first three blocks and the fourth block of file B are detected as *other* and *sequential*, respectively. After that, all references to file B are classified as *looping*. Similar classification results are observed for references to file C.

PCC. Although the blocks of a sequential access detected by UBM have to be contiguous within a file, PCC considers sequential references as a set of distinct blocks that may belong to different files. The initial three blocks accessed by PC1 are classified as *other* and, then, PC1 is classified as *sequential*. Although PC2 is accessing the same set of blocks as PC1, it is still classified first as *other* and then as *sequential* when the threshold is reached. Before file C is accessed, the values of both *seq* and *loop* of PC1 are 6. Since *seq* of PC1 is increased and becomes larger than *loop*, accesses to file C made by PC1 are classified as *sequential*. Before file C is revisited by PC2, the values of both *seq* and *loop* of PC2 have changed to be 0 and 6, respectively, through the references made by PC1. Thus, references to file C are detected as *looping*. After file C is accessed, the values of both *seq* and *loop* of PC2 are 6. References to file A made by PC2 are classified first as *sequential* and then as *looping*.

AMP. The classification results are reported by the AMP simulator from its original author. To reduce the computation overhead, AMP uses a sampling method, with some sacrifice to the detection accuracy. Since the sample trace used here is not large, the entire results are collected without using the sampling function in the AMP simulator. The initial *recency* of a PC, defined as the average ratios between the LRU stack positions and the stack length for all blocks accessed by the current PC, is set to be 0.4. Last references to file A made by PC2 are incorrectly detected as *other*, which indicates that AMP has a tendency to classify looping references as *other* in

TABLE 1
Traces Used and Their Statistics

Trace	Request Num.	Data Size (MB)	File Num.	PC Num.
gcc	8765174	89.4	19875	69
gnuplot	677442	121.8	8	26
cscope	2131194	240	16613	40
glimpse	2810992	194	16526	7
BLAST	11042696	1789.6	13	20
tpch	13468995	1187	49	150850
tpcr	9415527	1087	49	150200

Trace	Concurrently Executed Applications	Request Num.	Data Size (MB)	File Num.	PC Num.
multi1	*glimpse + cscope*	4942186	434	33139	47
multi2	*gnuplot + BLAST*	11720138	1911	21	46
multi3	*cscope + BLAST + gcc*	21939064	2119	36501	129

the long term. We can use a shorter and simpler reference stream to further explain it. Given a looping reference stream $L = \{1, 2, 3, 4, 3, 4, 3, 4\}$, the average recency of L is 0.67, which is higher than the threshold, 0.4. Accordingly, AMP falsely considers the pattern of L as *other*. In addition, AMP has another anomaly in which it has a tendency to erroneously classify a *sequential* stream as a *looping* one. For example, for a sequential reference stream $S = \{1, 2, 1, 2, 3, 4, 5, 6, 7, 8\}$, the average recency of S is 0 and AMP identifies this sequential pattern as *looping*. The first anomaly is more commonly observed in the workload studies in this paper, which explains why the performance of AMP tends to be close to that of ARC in our experiments shown in Section 5.

4 Application Traces Used in the Simulation Study

The traces used in this paper are obtained by using a trace collection tool provided in [7]. This tool is built upon the Linux *strace* utility that intercepts and records all system calls and signals of traced applications. A PC signature is obtained by tracing backward the function call stack in *strace*. The modified *strace* investigates all I/O-related activities and reports the I/O triggering PC, file identifier (*inode*), I/O starting address, and request size (in bytes).

We use trace-driven simulations with various types of workloads to evaluate the RACE algorithm and compare it with other algorithms. These traces are considered typical and representative of applications in that most of them are routinely used in other caching algorithm studies. For example, the *cscope*, *glimpse*, and *gcc* traces are used in [5], [7], [17], [34], *gnuplot* in [6], and *tpch* and *tpcr* in [35]. Table 1 summarizes the characteristics of these traces and a more detailed description of each trace is presented below. The file and PC numbers represent the total numbers of unique files and PC signatures, respectively:

1. **gcc** is a GNU C compiler trace and it compiles and builds Linux kernel 2.6.10.
2. **cscope** [36] is an interactive utility that allows users to view and edit parts of the source code relevant to specified program items under the auxiliary of an index database. In *cscope*, an index database needs to be built first by scanning all examined source code. In our experiments, only the I/O operations during the searching phases are collected. The total size of the source code is 240 Mbytes and the index database is around 16 Mbytes.
3. **glimpse** [37] is a text information retrieval tool, searching for keywords through large collections of text documents. It builds approximate indices for words and searches relatively fast with small index files. Similarly to cscope, the I/O activities during the phase of index generation are not included in our collected trace. The total size of text is around 194 Mbytes and the glimpse index file is about 10 Mbytes.
4. **gnuplot** is a command-line-driven interactive plotting program. Five figures are plotted by using four different plot functions that read data from two raw data files with sizes of 52 and 70 Mbytes, respectively.
5. **BLAST** [38] is a widely used scientific application in computational biology. It is designed to find regions of local similarity between a query sequence and all sequences in a large gene database. In this study, a large database named *human EST* is used and it is roughly 1.8 Gbytes in size. Previous research has shown that the length of 90 percent of the query sequences used by biologists is within the range of 300-600 characters [39]. Thus, in this work, we choose to use a sequence of 568 characters extracted from the *ecoli.nt* database as the query sequence.
6. **tpch** and **tpcr** benchmarks [40] perform random access to a few large MySQL database files. The traces used in this study are obtained from [35]. Butt et al. [35] suggest that disk I/O prefetching should be disabled in both tpch and tpcr to prevent cache pollution. Thus, we only use the traces without prefetching.
7. **multi1** is obtained by executing *glimpse* and *cscope* concurrently, which represents a text-searching environment.
8. **multi2** is obtained by executing *gnuplot* and *BLAST* concurrently, which simulates an environment of database queries and scientific visualization.

