RACE: A Robust Adaptive Caching Strategy for Buffer Cache

Yifeng Zhu
Electrical and Computer Engineering
University of Maine
Orono, ME 04469
zhu@eece.maine.edu

Hong Jiang
Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588
jiang@cse.unl.edu

***Abstract*— While 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 RACE, which differentiates the locality of I/O streams by actively detecting access patterns inherently exhibited in two correlated spaces: 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 global I/O regularities of an application and local I/O regularities of individual files accessed in that application to accurately estimate the locality strength, which is crucial in deciding which blocks to be replaced upon a cache miss. Through comprehensive simulations on real-application traces, RACE is shown to significantly outperform LRU and all other state-of-the-art cache management schemes studied in this paper, in terms of absolute hit ratios.**

I. Introduction

Designing an effective block replacement algorithm is an important issue in improving memory performance. In most systems, the replacement algorithm is based on the Least-Recently-Used (LRU) scheme [1], [2] or its clock-based approximation [3]. While LRU has the advantages of simple implementation and constant space and time complexity. it often suffers severely from two pathological cases.

- *Scan pollution*. After a long series of sequential accesses to cold blocks, many frequently accessed blocks may be evicted out from the cache immediately, leaving all these cold blocks occupying the buffer cache for an unfavorable amount of time and thus resulting in a waste of the memory resources.
- *Cyclic access to 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.

To address the limitations of the LRU scheme, several novel and effective replacement algorithms [4], [5], [6], [7] 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, DEAR [4] observes the patterns of references issued by a single application, assuming that the I/O patterns of each application is consistent. Since many applications access multiple files and exhibit a mixture of access patterns, as shown in [6] and later in this paper, this approach tends to have a large amount of inertia or reluctance and may not responsively detect local patterns, although it can correctly recognize global patterns.
2) At the file level, UBM [6] 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. The file-based detection [6] 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 thus is 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) At the program context level, PCC [5] and AMP [7] 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. This approach trains only for each program context and has a relatively shorter learning period than the file-based one. While 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 bases its decision on aggregate statistical information and thus is not sensitive to pattern changes over an individual file. In fact, as explained in Section III, multiple program contexts may access the same set of files but exhibit different patterns if observed from the program-context point of view.

This paper presents a novel approach for buffer cache management, called RACE (Robust Adaptive Caching Enhancement for buffer cache). Our new scheme 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. Our trace-driven simulations by using real life workloads show that RACE significantly outperforms LRU, PCC, AMP, UBM, LIRS and ARC cache management schemes.

The rest of this paper is organized as follows. Section II briefly reviews relevant studies in buffer cache management. Section III explains our RACE design in detail. Section IV presents the trace-driven evaluation method and Section V evaluates the performances of RACE and other algorithms and discusses the experimental results. Finally, Section VI concludes this paper.

II. Related Work on Buffer Cache Replacement Strategies

Off-line optimal policy [8], [9], replacing the block whose next reference is farthest in the future, provides a useful theoretical upper bound on the achievable hit ratio of all practical cache replacement policies. As described below, the practical replacement algorithms proposed in the last few decades 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 detects the I/O access patterns.

A. Deeper-history Based Replacement

To avoid the two pathological cases in LRU, described in the previous section, The replacement algorithms LRU-K [10], 2Q [11], LRFU [12], EELRU [13], MQ [14], LIRS [15], and ARC [16] incorporate longer reference histories than LRU. These algorithms base their cache replacement decisions on a combination of recency [17] 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 [5].

B. Reactive Hint Based Replacement

Application-informed caching management schemes are proposed in ACFS [18] and TIP [19], and they rely on programmers to insert useful hints to inform operating systems of future access patterns. However, this technique cannot achieve satisfactory performance level if the I/O access pattern is only known at runtime.

Artificial intelligence tools [20] are proposed to learn these I/O patterns at execution time and thus obtain the hints dynamically.

C. 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 block-level pattern detection policy is SEQ [21], which detects the long sequences of page cache misses and applies the Most-Recently-Used(MRU) [22] policy to such sequences to avoid scan pollution.

