Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance

Song Jiang and Xiaodong Zhang, *Senior Member*, *IEEE*

Abstract—Although the LRU replacement algorithm has been widely used in buffer cache management, it is well-known for its inability to cope with access patterns with weak locality. Previously proposed algorithms to improve LRU greatly increase complexity and/or cannot provide consistently improved performance. Some of the algorithms only address LRU problems on certain specific and predefined cases. Motivated by the limitations of existing algorithms, we propose a general and efficient replacement algorithm, called *Low Inter-reference Recency Set* (LIRS). LIRS effectively addresses the limitations of LRU by using recency to evaluate Inter-Reference Recency (IRR) of accessed blocks for making a replacement decision. This is in contrast to what LRU does: directly using recency to predict the next reference time. Meanwhile, LIRS mostly retains the simple assumption adopted by LRU for predicting future block access behaviors. Conducting simulations with a variety of traces of different access patterns and with a wide range of cache sizes, we show that LIRS significantly outperforms LRU and outperforms other existing replacement algorithms in most cases. Furthermore, we show that the additional cost for implementing LIRS is trivial in comparison with that of LRU. We also show that the LIRS algorithm can be extended into a family of replacement algorithms, in which LRU is a special member.

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

1 Introduction

1.1 The Problems of the LRU Replacement Algorithm

The effectiveness of cache block replacement algorithms is critical to the performance stability of I/O systems. The LRU (Least Recently Used) replacement is widely used in managing buffer cache due to its simplicity, but many anomalous behaviors have been found with some typical workloads, where the hit rates of LRU may only slightly increase with a significant increase of cache size. The observations reflect LRU's inability to cope with access patterns with weak locality such as file scanning, regular accesses over more blocks than the cache size, and accesses on blocks with distinct frequency. Here are some representative examples reported in the research literature to illustrate how poorly LRU behaves:

1. Under the LRU algorithm, a burst of references to infrequently used blocks, such as sequential scans through large files, may cause the replacement of frequently referenced blocks in cache. This is a common complaint in many commercial systems: Sequential scans can cause interactive response time to deteriorate noticeably [17]. An effective replacement algorithm would be able to prevent hot blocks from being evicted by cold blocks.
2. For a cyclic (loop-like) pattern of accesses to a file that is only slightly larger than the cache size, LRU always mistakenly evicts the blocks that will be accessed the soonest because these blocks have not been accessed for the longest time [22]. A wise replacement algorithm would maintain a hit rate proportional to the buffer cache size.
3. In an example of multiuser database application, each record is associated with a B-tree index [17]. For a given number of records, assume their index entries can be packed into 100 blocks and 10,000 blocks are needed to hold the records. We use $R(i)$ to represent an access to Record i and $I(i)$ to Index i. The database application alternates its references to random index blocks and to the record blocks in the access sequence of $I(1)$, $R(1)$, $I(2)$, $R(2)$, $I(3)$, $R(3)$, ... Thus, the index blocks will be referenced with a probability of 0.005 and the data blocks are with a probability of 0.00005. Suppose that the cache can only hold 101 blocks. Ideally, all 100 index blocks are cached and only one record block is cached. However, LRU caches the 101 most recently accessed blocks. So, LRU keeps an equal number of index and record blocks in the cache and perhaps even more record blocks than index blocks. An intelligent replacement algorithm would choose the resident blocks according to their reference probability. Only those blocks with relatively high access probability deserve to stay in the cache for a longer time.

The reason for LRU to behave poorly in these situations is that LRU makes a bold assumption—a block that has not

- *S. Jiang is with the Performance and Architecture (PAL) Group, Los Alamos National Laboratory, CCS-3, B256, PO Box 1663, Los Alamos, NM 87545. E-mail: sjiang@lanl.gov.*
- *X. Zhang is with the Computer Science Department, College of William and Mary, Williamsburg, VA 23187. E-mail: zhang@cs.wm.edu.*

Manuscript received 26 Nov. 2003; revised 5 Nov. 2004; accepted 2 Mar. 2005; published online 15 June 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0227-1103.

been accessed for the longest time would wait for the longest time to be accessed again. This assumption cannot capture the access patterns exhibited in those workloads with weak locality. Generally speaking, there is less locality in buffer caches than that in CPU caches or virtual memory systems [20].

Meanwhile, LRU has its distinctive merits: simplicity and adaptability. It only samples and makes use of very limited history information—recency. While addressing the weakness of LRU, existing algorithms either take more history information into consideration, such as LFU (Least Frequently Used)-like ones in the cost of simplicity and adaptability or switch temporarily from LRU to other algorithms whenever certain predefined regularities are detected. In the switch-based approach, these algorithms actually act as supplements of LRU in a case-by-case fashion. To make a prediction of future access times, these algorithms assume the existence of a relationship between the future reference of a block with the behaviors of those blocks in its temporal or spatial locality, while LRU only associates the future behavior of a block with the block's own previous reference. This additional assumption increases the complexity of their implementations as well as their performance dependence on some specific characteristics of workloads. The replacement algorithm we propose, called LIRS, only samples and makes use of the same history information as LRU does—recency, and mostly retains the LRU assumption. Thus, it is simple and adaptive. In our design, LIRS does not directly target specific LRU problems, but fundamentally addresses the limitations of LRU.

1.2 An Executive Summary of Our Algorithm

We use recent Inter-Reference Recency (IRR) as the history information of a block, where the IRR of a block refers to the number of other distinct blocks accessed between two consecutive references to the block (IRR is also called reuse distance in some literature). In contrast, recency refers to the number of other distinct blocks accessed from last reference to the current time. We refer to the IRR between the last and the second-to-last references to a block as recent IRR or simply call it IRR without ambiguity in the rest of the paper. We assume that if the IRR of a block is large, the next IRR of the block is likely to be large. Following this assumption, we select the blocks with large IRRs for replacement because it is highly possible that these blocks will be evicted later by LRU before being referenced again under our assumption. It is noted that these evicted blocks may have been recently accessed, i.e., each has a small recency.