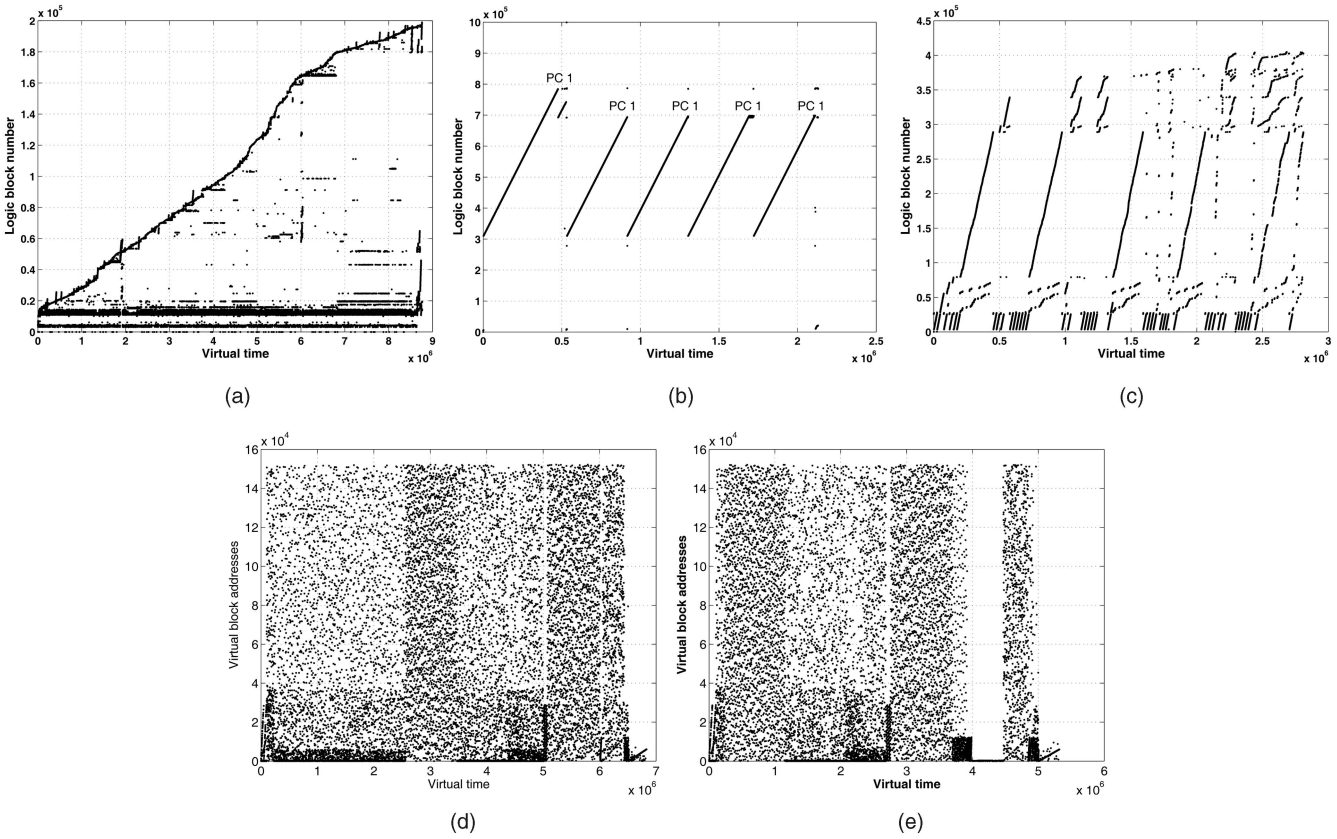


Fig. 8. Block references of (a) *gcc*, (b) *cscope*, (c) *glimpse*, (d) *tpch*, and (e) *tpcr*.

9. **multi3** is obtained by concurrently executing three workloads: *cscope*, *BLAST*, and *gcc*, which provides a workload of database queries and code programming.

The traces of *gnuplot*, *BLAST*, *gcc cscope*, *glimpse*, *tpch*, and *tpcr* in Figs. 2, 3, and 8, respectively, show trace address as a function of the virtual time that is defined as the number of references issued so far and is incremented for each request. For the sake of visibility, the tpch and tpcr traces are shown with a sampling period of 400.

5 Performance Evaluation

This section presents the performance evaluation of RACE through a trace-driven simulation study with 10 different but typical traces from real applications. We compare the performance of RACE with seven other replacement algorithms, including UBM [5], [9], PCC [7], AMP [8], LIRS [17], [20], ARC [18], LRU, and OPT. Simulation results of UBM, LIRS, and AMP were obtained using simulators from their original authors, respectively. We implemented the ARC algorithm according to the detailed pseudocode provided in [22]. We also implemented the PCC simulator and our RACE simulator by modifying the UBM simulator code. UBM's cache management scheme based on the notion of marginal gain is used in PCC and RACE without any modification, which allows an effective and fair comparison of the pattern detection accuracies of UBM, PCC, and RACE.

The measure of *hit ratio* is used as our primary metric in the performance comparison. Hit ratio is defined as the fraction of I/O requests that are successfully served by the cache without going off to the secondary disk storage. We believe that hit ratio is a comprehensive metric for evaluating the accuracy, responsiveness, and stability of pattern-detection-based algorithms since these three factors can directly impact the hit ratios. More specifically, the accuracy of locality detection, in terms of access periods in this paper, directly influences the order of block eviction. Promptly adapting to patterns changes can avoid hit ratio degradation caused by obsolete information. Stability will guarantee consistently high hit ratios across different cache sizes and a wide spectrum of workloads.