At the application level, DEAR (Detection Adaptive Replacement) [4] 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.

At the file level, the UBM (Unified Buffer Management) [6] scheme separates the I/O references according to their target files and automatically classifies the access pattern of each individual file into one of three categories: *sequential references*, *looping references* and *other references*. It divides the buffer cache into three partitions, one for blocks belonging to each pattern category, and then uses different replacement policies on different partitions. For blocks in the sequentially-referenced partition, MRU replacement policy is used, since those blocks are never revisited. For blocks in the periodically referenced partition, a block with the longest period is first replaced 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 Program Counter based Cache (PCC) [5] algorithm exploits virtual program counters exhibited in 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. While UMB classifies the I/O access patterns based on files, PCC classifies the patterns based on the virtual program counters of the I/O instructions in the program code. Adaptive Multi-Policy caching scheme (AMP) [7] inherits the design of PCC but proposes a new pattern detection algorithm. It defines an experiential mathematical expression to measure *recency* and classifies program counters according to the comparison between the average recency and a static threshold.

III. The Design of RACE Algorithm

This section presents the design of the RACE caching algorithm in detail. We first introduce the recently developed PC-based technology in buffer caching and then presents the details of our RACE algorithm.

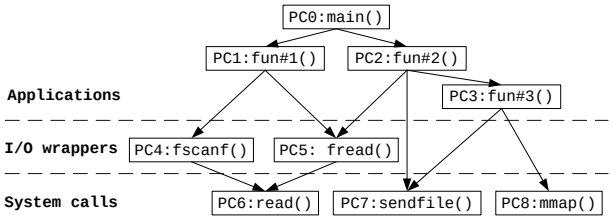

Fig. 1. An example call graph of some application

A. Concepts of Program Counters

Program counters, which indicate the location of the instructions in memory, have been utilized in cache management algorithms, such as including PCC [5] and AMP [7]. 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 example call is given in Figure 1. To uniquely identify the program context from which an I/O operation is invoked, a *program counter signature* is defined as the sum of the program counters of all functions along the I/O call path back to *main()*. For simplicity, program signatures are denoted as PCs in the rest of this paper.

B. 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 behaviors of current requested data.* While UBM only associates its prediction with the data's past access behaviors, 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 V.

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

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

Figure 2 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 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 statistic status of each PC process, resulting in 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.

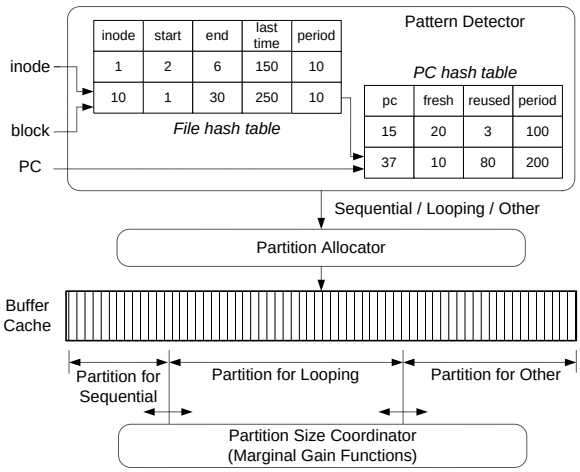

Fig. 2. Overall structures of the RACE scheme. The partition allocator and the Partition size coordinator take the results of pattern detector to adaptively fine-tune the size of each cache partition.

In fact, many PCs exist in one application and it is likely more than two 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 access period. The updating method is based on the assumption that the access periods of all blocks in a sequence are exactly the same. Thus we reduce the problem of finding the access period of the current block to identifying the access period of the first block in this sequence. 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 or 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