By adequately considering IRR in history information in our algorithm, we are able to eliminate negative effects caused by only considering recency, such as the problem shown in the aforementioned examples. When deciding which block to evict, our algorithm utilizes the block IRR information. It dynamically and responsively distinguishes low IRR (denoted as LIR) blocks from high IRR (denoted as HIR) blocks and keeps the LIR blocks in the cache, where the block recency is only used to help determine the LIR or HIR statuses of the blocks. We maintain an LIR block set and an HIR block set and manage to limit the size of the LIR set so that all the LIR blocks fit in the cache. The blocks in the LIR set are not selected for replacement and there are no misses for the references to these blocks. Only a very small portion of cache is allocated to store HIR blocks. Resident HIR blocks may be evicted at any recency. However, when the recency of an LIR block increases to a certain value and an HIR block gets accessed at a smaller recency than that of the LIR block, the statuses of the two blocks are switched. We name the proposed algorithm *Low Inter-reference Recency Set* (denoted as LIRS) replacement because the LIR set is what the algorithm tries to identify and keep in the cache. LIRS aims at addressing three issues in designing replacement algorithms: 1) how to effectively utilize multiple sources of history access information, 2) how to dynamically and responsively distinguish blocks by comparing their possibility to be referenced in the near future, and 3) how to minimize implementation overheads.

In the next section, we give an overview of the related work and highlight our technical contributions. The LIRS algorithm is described in Section 3. In Section 4, we present the trace-driven simulation results for performance evaluation and comparisons. We provide sensitivity and overhead analysis of the proposed replacement algorithm in Section 5 and conclude the paper in Section 6.

2 Related Work

The LRU replacement is widely used for the management of virtual memory, file buffer caches, and data buffers in database systems. The three representative problems described in the previous section are found in the different application fields. Many efforts have been made to address the LRU problems. We classify existing algorithms into three categories: 1) replacement algorithms based on user-level hints, 2) replacement algorithms based on tracing and utilizing history information of block accesses, and 3) replacement algorithms based on regularity detections.

2.1 User-Level Hints

Application-controlled file caching [3] and application-informed prefetching and caching [19] are the schemes based on user-level hints. These schemes identify blocks less likely to be reaccessed in the near future based on the hints provided by user programs. To provide appropriate hints, programmers need to understand the data access patterns, which adds to the programming burden. In [15], Mowry et al. attempted to abstract hints by compilers to facilitate I/O prefetching. In contrast, the LIRS algorithm can adapt its behavior to different access patterns without explicit hints. While the hint-based methods are orthogonal to the LIRS replacement, the collected hints may help LIRS refine the correlation of consecutive IRRs.

2.2 Tracing and Utilizing History Information

Realizing that LRU only utilizes limited access information, some researchers have proposed several algorithms to collect and use "deeper" history information, which include the LFU-like algorithms such as FBR, MQ, LRFU, as well as LRU-K and 2Q. We adopt a similar approach by effectively collecting and utilizing access information to design the LIRS replacement.

Robinson and Devarakonda proposed a frequency-based replacement algorithm (FBR) by maintaining reference counts for the purpose to "factor out" locality [20]. Zhou et al. proposed Multi-Queue (MQ), which sets up multiple queues and uses access frequencies to determine which

queue a block should be in [23]. However, it is slow for the frequency-based algorithms to respond to reference frequency changes and some of their parameters have to be found by trial and error. Having analyzed the advantages and disadvantages of LRU and LFU, Lee et al. proposed LRFU by combining them through weighing block recency and frequency factors [14]. The performance of the LRFU algorithm largely relies on a parameter called λ, which determines the relative weight of LRU or LFU and has to be adjusted according to the system configurations, even according to different workloads. However, LIRS does not have a tunable parameter that is sensitive to workloads.

The LRU-K algorithm addresses the LRU problems presented in examples 1 and 3 in the previous section [17]. LRU-K makes its replacement decision by comparing the times of the Kth-to-last references to blocks. After such a comparison, the oldest resident block is evicted. For simplicity, the authors recommended K = 2. By taking the time of the second-to-last reference to a block as the basis for comparison, LRU-2 can quickly remove cold blocks from the cache. However, for blocks without significant differences of reference frequencies, LRU-2 does not work well. In addition, LRU-2 is expensive: Each block access requires $\log(N)$ operations to manipulate a priority queue, where N is the number of blocks in the cache.

Johnson and Shasha proposed the 2Q algorithm that has constant time overhead [10]. They showed that the algorithm performs as well as LRU-2. The 2Q algorithm can quickly remove sequentially referenced blocks and loopingly referenced blocks with long looping intervals out of the cache. This is achieved by using a special buffer, called queue *A1in*, in which all missed blocks are initially placed. When the blocks are replaced from the *A1in* queue in a FIFO order, the addresses of those replaced blocks are temporarily placed in a ghost buffer called queue *A1out*. When a block is rereferenced, it is promoted to a main buffer called queue *Am* if its address is in the *A1out* queue. That is, only blocks that have a short reuse distance measured in *A1in* and *A1out* can be cached for a long time in *Am*. In this way, they are able to distinguish frequently referenced blocks from those infrequently referenced. By setting the sizes of the *A1in* and *A1out* queues as constants Kin and $Kout$, respectively, 2Q provides a victim block either from *A1in* or from *Am*. However, Kin and $Kout$ are predetermined parameters, which need to be carefully tuned and are sensitive to the types of workloads. While both 2Q and LIRS have simple implementations with low overheads, LIRS has overcome the drawbacks of 2Q by properly updating the LIR block set. Another recent algorithm, ARC, maintains two variable-size lists [16]. Their combined size is two times the number of blocks that are held in the cache. One half of the lists contain the blocks in the cache and the other half are for the history access information of replaced blocks. The first list contains the blocks that have been seen only once recently (cold blocks) and the second list contains the blocks that have been seen at least twice recently (hot blocks). The buffer spaces allocated to the blocks in these two lists are adaptively changed, depending upon in which list recent misses take place. More buffer spaces will serve cold blocks (respectively, hot blocks) if there are more cold block (respectively, hot block) accesses. However, although the authors advocated the superiority of the ARC algorithm with its adaptiveness and avoidance of tunable parameters, the locality of the blocks in the two lists, quantified by recency or frequency, cannot be directly and consistently compared. For example, a block that is regularly accessed with an IRR a little bit more than the cache size may have no hits at all, while a block in the second list can stay in the cache without any accesses since it has been accepted into the list.