5.1 Cache Management Scheme

PCC and RACE use the *marginal gain function* in the original UBM [5], [9] simulator to manage the three partitions of the buffer cache. Marginal gain is defined as the expected extra hit ratios increased by adding one additional buffer [25], [41]. The marginal gain of the *sequential* partition is zero since no benefit can be obtained from caching one-time-use-only data. The marginal gain for the looping partition is $\frac{1}{p_{max}+1}$, where p_{max} is the maximum looping period of blocks in the looping partition. The marginal gain for the *other* partition is estimated according to Belay's life function [42]. UBM, PCC, and RACE aim at maximizing the expected hit ratios by dynamically allocating the cache space to the three partitions: *looping*, *sequential*, and *other*. For the *sequential* partition, not more than one buffer is allocated, except when the buffers are not fully utilized, since its marginal gain is zero. The cache space is switched between the *looping* and

TABLE 2
Parameters for Cache Replacement Policies

Policy	Parameters
UBM	*max loop or sequential lists = 2000*, *threshold for detecting sequential access = 10*, *exponential average* $\alpha = 0.5$
PCC	*exponential average* α *= 0.5*, *no sampling*, *threshold for detecting sequential access = 100*
AMP	*threshold for detecting looping access = 0.4*, *exponential average* α *= 0.1*, *hit ratio threshold for detecting sequential = 0.001*
LIRS	*HIR* = 1% *of cache, LRU stact = 2* $\times$ *cache size*
ARC	*ghost cache = cache size*
RACE	*size of the file hash table = 2000*, *size of the PC hash table = 200*, $\alpha = 0.5$, $\beta = 0.1$ and *threshold for detecting sequential access = 100*

the *other* partitions according to the comparison of their estimated marginal gains. It always frees a buffer in the partition with a smaller marginal gain and allocates it to the *other* partition until both marginal gains converge to the same value.

AMP proposes a randomized eviction policy to manage the cache partitions. Upon a cache miss, AMP randomly chooses a nonempty partition and frees the block at the MRU position of that partition. Although this randomized eviction works well for their design, which employs ARC [18] to manage cache replacement, this replacement algorithm does not work efficiently for UBM, PCC, and RACE for the simple reason that UBM, PCC, and RACE only use LRU or MRU to manage cache in order to achieve a low overhead. Although ARC itself can automatically adapt to the workload changes, LRU and MRU do not provide such adaptability.

5.2 Simulation Results

Based on access patterns, the 10 traces used in the simulation study are divided into two main groups. Traces *gnuplot*, *BLAST*, and *cscope* fit in the group in which looping patterns dominate. Traces *gcc*, *glimpse*, *tpch*, *tpcr*, *multi1*, *multi2*, and *multi3* are in the group with mixed patterns. In what follows, we report our simulation results in both groups and compare RACE with UBM, PCC, AMP, LIRS, ARC, LRU, and OPT. The simulation parameters for these algorithms are given in Table 2 and all of them are suggested by their original authors in the literature, except that the exponential average parameter α is not given in PCC [7].

5.2.1 Performance under Workloads with Looping Patterns

- ***gnuplot.*** Fig. 9a shows the hit ratio comparisons for the workload *gnuplot* that has a looping pattern with long intervals. This workload generates a pathological case for LRU when the size of accessed blocks in the loop is larger than the cache. Accordingly, LRU performs poorly and has the lowest hit ratios. A similar behavior is present in ARC as all blocks are accessed more than once and the frequency list is consequently managed by LRU. ARC achieves almost optimal hit rates when the cache is large since it can successfully evict the least frequently used blocks. The benefit of LRU and ARC caching is only observed when the entire looping file set fits in the cache. Since the long sequential accesses are made by four PCs, respectively, as presented in Fig. 2, PCC incorrectly classifies all references as *sequential*, as expected, and results in very low hit ratios. Both UBM and RACE, on the other hand, can correctly classify the references as *looping* after the first long sequence of sequential accesses, achieving much higher hit ratios. However, the cache management scheme employed in AMP is not efficient, thus resulting in lower performance than UBM and RACE. In summary, RACE achieves the same performance as UBM and a maximum of 52.3 percent and 39.9 percent improvement in hit ratio over LRU and AMP, respectively.
- ***BLAST.*** The performance comparisons under the *BLAST* workload are presented in Fig. 9b. The accesses are dominated by three major PCs initiated by three concurrently running threads in the BLAST application. PCC cannot detect the access pattern sharing among these three PCs and marks many blocks as *sequential.* This is the main reason that the hit ratios of PCC are 20.8 percent, 19.8 percent, and 5.0 percent lower than those of RACE, UBM, and AMP, respectively, on average. Since there are only 13 files accessed in this trace, the misclassification of the first iteration of accesses to new files does not significantly degrade the UBM performance under this workload. Thus, UBM achieves comparable hit ratios with RACE. Although RACE improves LRU by as much as 56.9 percent, for an average of 30.1 percent, ARC only improves LRU by 15.1 percent at the maximum, with an average of 2.3 percent. ARC is inherently capable of recording a reference history that is only twice the cache size. Under a workload with a large working set such as *BLAST*, ARC fails to detect the looping patterns due to the lack of history information. Similarly, LIRS also suffers from limited history information that is stored in its two LRU stacks.
- ***cscope.*** Fig. 9c shows the hit ratio comparison for the *cscope* application. As explained in Section 3, AMP