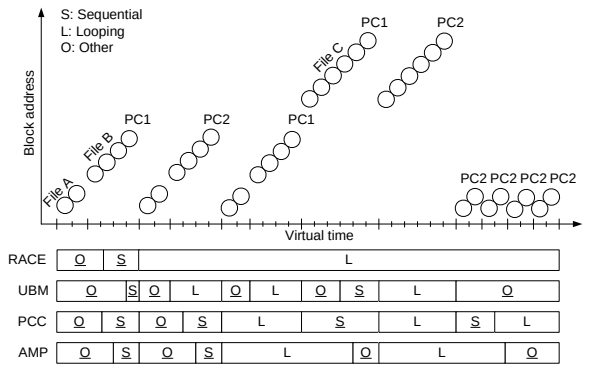

Fig. 3. An example of reference patterns. The sequentiality thresholds for UBM, PCC and RACE are 3. The sequentiality threshold, looping threshold and exponential average parameter for AMP are 0.4, 0.01, and 4 respectively. All incorrect classification results are underscored.

block has been visited before, a "looping" pattern is returned. 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 are 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 $O(1)$.

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 the access patterns. An example, shown in Figure 3, is used to illustrate and compare 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 File A and B, RACE incorrectly classifies the first three blocks as *other* and then 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 File A and File 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 File B and C.

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 While the blocks of a sequential access detected by UBM has 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 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 that 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 workloads 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 V.

IV. Application Traces Used in the Simulation Study

The traces used in this paper are obtained by using a trace collection tool provided by [5]. This tool is built upon the Linux *strace* utility that intercepts and records all system calls and signals of traced applications. The modified *strace* investigates all I/O-related activities and reports the I/O triggering PC, file identifier(*inode*), I/O staring address and request size in bytes.

We use trace-driven simulations with three types of workloads, *cscope*, *gcc* and *gnuplot*, to evaluate the RACE algorithm and compare it with other algorithms. The *cscope* and *gcc* are used in [23], [5], [15], [6], [7] and the *gnuplot* is used in [4]. Table I summarizes the characteristics of these traces.

TABLE I
Traces used and their statistics

Trace	*cscope*	*gcc*	*gnuplot*
Request Num.	2131194	8765174	677442
Data Size (MB)	240	89.4	121.8
File Num.	16613	19875	8
PC Num.	40	69	26

1) **cscope** [24] 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 240MB and the index database is around 16MB.
2) **gcc** is a GNU C compiler trace and it compiles and builds Linux kernel 2.6.10.
3) **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 52MB and 70MB, respectively.

We plot the traces of *cscope*, *gcc* and *gnuplot* in Figures 4, 5, and 6, respectively, showing 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.

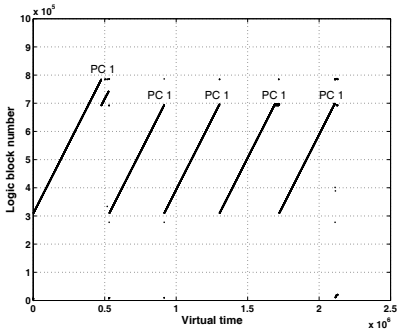

Fig. 4. cscope trace.

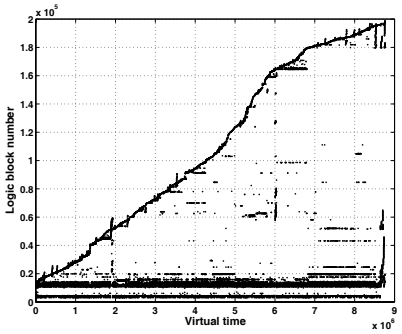

Fig. 5. gcc trace.

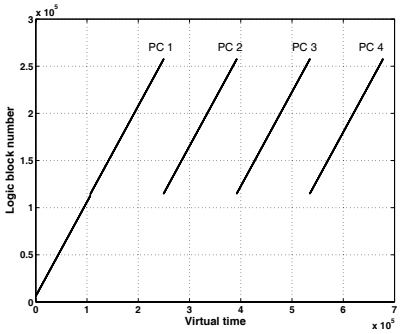

Fig. 6. gnuplot trace.

V. Performance Evaluation

This section presents the performance evaluation of RACE by comparing it with a number of most recent and representative cache replacement algorithms, through trace-driven simulations. We use hit ratio as our primary metric and compare the performance of RACE with seven other replacement algorithms, including UBM [6], PCC [5], AMP [7], LIRS [15], [25], ARC [16], LRU and the off-line optimal policy (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 [26]. We also implemented the PCC simulator and our RACE simulator by modifying the UBM simulator code. The 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.