The Inter-Reference Gap (IRG) of a block is the number of the references between consecutive references to the block, which is different from IRR on whether duplicate references to a block are counted. Phalke and Gopinath considered the correlation between history IRGs and future IRGs [18]. The past string of IRGs of a block is modeled by Markov chain to predict its next IRG. However, as Smaragdakis et al. indicated, replacement algorithms based on a Markov model fail in practice because they try to solve a much harder problem than the replacement problem itself [22]. An apparent difference in their algorithm from the LIRS algorithm is in how to measure the distance between two consecutive references to a block. Our study shows that IRR is more justifiable than IRG in this circumstance. First, IRR only counts the distinct blocks and filters out high-frequency events, which may be volatile with time. Thus, the IRR is more relevant to the next IRR than the IRG to the next IRG. Moreover, it is the "recency" rather than the "gap" information that is used by LRU. An elaborate argument favoring IRR in the context of virtual memory page replacement can be found in [22]. Second, IRR can be easily dealt with under the LRU stack model [2], on which most popular replacements are based.

2.3 Detection and Adaptation of Access Regularities

More recently, some researchers took another approach to detect access regularities from the history information by relating the accessing behavior of a block to those of the blocks in its temporal or spatial locality scope. Then, different replacements, such as Most Recently Used (MRU), can be applied to those blocks with specific access regularities.

Glass and Cao proposed the SEQ algorithm for adaptive page replacement in virtual memory management [9]. It detects sequential address reference patterns. If a long sequence of page faults with continuous addresses is found, MRU is applied to the sequence. If such a sequence is not detected, SEQ performs the LRU replacement. These detections only take place when there are page faults, so it has a low overhead acceptable in virtual memory management. However, Smaragdakis et al. argued that address-based detection lacks generality and advocated using aggregate recency information to characterize page behaviors [22]. Their EELRU examines aggregate recency distributions of accessed pages and changes the page eviction points using an online cost/benefit analysis by assuming the correlation among temporally contiguously referenced pages. This is different from LRU, which actually always sets the eviction point at the bottom of the LRU stack. However, EELRU has to choose an eviction point from a predetermined set of LRU stack positions. And, the way to select the set affects its performance. Moreover, by an aggregate analysis, EELRU cannot quickly respond to the changing access patterns. Without spatial or temporal detections, LIRS uses the independent recency events of each block to effectively characterize their references.

Choi et al. proposed an adaptive buffer management algorithm called DEAR, which automatically detects the block reference patterns of applications and applies different replacement algorithms to different applications based on their detected reference patterns [5]. Further, they proposed an Application/File-level Characterization (AFC) algorithm in [4], which first detects the reference characteristics at the application level and then at the file level if necessary. Accordingly, appropriate replacement algorithms are used to the blocks with different patterns. The Unified Buffer Management (UBM) algorithm by Kim et al. also detects patterns in the recorded history [13]. Unlike the detection method used in DEAR, which associates the backward distance and frequency with the forward distances of blocks between two consecutive detection invocation points, UBM tracks the reference information such as the file descriptor, start block number, end block number, and loop period if a rereference occurs. More recently, Gniady et al. proposed the PCC replacement algorithm, which conducts its access pattern detection on a per-system-call-site basis to improve the detection accuracy and efficiency [8]. Although these elaborate detections of access patterns provide a large potential for significant performance improvements, they addressed the LRU problems in a case-by-case fashion and have to deal with the allocation problem, which does not appear in LRU. To facilitate the online evaluation of buffer utilizations, certain premeasurements are needed to set predefined parameters used in the buffer allocation schemes [4], [5], [8]. LIRS does not have these design challenges. While it chooses the victim block in a global stack as LRU does, it can take the advantages provided by the detection-based algorithms.

More work on program locality analysis, prediction, and enhancement is conducted in the program behavior studies using static compiler analysis, data profiling, and runtime data analysis techniques (e.g., see [6]). There are two major differences between these studies and those on replacement algorithms in operating systems. First, program behavior studies are usually conducted at a finer level such as data elements and instructions rather than at the block or page level defined by the system. Usually, they require much more computing effort, which could be too expensive for a replacement algorithm running in the operating system. Second, program behavior studies focus on understanding the behavior of a specific program. It doesn't consider system parameters such as memory size and interaction among simultaneously running programs. However, a replacement algorithm must be designed from the system perspective, taking both the properties of workloads and system configurations into consideration. These constraints prevent the replacement algorithm from conducting an aggressive locality analysis or pattern detection. Thus, a simple yet effective replacement algorithm becomes a critical system design issue.