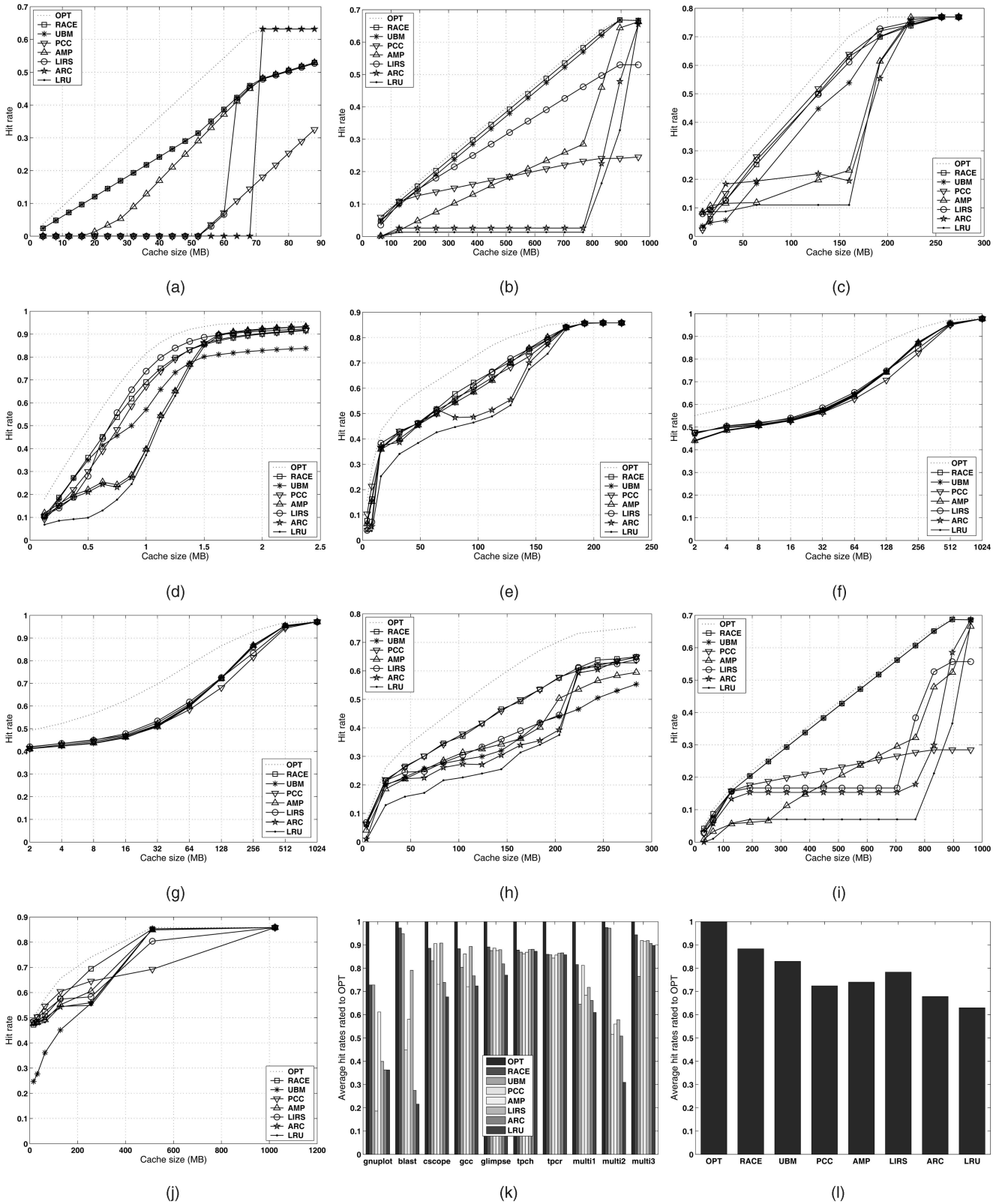


Fig. 9. Comparison of hit ratios. (a) *gnuplot*. (b) *BLAST*. (c) *cscope*. (d) *gcc*. (e) *glimpse*. (f) *tpch*. (g) *tpcr*. (h) *multi1*. (i) *multi2*. (j) *multi3*. (k) Average hit ratios (normalized to the average hit ratio of OPT). (l) Average hit ratios for all traces (normalized to the average hit ratio of OPT).

tends to classify the looping references as *other* due to the fact that the average recency in AMP is highly sensitive to stale history information as the center of working set shifts. As a result, the performance of AMP is close to that of ARC, which is used in AMP to manage the cache partition for the *other* pattern. PCC, LIRS, and RACE achieve almost the same hit ratios and their hit ratios are 10.0 percent higher than

that of UBM. Among the pattern-detection-based algorithms, two main factors contribute to the inferior performance of UBM to RACE and PCC. First, with a total of 16,613 files accessed in *cscope*, there are around 13.6 percent of files whose sizes are smaller than the threshold used by UBM. These small files form implicit looping patterns in a nonconsecutive manner and thus are ignored in UBM. On the contrary, RACE and PCC can detect the implicit looping patterns in which a group of small files are repeatedly accessed. Second, there are around 5 percent of references that are issued to access files for the first time and UBM cannot make a correct prediction for these references since UBM is intrinsically incapable of making an accurate prediction when a file has not been accessed previously.