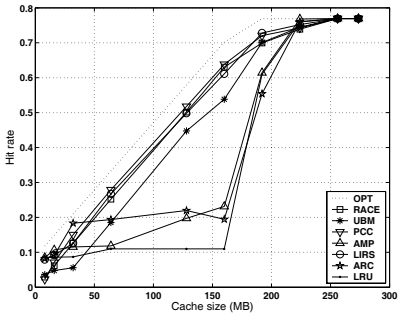

Fig. 7. Hit ratios for cscope.

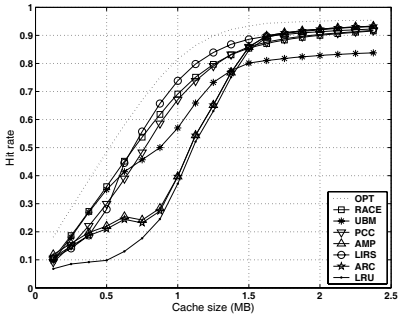

Fig. 8. Hit ratios for gcc.

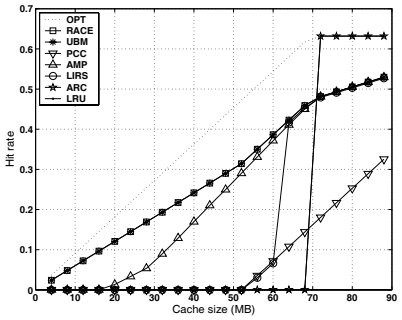

Fig. 9. Hit ratios for gnuplot.

The experimental results on these workloads, given in Figures 7, 8 and 9 show that RACE is more robust than all other algorithms. RACE can successfully overcome the drawbacks of LRU and improve its absolute hit ratios by as much as 52.0%, 37.3% and 45.9% in *cscope*, *gcc* and *gnuplot* respectively. Compared with other state-of-the-art pattern-detection based schemes, RACE outperforms UBM, PCC and AMP by as much as 6.0%, 21.5% and 12.4%, with an average of 3.0%, 7.4% and 8.3%, across all traces, respectively. In the *cscope* trace, RACE is 1.0% inferior to PCC on average due to the following fact: Although RACE correctly classifies files accessed at the end of first iteration as *looping*, these files are only accessed twice, as shown in Figure 4, 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% and 0.7% inferior to LIRS in absolute hit ratio. Since RACE improves the hit ratios of LIRS with an average of 6.0% over the eight workloads, we believe that such slight performance degradation in *cscope* and *gcc* is not severe. The *gcc* workload is extremely LRU-friendly, in which 89.4 MB data is accessed and a LRU cache with a size of 1.5MB can achieve a hit ratio of 86%. 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 sum, RACE improves the hit ratios of UBM, PCC, AMP, LIRS, ARC, and LRU relatively by 18.1%, 25.9%, 39.8%, 43.2%, 50.9% and 60.5% 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 the future access patterns are highly correlated with both program contexts and requested data.

VI. Conclusions

In this paper, we propose a novel and simple block replacement algorithm called RACE. Simulation study conducted under three 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.

Our study has two limitations. First, we have not implemented our design and evaluated it in real systems. Secondly, 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.

References

[1] M. J. Bach, *The design of the UNIX operating system*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[2] A. S. Tanenbaum and A. S.Woodhull, *Operating Systems Design and Implementation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[3] R. W. Carr and J. L. Hennessy, "WSCLOCK - a simple and effective algorithm for virtual memory management," in *Proceedings of the eighth ACM symposium on Operating systems principles (SOSP)*. New York, NY, USA: ACM Press, 1981, pp. 87–95.

[4] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "An implementation study of a detection-based adaptive block replacement scheme," in *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999, pp. 239–252.

[5] C. Gniady, A. R. Butt, and Y. C. Hu, "Program-counter-based pattern classification in buffer caching." in *Proceedings of 6th Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2004, pp. 395–408.

[6] 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," in *4th Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2000, pp. 119–134.