3 The LIRS Algorithm

3.1 General Idea

We classify referenced blocks into two sets: High Inter-reference Recency (HIR) block set and Low Inter-reference Recency (LIR) block set. Each block with its history information in cache has a status—either LIR or HIR. Some HIR blocks may not reside in the cache, but keep their metadata in the cache, recording their status as nonresident HIR. We divide the cache, whose size in blocks is L, into a major part and a minor part in terms of their sizes. The major part, with its size of L_{lirs}, is used to store LIR blocks and the minor part, with its size of L_{hirs}, is used to store blocks from HIR block set, where $L_{lirs} + L_{hirs} = L$. When a miss occurs and a block is needed for replacement, we choose an HIR block that is resident in the cache. The blocks in the LIR block set always reside in the cache, i.e., there are no misses for the references to the LIR blocks. However, a reference to an HIR block is likely to encounter a miss because L_{hirs} is very small (its practical size can be as small as 1 percent of the cache size).

We use Table 1 as a simple example to illustrate how a replaced block is selected by the LIRS algorithm and how LIR/HIR statuses are maintained. In Table 1, symbol "X" denotes a block access at a virtual time.[1] As an example, block A is accessed at times 1, 6, and 8. Based on the definition of recency and IRR in Section 1.2, at time 10, blocks A, B, C, D, E have their IRR values of 1, 1, "infinite," 3, and "infinite," respectively, and have their recency values of 1, 3, 4, 2, and 0, respectively. We assume the cache can hold three blocks, $L_{lirs} = 2$ and $L_{hirs} = 1$, thus, at time 10, the LIRS algorithm leaves two blocks in the LIR set (the LIR set = {A, B}). The rest of the blocks go to the HIR set (the HIR set = {C, D, E}). Because block E is the most recently referenced, it is the only resident HIR block due to $L_{hirs} = 1$. If there is a reference to an LIR block, we keep it in the LIR block set. If there is a reference to an HIR block, we need to know whether we should change its status to LIR.

The key to successfully making the LIRS idea work in practice rests on whether we are able to dynamically and responsively maintain the LIR block set and HIR block set. When an HIR block is referenced, it gets a new IRR equal to its recency. Then, we determine whether the new IRR should be considered small relative to the current LIR blocks so that we know whether we need to change its status to LIR. Here, we have two options: compare the new IRR either with the IRRs or with the recencies of the LIR blocks. We take the recencies for the comparison for two reasons: 1) The IRRs are generated before their respective recencies and may be outdated, which is not directly relevant to the new IRR of the HIR block. A recency of a block is determined not only by its own reference activity, but also by the recent activities of other blocks. The outcome of comparing the new IRR and the recencies of the LIR blocks determines the eligibility of the HIR block to be considered as a hot block. While we state that IRRs are used to determine which blocks should be replaced, it is the new IRRs that are directly used in the comparisons. 2) If the new IRR of the HIR block is smaller than the recency of an LIR block, it will be smaller than the upcoming IRR of the LIR block. This is because the recency of the LIR block is a part of its upcoming IRR and not greater than the IRR. Thus, the comparisons with the recencies are actually the comparisons with the relevant IRRs. Once we know that the new IRR of the HIR block is smaller than the maximum recency of all the LIR blocks, we switch the LIR/HIR statuses of the HIR block and the LIR block with the maximum recency. Following this rule, we can 1) allow an HIR block with a relatively small IRR to join the LIR block

1. Virtual time is defined on the reference sequence, where a reference represents a time unit.

TABLE 1
An Example to Explain How a Victim Block Is Selected by the LIRS Algorithm and How LIR/HIR Statuses Are Maintained

Blocks / Virtual time	1	2	3	4	5	6	7	8	9	10	Recency	IRR
A	x					x		x			1	1
B			x		x						3	1
C				x							4	inf
D		x					x				2	3
E									x		0	inf

An "X" refers to the block in a row that is referenced at the virtual time of a column. The recency and IRR columns represent their respective values at virtual time 10 for each block. We assume $L_{lirs} = 2$ and $L_{hirs} = 1$ and, at time 10, the LIRS algorithm leaves two blocks in the LIR set (= {A, B}) and the HIR set is {C, D, E}. The only resident HIR block is E.

set in a timely fashion by replacing a LIR block from the set and 2) keep the size of LIR block set no larger than L_{lirs}, thus the entire set of blocks can reside in the cache.

Again, in the example of Table 1, if there is a reference to block D at time 10, a miss occurs. The LIRS algorithm replaces resident HIR block E, instead of block B, which would be replaced by LRU due to its largest recency. Furthermore, because block D is referenced, its new IRR is 2, which is smaller than the recency of LIR block B (= 3), indicating that the upcoming IRR of block B will not be smaller than 3. So, the status of block D is switched to LIR and the block joins the LIR block set, while block B becomes an HIR block. Since block B becomes the only resident HIR block, it is going to be evicted from the cache once another free block is requested. If, at virtual time 10, block C, with its recency of 4, rather than block D, with its recency of 2, gets accessed, there will be no status switching. Then, block C becomes a resident HIR block, while the replaced block is still E at virtual time 10. In this way, the LIR block set and HIR block set are formed and dynamically maintained.

3.2 The LIRS Algorithm Based on LRU Stack

The LIRS algorithm can be efficiently built on the model of LRU stack, which is an implementation structure of LRU. The LRU stack contains L entries, each of which represents a block.[2] Usually, L is the cache size in blocks. The LIRS algorithm makes use of the stack to keep track of recency and to dynamically maintain LIR block set and HIR block set. In contrast to the LRU stack, where only resident blocks are managed by the LRU algorithm in the stack, we store LIR blocks and HIR blocks with their recencies less than the maximum recency of the LIR blocks in a stack called LIRS stack S. S is similar to the LRU stack in operation but has a variable size. With this design, we do not need to explicitly record the IRR and recency values and to search for the maximum recency value. Each entry in the stack records the LIR/HIR status of a block and its residence status, indicating whether or not the block resides in the cache. To facilitate the search of the resident HIR blocks, we link all these blocks into a small stack, Q, with its size of L_{hirs}. Once a free block is needed, the LIRS algorithm removes a resident HIR block from the bottom of stack Q for replacement. However, the replaced HIR block remains in stack S with its residence status changed to "nonresident" if it is originally in the stack. We ensure the block in the bottom of stack S is an LIR block by removing HIR blocks below it. Once an HIR block in the LIRS stack gets referenced, which means there is at least one LIR block whose upcoming IRR will be greater than the new IRR of the HIR block (such as the one at the bottom of the stack), we switch the LIR/HIR statuses of the HIR block and the LIR block at the bottom. Then, the LIR block at the bottom is evicted from stack S and goes to the top of stack Q as a resident HIR block. This block will soon be replaced from the cache due to the small size of stack Q (at most L_{hirs}).