5.2.2 Performance under Workloads with Mixed Patterns

- ***gcc.*** Fig. 9d shows the hit ratios under the workload of *gcc* that builds the newest version of the Linux kernel at the time of our experiments. Gniady et al. [7] use the trace collected only during the preprocessing of an old Linux kernel (2.4.20). The trace does not reflect the whole I/O characteristics of building the kernel and it has only around 80,000 read operations. We choose to use the trace that is collected during the enitre building process of the newest Linux kernel 2.6.10 and it contains more than 9×10^6 read operations. We believe that this gives us a more comprehensive evaluation. In this trace, around 68 percent of the references are targeted at small files that are shorter than the threshold. When the cache size is smaller than 0.7 Mbyte, UBM and RACE perform the best. However, the hit rates of LIRS are the highest when the cache size is between 0.7 Mbyte and 1.5 Mbytes, although those of RACE come extremely close. Since the working set of *gcc* is not large, LIRS can better differentiate the locality strength of referenced blocks and is 2.4 percent and 0.7 percent better than PCC and RACE, respectively. AMP cannot adapt to the shift of working set centers and falsely classifies almost all references as *others*, which explains why AMP and ARC share similarly poor performance.
- ***glimpse.*** Fig. 9e shows the hits ratio comparisons under the *glimpse* workload. The performances of RACE, UBM, AMP, PCC, and LIRS are very close to one another and perform much better than LRU. Specifically, RACE improves the hit ratios of LRU by as much as 17.5 percent, with an average of 8.1 percent. Surprisingly, ARC clearly shows the Belady behavior, where the hit ratio decreases while the cache size increases. This anomaly can be observed in the previous workloads as well. As introduced in Section 2, ARC divides the cache of size c into two LRU lists, L_1 and L_2, and they retain a total of c physical blocks and c identifiers of recently evicted blocks. Although blocks in L_1 have been used only once, blocks in L_2 have been used twice or more. A hit in L_1 promotes the referenced block to L_2 so that it can stay in the cache for a longer time. ARC bases its replacement strategy on the following assumption: If the requested block identifier is in L_1 on a cache miss, then it is likely that the number of physical blocks in L_1 is too small. Similarly, if the identifier of a missed block is in L_2, then the number of physical blocks in L_2 is conjectured to be too small. Thus, ARC adaptively allocates more cache space to a list that has more misses. It achieves this goal by dynamically changing the number of physical blocks allocated to L_1 with a variable step size. Under the same I/O workload, the step size is continuously updated by its exponential average and is influenced by the size of the cache. Thus, the step size cannot truly distinguish the "cold" blocks from the "hot" ones and leads to the severe Belady anomaly.
- ***tpch and tpcr.*** Although the tpch and tpcr benchmarks repeatedly access a total of six large database files, only 3 percent of the references occur to immediately consecutive blocks [35] and over 80 percent of stride distances between consecutive references are larger than 10 blocks. Figs. 9f and 9g compare the hit ratio performance. PCC performs slightly worse than LRU when the cache size is larger than 64 Mbytes. Although the majority of references are absorbed by the six databases, there are over 15,000,00 program signatures and the misprediction of access periods degrades the performance in PCC. Compared with PCC, RACE not only correctly identifies more periodical accesses but also provides more accurate access periods.
- ***multi1, multi2, and multi3.*** The hit ratio comparisons under the workloads of *multi1*, *multi2*, and *multi3* are presented in Figs. 9h, 9i, and 9j, respectively. In summary, in *multi1*, RACE and PCC achieve the best hit ratios. In *multi2*, the hit ratios of RACE and UBM are the highest. RACE outperforms all other algorithms in *multi3*.

5.3 Average Hit Ratio Comparisons

Fig. 9k shows the average hit ratios normalized to the average hit ratio of the OPT replacement algorithm for the eight workloads studied in this paper. Fig. 10 compares the classification results between RACE, UBM, PCC, and AMP, which helps explain the reasons behind the superiority of RACE in terms of hit ratios. For *gnuplot*, *BLAST*, and *multi2*, the PCC algorithm is pathological in that PCC cannot distinguish the pattern sharing among different PCs and falsely classifies many looping patterns as *sequential* patterns, as shown in Fig. 10. For *cscope*, *gcc*, *glimpse*, *multi1*, and *multi3*, the UBM algorithm is pathological since it ignores the patterns clearly exhibited in small files and it is incapable of correctly detecting the access patterns for files that have not been referenced before. In almost all workloads, AMP erroneously identifies a larger fraction of accesses as the *other* pattern by as much as 74.9 percent, 19.7 percent, and 74.9 percent, on average, more than RACE, UBM, and PCC respectively, which dramatically lowers its hit ratios. RACE, on the contrary, exploits the

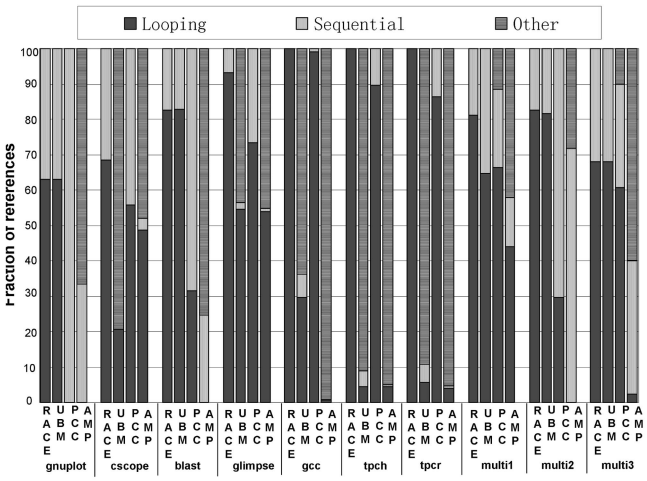


Fig. 10. Comparisons of classification results.

detection mechanism in both the continuous block address space within files and the discrete block address space in program contexts, resulting in an average of 52.4 percent, 24.5 percent, and 70.8 percent more looping patterns being correctly detected than UBM, PCC, and AMP, respectively.