[7] F. Zhou, R. von Behren, and E. Brewer, "AMP: Program context specific buffer caching," in *Proceedings of the USENIX Technical Conference*, Apr. 2005.

[8] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[9] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies." *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[10] 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 international conference on Management of data*. New York, NY, USA: ACM Press, 1993, pp. 297–306.

[11] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450.

[12] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies," in *Proceedings of the 1999 ACM SIGMETRICS International conference on Measurement and Modeling of Computer Systems*, 1999, pp. 134–143.

[13] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: simple and effective adaptive page replacement," in *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, 1999, pp. 122–133.

[14] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 91–104.

[15] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2002, pp. 31–42.

[16] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, Mar. 2003, pp. 115–130.

[17] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2003, pp. 245–257.

[18] 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 Transactions on Computer Systems*, vol. 14, no. 4, pp. 311–343, 1996.

[19] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP)*. New York, NY, USA: ACM Press, 1995, pp. 79–95.

[20] T. M. Madhyastha and D. A. Reed, "Learning to classify parallel input/output access patterns," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 8, pp. 802–813, 2002.

[21] G. Glass and P. Cao, "Adaptive page replacement based on memory reference behavior," in *Proceedings of the ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM Press, 1997, pp. 115–126.

[22] K. So and R. N. Rechtschaffen, "Cache operations by MRU change." *IEEE Trans. Computers*, vol. 37, no. 6, pp. 700–709, 1988.

[23] P. Cao, E. W. Felten, and K. Li, "Application-controlled file caching policies," in *USENIX Summer Technical Conference*, June 1994, pp. 171–182.

[24] J. L. Steffen, "Interactive examination of a C program with Cscope," in *Proceedings of Winter USENIX Technical Conference*, Jan. 1985.

[25] J. Song and Z. Xiaodong, "Making LRU friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance," *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 939–952, 2005.

[26] N. Megiddo and D. S. Modha, "One up on LRU," *;login: - The Magazine of the USENIX Association*, vol. 4, no. 18, pp. 7–11, 2003.

Algorithm 1 Pseudocode for RACE pattern detection algorithm.

1: **RACE(**$inode$**,** $block$**,** PC**,** $currTime$**)**
2:
3: /* Updating the file hash table */
4: Find an entry f in the file hash table such that $f.inode == inode$ and $f.start \leq block \leq f.end$;
5: **if** $f \neq NULL$ **then**
6: $lastTime^{new} = currTime - (block - f.start)$; /* infer the last reference time of the block $start$ */
7: $f.period = \alpha \cdot f.period + (1-\alpha) \cdot (lastTime^{new} - f.lastTime)$; /* exponential average */
8: $f.lastTime = lastTime^{new}$;
9: **else**
10: Find an entry $f2$ in the file hash table, such that $f2.inode == inode$ and $f2.end == block - 1$;
11: **if** $f2 \neq NULL$ **then**
12: $f2.end = block$; /* extend existing sequence */
13: **else**
14: Insert ($inode$, $block$, $block$, $currTime$, ∞) into the file hash table; /* start a new sequence */
15: **end if**
16: **end if**
17:
18: /* Updating the PC hash table */
19: **if** PC is not in the PC hash table $pcTable$ **then**
20: Insert (pc, 0, 0, ∞) into $pcTable$;
21: **else**
22: **if** $f \neq NULL$ **then**
23: $pcTable[PC].reused++$;
24: $pcTable[PC].fresh--$;
25: $pcTable[PC].period = \beta \cdot period + (1-\beta) \cdot pcTable[PC].period$; /* exponential average */
26: **else**
27: $pcTable[PC].fresh++$;
28: **end if**
29: **end if**
30:
31: /* Detecting pattern */
32: **if** $f \neq NULL$ **then**
33: **return**("looping", $f.period$);
34: **else if** $pcTable[PC].reused \geq pcTable[PC].fresh$ **then**
35: **return**("looping", $pcTable[PC].period$);
36: **else if** $pcTable[PC].fresh >$ threshold **then**
37: **return**("sequential");
38: **else**
39: **return**("other");
40: **end if**