Such a design is partially inspired by the observation of improper LRU replacement behavior: If a block is evicted from the bottom of an LRU stack, it means the block occupies a buffer during the period of time when it moves from the top to the bottom of the stack without being referenced. Why do we have to afford a buffer for another long idle period when the block is loaded into the cache the next time as what LRU does? The rationale for the correction of the LRU decision is the assumption that temporal IRR locality holds for block references.

3.3 A Detailed Description

In the LIRS replacement, there is an operation called "stack pruning" on LIRS stack S, which removes the HIR blocks at the stack bottom until an LIR block sits there. This operation serves two purposes: 1) We ensure the block at the stack bottom always belongs to the LIR block set. 2) After the LIR block in the bottom is removed, those HIR blocks contiguously located above it will not have a chance to change their status from HIR to LIR since their recencies are larger than the new maximum recency of the LIR blocks.

When an LIR block set is not full, all the accessed blocks are given LIR status until its size reaches L_{lirs}. After that, HIR status is given to any blocks that are accessed for the first time and to blocks that have not been accessed for a long time so that currently they are not in stack S.

Fig. 1 shows a scenario where stack S holds three types of blocks, LIR blocks, resident HIR blocks, nonresident HIR blocks, and stack Q holds all of the resident HIR blocks. An HIR block could either be in stack S or not. Fig. 1 does not depict the nonresident HIR blocks that are not in stack S. There are three cases for the references to these blocks in the LIRS algorithm, which are also illustrated in Fig. 2, using the example shown in Table 1.

2. For simplicity, in the rest of the paper we use "a block in the stack" instead of "the entry of a block in the stack" without ambiguity.

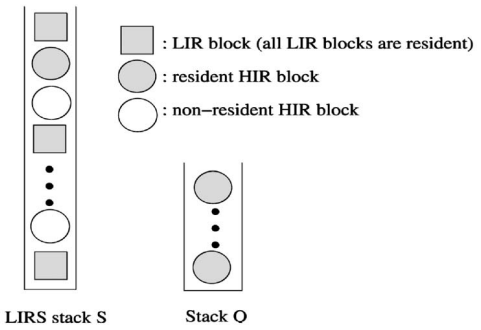

Fig. 1. LIRS stack S holds LIR blocks as well as some HIR blocks, with or without resident status, and stack Q holds all the resident HIR blocks.

1. **Upon accessing an LIR block X.** This access is guaranteed to be a hit in the cache. We move it to the top of stack S. If the LIR block is originally located at the bottom of the stack, we conduct a stack pruning. This case is illustrated in the transition from state (a) to state (b) in Fig. 2.
2. **Upon accessing an HIR resident block X.** This is a hit in the cache. We move it to the top of the stack S. There are two cases for the original location of block X: a) If X is in stack S, we change its status to LIR. This block is also removed from stack Q. The LIR block at the bottom of S is moved to the top of stack Q with its status changed to HIR. A stack pruning is then conducted. This case is illustrated in the transition from state (a) to state (c) in Fig. 2. b) If X is not in stack S, we leave its status unchanged and move it to the top of stack Q.
3. **Upon accessing an HIR nonresident block X.** This is a miss. We remove the HIR resident block at the bottom of stack Q (it then becomes a nonresident block) and evict it from the cache. Then, we load the requested block X into the freed buffer and place it at the top of stack S. There are two cases for the original location of block X: a) If X is in the stack S, we change its status to LIR and move the LIR block at the bottom of stack S to the top of stack Q with its status changed to HIR. A stack pruning is then conducted. This case is illustrated in the transition from state (a) to state (d) in Fig. 2. b) If X is not in stack S, we leave its status unchanged and place it at the top of stack Q. This case is illustrated in the transition from state (a) to state (e) in Fig. 2.

4 Performance Evaluation

4.1 Experiment Settings

We use trace-driven simulations with various types of workloads to evaluate the LIRS algorithm and compare it with other algorithms. We have adopted many application workload traces used in the previous studies aiming at addressing the LRU limitations. These are traces recording file access requests from one or multiple running applications, representing a wide range of access patterns, sizes, and sources. We have also generated a synthetic trace. Among these traces, cpp, cs, $glimpse$, and $postgres$ are used in [4], [5] (cs is named as $cscope$, and $postgres$ is named as $postgres2$ there), $sprite$ is used in [14], $multi1$, $multi2$, and $multi3$ are used in [13]. We briefly describe the traces here.

1. **2-pools** is a synthetic trace which simulates application behavior described in the third example in Section 1.1. The trace contains 100,000 references.
2. **cpp** is a GNU C compiler preprocessor trace. The total size of C source programs used as input is roughly 11 MB.
3. **cs** is an interactive C source program examination tool trace. The total size of the C programs used as input is roughly 9 MB.
4. **glimpse** is a text information retrieval utility trace. The total size of the text files used as input is roughly 50 MB.
5. **postgres** is a trace of join queries among four relations in a relational database system from the University of California at Berkeley.
6. **sprite** is from the Sprite network file system, which contains requests to a file server from client workstations for a two-day period.
7. **multi1** is obtained by executing two workloads, cs and cpp, together.
8. **multi2** is obtained by executing three workloads, cs, cpp, and $postgres$, together.
9. **multi3** is obtained by executing four workloads, cpp, $gnuplot$, $glimpse$, and $postgres$, together. $gnuplot$ is a popular graph plotting tool.