The experimental results on the 10 workloads show that RACE is the most robust among all of the algorithms. Fig. 9l presents the average values of hit ratios (normalized to the optimal hit ratios) presented in Fig. 9k. Compared with UBM, PCC, AMP, LIRS, ARC, and LRU, the normalized hit ratio of RACE is higher by an average of 5.4 percent, 15.9 percent, 14.3 percent, 10.0 percent, 20.5 percent, and 25.4 percent, respectively. Table 3 shows the arithmetic average of absolute hit ratios of these algorithms, with different sizes in each workload. RACE can successfully overcome the drawbacks of LRU and improve its absolute hit ratios by as much as 56.9 percent, with an average of 15.5 percent.

Compared with other state-of-the-art pattern-detection-based schemes, RACE outperforms UBM, PCC, and AMP by as much as 22.5 percent, 42.7 percent, and 39.9 percent, with an average of 3.3 percent, 6.6 percent, and 6.9 percent, respectively. In the *cscope* trace, RACE is 1.0 percent inferior to PCC, on average, due to the fact that, although RACE correctly classifies files accessed at the end of the first iteration as *looping*, these files are only accessed twice, as shown in Fig. 8b, and RACE wastes partial memory by caching them. Compared with the state-of-the-art recency/frequency-based schemes, RACE consistently beats ARC in all workloads and outperforms LIRS in most workloads, except *cscope* and *gcc*. In the *cscope* and *gcc* traces, RACE is, on average, 1.1 percent and 0.7 percent inferior to LIRS in the absolute hit ratio. Since RACE improves the hit ratios of LIRS by an average of 6.0 percent over the eight workloads, we conclude that such a slight performance degradation in *cscope* and *gcc* is not severe. The *gcc* workload is extremely LRU friendly, in which an 89.4 Mbyte data is accessed and an LRU cache with a size of 1.5 Mbytes can achieve a hit ratio of 86 percent. It is our future work to avoid such slight performance degradation by improving our detection algorithm or by incorporating LIRS into RACE to manage the cache partitions. In summary, RACE relatively improves the hit ratios of UBM, PCC, AMP, LIRS, ARC, and LRU by 6.8 percent, 14.6 percent, 15.2 percent, 8.7 percent, 21.7 percent, and 29.3 percent on average. This superiority indicates that our RACE scheme is more robust and adaptive than any of the other six caching schemes and also proves our assumption that future access patterns are highly correlated with both program contexts and requested data.

5.4 Sensitivity Study on the Sampling Frequency

Our RACE algorithm needs to update both the file hash table and the PC hash table. The cache sizes are 1 Mbyte for *Linux*, 500 Mbytes for *mult1*, *mult2*, and *mult3*, and 50 Mbytes for the others. Fig. 11 shows the sensitivity of the updating frequency. With a sampling frequency of less than 16 blocks, the hit ratios are barely adversely affected for most benchmarks, except for *gcc* and *cscope.* In *gnuplot*, the performance of RACE reduces to LRU when the sampling frequency is large. From this analysis, we believe that the updating overhead and the memory requirement of the file hash tables and the PC tables can be traded off through appropriate sampling. A sampling period of 16 blocks provides a good trade-off for the studied benchmarks.

TABLE 3
Hit Ratios under the 10 Traces, Averaged over Different Cache Sizes

	OPT	RACE	UBM	PCC	AMP	LIRS	ARC	LRU
gnuplot	0.3961	**0.2884**	**0.2884**	0.0738	0.2423	0.1582	0.1437	0.1436
BLAST	0.3982	**0.3872**	0.3773	0.1789	0.2311	0.3149	0.1094	0.0860
cscope	0.5167	0.4577	0.4295	0.4681	0.3776	**0.4691**	0.3818	0.3496
gcc	0.7587	0.6704	0.6100	0.6536	0.5468	**0.6781**	0.5820	0.5492
glimpse	0.6681	**0.5954**	0.5849	0.5923	0.5849	0.5874	0.5469	0.5144
tpch	0.7724	0.6781	0.6711	0.6659	0.6709	0.6796	**0.6807**	0.6739
tpcr	0.7419	0.6378	0.6363	0.6257	0.6354	0.6413	**0.6423**	0.6364
multi1	0.5381	**0.4389**	0.3471	0.4371	0.3676	0.3863	0.3558	0.3280
multi2	0.4081	**0.3978**	0.3967	0.2104	0.2285	0.2363	0.2075	0.1263
multi3	0.6734	**0.6353**	0.5148	0.6190	0.6161	0.6187	0.6103	0.6047
overall	0.5872	**0.5187**	0.4856	0.4525	0.4501	0.4770	0.4261	0.4012

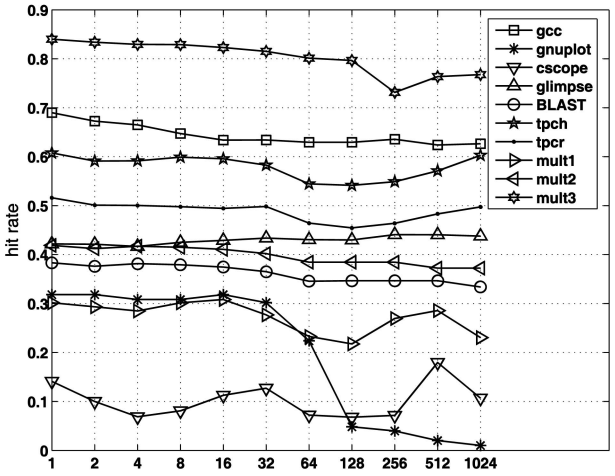


Fig. 11. Impacts of sampling frequency on hit ratios.