The only parameter of the LIRS algorithm, L_{hirs}, is set as 1 percent of the cache size or $L_{lirs} = 99\%$ of the cache size in the experiments. This selection results from a sensitivity study on the parameter, which is described in Section 5.1.

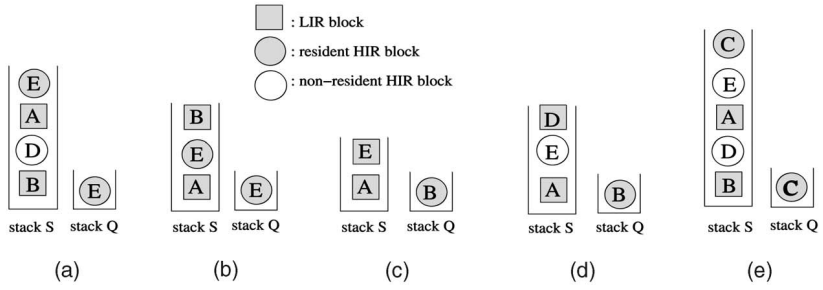

Fig. 2. Illustration of reference effects on the stacks using the example shown in Table 1. In the figure, (a) corresponds to the state at virtual time 9. References to B, E, D, or C at virtual time 10 result in states (b), (c), (d), and (e), respectively.

4.2 Access Pattern-Based Performance Evaluation

Through an elaborate investigation, Choi et al. classified the file cache access patterns into four types [4]:

- Sequential references: All blocks are accessed one after another and never reaccessed;
- Looping references: All blocks are accessed repeatedly with a regular interval (period);
- Temporally clustered references: Blocks accessed more recently are the ones more likely to be accessed in the near future;
- Probabilistic references: Each block has a stationary reference probability and all blocks are accessed independently with their associated probability.

The classification serves as a basis for their access pattern detections and for adapting to different replacement algorithms. For example, MRU applies to sequential and looping patterns, LRU applies to temporally clustered patterns, and LFU applies to probabilistic patterns. Though the LIRS algorithm does not rely on such a classification, we would like to use it to present and explain our experiment results. Because a sequential pattern is a special case of looping pattern (with infinite interval), we only use the last three types: looping, temporally clustered, and probabilistic patterns.

Algorithms LRU, LRU-2, 2Q, ARC, LRFU, and LIRS belong to the same replacement algorithm category. In other words, these algorithms take the same technical approach—predicting the access possibility of a block through its own history access information. Thus, we focus on the performance comparisons between LIRS and other algorithms in this category. As representative algorithms in the category of regularity detections, we choose two algorithms for comparisons: UBM for its spatial regularity detection and EELRU for its temporal regularity detection. UBM simulation requires the file ID, offset, and process ID of a reference. However, some traces available to us only consist of logical block numbers, which are unique numbers for the accessed blocks. Thus, we only produce the UBM simulation results for the traces used in paper [13], which are $multi1$, $multi2$, $multi3$. We also include the results of OPT, an optimum, offline replacement algorithm [2] for comparison.

We divide the traces into four groups based on their access patterns. Traces cs, $glimpse$, and $postgres$ belong to the looping type, traces *cpp* and *2-pools* belong to the probabilistic type, trace $sprite$ belongs to the temporally clustered type, and traces $multi1$, $multi2$, and $multi3$ belong to the mixed type.

We present the performance results for each trace using a pair of figures: the time-space maps and the hit rate curves. In a time-space map, the x axis represents virtual time, a position in the reference sequence of a given workload, and the y axis represents the logical block numbers of the accessed blocks. The hit rate curves show the hit rates with different cache sizes for the various replacement algorithms on a workload trace.

4.2.1 Performance for Looping Type Workloads

Fig. 3 plots three pairs of time-space maps and the hit rate curves generated by the various algorithms for workloads cs, $glimpse$, and $postgres$, respectively. The time-space maps show that all three programs have looping patterns with long intervals. As expected, LRU performs poorly for these workloads with the lowest hit rates. Let us take cs as an example, which has a pure looping pattern. Each block is accessed at almost the same frequency. Since all blocks in a loop have the same eligibility to be kept in the cache, it is desirable to keep the same set of blocks in the cache no matter what blocks are referenced currently. That is indeed what LIRS does: The same set of LIR blocks is fixed in the cache because the HIR blocks do not have IRRs small enough to change their status. In the looping pattern, recency indicates the opposite of the future reference time of a block: The larger the recency of a block is, the sooner the block will be referenced. The hit rate of LRU for cs is almost 0 percent until the cache size approaches 1,400 blocks, which can hold all the accessed blocks in a loop. It is interesting to see that the hit rate curve of LRU-2 overlaps with the LRU curve. This is because LRU-2 selects the same victim block as the one selected by LRU for replacement. When making a decision, LRU-2 compares the second-to-last reference time, which is the recency plus the recent IRG. However, the IRGs are the same for all the blocks at any time after the first reference. Thus, LRU-2 relies only on recency to make its decision, the same as LRU does. In general, when recency makes a major contribution to the second-to-last reference time, LRU-2 behaves similarly to LRU.

Except for cs, the other two workloads have mixed looping patterns with various sizes of intervals. LRU exhibits the stair-step hit rate curves for the workloads. LRU is not effective until all the blocks in its locality scope are brought into the cache. For example, only after the cache can hold 355 blocks does the LRU hit rate curve of $postgres$ have a sharp increase from 16.3 percent to 48.5 percent. Because LRU-2 considers the last IRG in addition to the recency, it is easier for it to distinguish blocks with different loop intervals than LRU does. However, LRU-2 lacks the capability of dealing with the varying recencies of these blocks. Our experiments show that the performance improvement achieved by LRU-2 over LRU is limited.

It is illuminating to observe the performance difference between 2Q and LIRS because both employ two linear data structures following a similar principle that only rereferenced blocks deserve to be in cache for a longer time. We can see that the hit rates of 2Q are significantly lower than those of LIRS for all three workloads. As the cache size increases, 2Q even performs worse than LRU for workloads $glimpse$ and $postgres$. Another observation for 2Q on $glimpse$ and $postgres$ is a serious "Belady's anomaly" [1]: Increasing the cache size could reduce the number of hits. Although ARC is an adaptive algorithm without tunable parameters, it actually shares the same problem as 2Q. The performance improvement of ARC over LRU is very limited. Belady's anomaly also appears in $glimpse$ for ARC. This is mainly caused by the inconsistent quantification and comparison of block locality in the two lists of ARC. This issue has been effectively addressed in LIRS. We will provide an in-depth analysis on this issue in Section 4.3.

LRFU, which combines LRU and LFU, is not effective on workloads with a looping pattern because the block reference frequencies in looping references are hard to distinguish. As an example, the LRFU and LRU hit rate curves for workload cs are overlapped.

Our simulation results show LIRS significantly outperforms all of the other algorithms and its hit rate curves are very close to those of OPT. Meanwhile, the results also

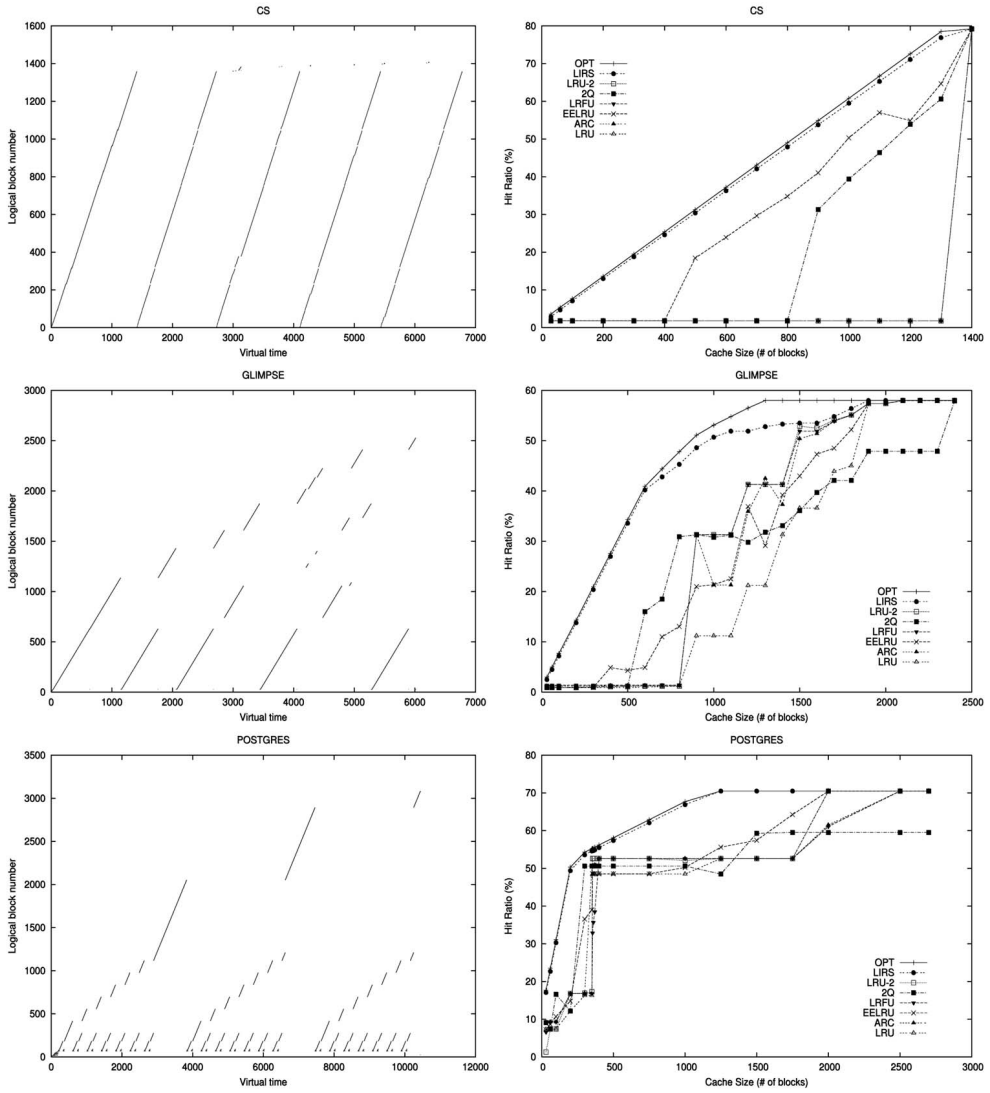

Fig. 3. The time-space maps and the hit rate curves of **cs**, **glimpse**, and **postgres** for the replacement algorithms.

show that the hit rates of cs and $postgres$ are closer to those of OPT than the hit rates of $glimpse$. This indicates that LIRS can make a more accurate prediction on the future LIR/HIR statuses when the looping intervals are of less variance. Because cs and $postgres$ have relatively fixed loop intervals, their consecutive IRRs are of less variance, which makes the IRR assumption hold well. However, the LIRS algorithm is not sensitive to the variance of IRRs, which is reflected by the significant hit rate improvements on workload $glimpse$. This is further evidenced by the results for the mixed pattern workloads described in Section 4.2.4.