6 Conclusions

Cache replacement algorithms are crucial in bridging the increasing performance gap between processors and disk drives. Motivated by the limitations of existing state-of-the-art cache replacement algorithms, we propose a novel and simple block replacement algorithm called RACE. We make three main contributions: 1) We collected the I/O traces for eight real applications and investigated I/O access patterns in two correlated spaces (the program context space from which I/O operations are issued and the file space to which I/O requests are addressed), 2) our comprehensive application trace study revealed some pathological behaviors in existing state-of-the-art cache replacement algorithms, including a file-level detection method (UBM) and two program-context-level detection methods (PCC and AMP), and 3) an extensive simulation study conducted under these real-application workloads demonstrated that RACE, through its exploitation of the detection mechanism in both the continuous block address space within files and the discrete block address space in program contexts, is able to accurately detect reference patterns from both the file level and the program context level and thus significantly outperforms other state-of-the-art recency/frequency-based algorithms and pattern-detection-based algorithms. Due to the very high buffer cache miss penalties, which are typically six orders of magnitude higher than buffer cache hit times, we believe that the significant gains in hit ratios obtained by RACE over other algorithms will likely have significant performance implications in application response times.

Our study has two limitations. First, we have not implemented our design and evaluated it in real systems. Compared with recency/frequency-based algorithms such as LRU, LIRS, and ARC, the program-context-based algorithms, including RACE, PCC, and AMP, need to pay the extra overhead of obtaining PC signatures. Gniady et al. [43] report that it is inefficient to obtain program signatures through stack traversals in their quick-hack implementation. Use of a library modification approach which can read the PC directly from the calling program's stack and, hence, requires the least amount of overhead is suggested. Second, in order to achieve a direct comparison of pattern detection accuracy, RACE, as well as PCC, uses the marginal gain functions proposed in the UBM scheme to dynamically allocate the buffer cache. We believe that a more effective allocation scheme will be helpful to further improve the hit ratios. In the future, we will implement RACE into Linux systems and investigate other efficient allocation schemes.

Acknowledgments

The authors are grateful to Dr. Song Jiang and Dr. Xiaodong Zhang for providing the LIRS cache simulator, Feng Zhou et al. for their AMP simulator, and Chris Gniady et al. for their trace collection tool and *tpch* and *tpcr* traces. The authors would like to thank the anonymous reviewers for their efforts in improving this paper. This work is supported by a University of Maine (UMaine) Startup Grant, US National Science Foundation (NSF) Grant CCF-0621493, NSF Grant CCF-0621526, NSF CNS #0723093, NSF DRL #0737583, and Chinese NSF 973 Project Grant 2004cb318201.

References

[1] M.J. Bach, *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[2] A.S. Tanenbaum and A.S. Woodhull, *Operating Systems Design and Implementation.* Prentice Hall, 1987.

[3] R.W. Carr and J.L. Hennessy, "WSCLOCK—A Simple and Effective Algorithm for Virtual Memory Management," *Proc. Eighth ACM Symp. Operating Systems Principles (SOSP '81),* pp. 87-95, 1981.

[4] A.J. Smith, "Analysis of the Optimal, Look-Ahead Demand Paging Algorithms," *SIAM J. Computing,* vol. 5, no. 4, pp. 743-757, Dec. 1976.

[5] J.M. Kim, J. Choi, J. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, "A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," *Proc. Fourth Symp. Operating System Design and Implementation (OSDI '00),* pp. 119-134, Oct. 2000.

[6] J. Choi, S.H. Noh, S.L. Min, and Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme," *Proc. 1999 Usenix Ann. Technical Conf.,* pp. 239-252, June 1999.

[7] C. Gniady, A.R. Butt, and Y.C. Hu, "Program-Counter-Based Pattern Classification in Buffer Caching," *Proc. Sixth Symp. Operating System Design and Implementation (OSDI '04),* pp. 395-408, Dec. 2004.

[8] F. Zhou, R. von Behren, and E. Brewer, "AMP: Program Context Specific Buffer Caching," *Proc. Usenix Technical Conf.,* Apr. 2005.

[9] J. Choi, S.H. Noh, S.L. Min, E.-Y. Ha, and Y. Cho, "Design, Implementation, and Performance Evaluation of a Detection-Based Adaptive Block Replacement Scheme," *IEEE Trans. Computers,* vol. 51, no. 7, pp. 793-800, July 2002.

[10] L.A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems J.,* vol. 5, no. 2, pp. 78-101, 1966.

[11] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, no. 2, pp. 78-117, 1970.

[12] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 297-306, 1993.

[13] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th Int'l Conf. Very Large Data Bases (VLDB '94),* pp. 439-450, 1994.

[14] 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," *Proc. 1999 ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 134-143, 1999.

[15] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 122-133, 1999.

[16] Y. Zhou, J. Philbin, and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," *Proc. General Track: 2002 Usenix Ann. Technical Conf.,* pp. 91-104, 2001.
[17] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. ACM SIGMATRICS Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 31-42, June 2002.
[18] N. Megiddo and D.S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proc. Second Usenix Conf. File and Storage Technologies (FAST '03),* pp. 115-130, Mar. 2003.
[19] D. Lee, J. Choi, J.-H. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Trans. Computers,* vol. 50, no. 12, pp. 1352-1361, Dec. 2001.
[20] J. Song and Z. Xiaodong, "Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance," *IEEE Trans. Computers,* vol. 54, no. 8, pp. 939-952, Aug. 2005.
[21] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," *Proc. 2005 Usenix Ann. Technical Conf.,* Apr. 2005.
[22] N. Megiddo and D.S. Modha, "One Up on LRU," *;login:—The Magazine of the Usenix Assoc.,* vol. 4, no. 18, pp. 7-11, 2003.
[23] N. Megiddo and D.S. Modha, "One Up on LRU," *The Magazine of the Usenix Assoc.,* vol. 4, no. 18, pp. 7-11, 2003.
[24] P. Cao, E.W. Felten, A.R. Karlin, and K. Li, "Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling," *ACM Trans. Computer Systems,* vol. 14, no. 4, pp. 311-343, 1996.
[25] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc. 15th ACM Symp. Operating Systems Principles (SOSP '95),* pp. 79-95, 1995.
[26] A.D. Brown, T.C. Mowry, and O. Krieger, "Compiler-Based I/O Prefetching for Out-of-Core Applications," *ACM Trans. Computer Systems,* vol. 19, no. 2, pp. 111-170, 2001.
[27] T.M. Madhyastha and D.A. Reed, "Learning to Classify Parallel Input/Output Access Patterns," *IEEE Trans. Parallel and Distributed Systems,* vol. 13, no. 8, pp. 802-813, Aug. 2002.
[28] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 115-126, 1997.
[29] K. So and R.N. Rechtschaffen, "Cache Operations by MRU Change," *IEEE Trans. Computers,* vol. 37, no. 6, pp. 700-709, June 1988.
[30] R. Floyd, "Short-Term File Reference Patterns in a UNIX Environment," Technical Report TR-177, Computer Science Dept., Univ. of Rochester, Mar. 1986.
[31] C. Staelin, "High-Performance File System Design," PhD dissertation, Dept. of Computer Science, Princeton Univ., Oct. 1991.
[32] V. Cate and T. Gross, "Combining the Concepts of Compression and Caching for a Two-Level File System," *Proc. Fourth Architectural Support for Programming Languages and Operating Systems (ASPLOS '91),* pp. 200-211, Apr. 1991.
[33] H. Tang and T. Yang, "An Efficient Data Location Protocol for Self-Organizing Storage Clusters," *Proc. ACM/IEEE Conf. Supercomputing (SC '03),* Nov. 2003.
[34] P. Cao, E.W. Felten, and K. Li, "Application-Controlled File Caching Policies," *Proc. Usenix Summer Technical Conf.,* pp. 171-182, June 1994.
[35] A.R. Butt, C. Gniady, and Y.C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms," *Proc. ACM SIGMETRICS Int'l Conf. Measurements and Modeling of Computer Systems,* pp. 157-168, June 2005.
[36] J.L. Steffen, "Interactive Examination of a C Program with Cscope," *Proc. Winter Usenix Technical Conf.,* Jan. 1985.
[37] U. Manber and S. Wu, "GLIMPSE: A Tool to Search through Entire File Systems," *Proc. Winter Usenix Technical Conf.,* pp. 23-32, 1994.
[38] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, "Basic Local Alignment Search Tool," *J. Molecular Biology,* vol. 215, no. 3, pp. 403-410, http://dx.doi.org/10.1006/jmbi.1990.9999, Oct. 1990.
[39] K.T. Pedretti, T.L. Casavant, R.C. Braun, T.E. Scheetz, C.L. Birkett, and C.A. Roberts, "Three Complementary Approaches to Parallelization of Local Blast Service on Workstation Clusters," *Proc. Fifth Int'l Conf. Parallel Computing Technologies (PACT '99),* invited paper, pp. 271-282, 1999.
[40] TPC, Transaction Processing Council, http://www.tpc.org, 2007.
[41] D. Thiebaut, H.S. Stone, and J.L. Wolf, "Improving Disk Cache Hit-Ratios through Cache Partitioning," *IEEE Trans. Computers,* vol. 41, no. 6, pp. 665-676, June 1992.
[42] J.R. Spirn, *Program Behavior: Models and Measurements.* Elsevier Science, 1977.
[43] C. Gniady, A.R. Butt, Y.C. Hu, and Y.-H. Lu, "Program Counter-Based Prediction Techniques for Dynamic Power Management," *IEEE Trans. Computers,* vol. 55, no. 6, pp. 641-658, June 2006.

Yifeng Zhu received the BSc degree in electrical engineering from Huazhong University of Science and Technology, Wuhan, China, in 1998 and the MS and PhD degrees in computer science from the University of Nebraska, Lincoln, in 2002 and 2005, respectively. He is an assistant professor in the Electrical and Computer Engineering Department at the University of Maine. His main research interests are parallel I/O storage systems, cluster computing, grid computing, and computer architecture and systems. He is a member of the ACM, the IEEE, the IEEE Computer Society, and the Francis Crowe Society.

Hong Jiang received the BSc degree in computer engineering from Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in computer engineering from the University of Toronto, Canada, in 1987, and the PhD degree in computer science from Texas A&M University, College Station, in 1991. Since August 1991, he has been with the University of Nebraska, Lincoln, where he is currently a professor and the vice chair of the Department of Computer Science and Engineering. His current research interests are computer architecture, computer storage systems and parallel I/O, parallel/distributed computing, cluster and grid computing, performance evaluation, real-time systems, middleware, and distributed systems for distance education. He has more than 130 publications in major journals and international conference proceedings in these areas. His research has been supported by the US National Science Foundation (NSF), US Department of Defense (DoD), and the State of Nebraska. He is a member of the ACM, the IEEE, the IEEE Computer Society, and ACM SIGARCH.