4.2.2 Performance for the Probabilistic Type Workloads

Fig. 4 plots two pairs of time-space maps and the hit rate curves generated by the various replacement algorithms for traces *cpp* and *2-pools*, respectively. According to the detection results in [4], workload *cpp* exhibits a probabilistic reference pattern. In *cpp*, before the cache size increases to 100 blocks, the hit rates of LRU are much lower than those of LIRS. For example, when the cache size is 50 blocks, the hit rate of LRU is 9.3 percent, while the hit rate of LIRS is 55.0 percent. This is because holding a reference locality scope needs about 100 blocks. LRU cannot exploit the locality until enough cache space is available to hold all the recently referenced blocks. However, the capability for LIRS to exploit locality does not rely on the cache size—when it is identifying the LIR set, it always makes sure that the set will be able to fit in the cache. *2-pools* is generated to evaluate the replacement algorithms on their abilities to recognize the long-term reference behaviors. Though the reference frequencies are very different between the record blocks and the index blocks, it is hard for LRU to distinguish them when the cache size is small relative to the number of the referenced blocks because LRU takes only recency into consideration. The LRU-2, 2Q, and LIRS algorithms take one more previous reference into consideration—the time for the second-to-last reference to a block is involved. Even though the reference events to a block are randomized (i.e., the IRRs of a block are random with a certain fixed frequency, which is unfavorable to LIRS), LIRS still outperforms LRU-2 and 2Q. However, LRFU utilizes "deeper" history information. The constant long term frequency becomes more visible to the LFU-like algorithm. Thus, the performance of LRFU is slightly better than that of LIRS. It

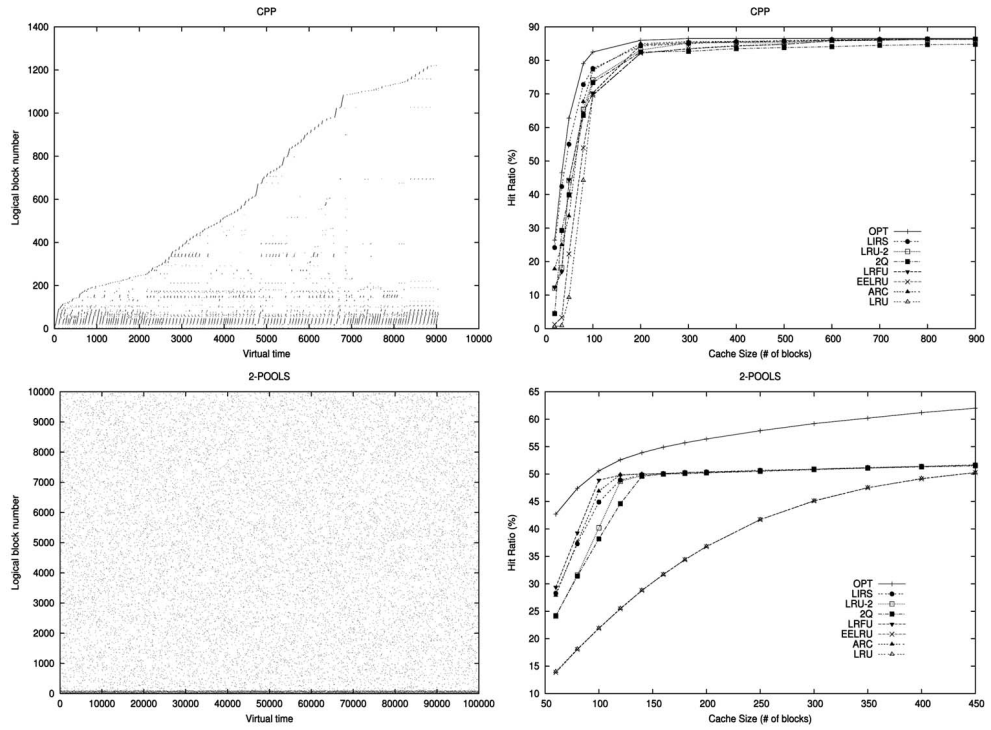

Fig. 4. The time-space maps and the hit rate curves of **cpp** and **2-pools** for the replacement algorithms.

is not surprising to see that the hit rate curve of EELRU overlaps with that of LRU, showing its poor performance. This is because EELRU relies on an analysis of a temporal recency distribution to decide whether to conduct an early point eviction. In *2-pools*, the blocks with high access frequency and the blocks with low access frequency are alternatively referenced, thus no sign of an early point eviction can be detected.

4.2.3 Performance for Temporally Clustered Type Workloads

Fig. 5 presents the time-space map of workload $sprite$ and its hit rate curves generated by the various replacement algorithms. $sprite$ exhibits a temporally clustered reference pattern. Fig. 5 shows that the LRU hit rate curve smoothly climbs with the increase of the cache size. Although there is still a gap between the LRU and OPT curves, the slope of the LRU curve is close to the OPT curve. $sprite$ is a so-called LRU-friendly workload [22], which seldom accesses more blocks than the cache size over a fairly long period of time. For this type of workload, the behavior of the other algorithms should be similar to that of LRU so that their hit rates could be close to those of LRU. Before the cache size reaches 350 blocks, the hit rates of LIRS are higher than those of LRU. After that point, the hit rates of LRU become slightly higher. Here is the reason for the slight performance degradation of LIRS beyond that cache size: Whenever there is a locality scope shift or transition, that is, some HIR blocks get referenced, one more miss than would occur in LRU may be experienced by an HIR block. Only the next reference to the block in the near future after the miss makes it switch from HIR to LIR status and then remain in the cache. However, because of the strong locality, there are not frequent locality scope changes. So, the negative effect of the extra misses is limited.

4.2.4 Performance for Mixed Type Workloads

Fig. 6 presents three pairs of time-space maps and the hit rate curves generated by the various replacement algorithms for workloads $multi1$, $multi2$, and $multi3$. The authors in [13] provided a detailed discussion why their UBM shows the best performance among the algorithms they have considered—UBM, LRU-2, 2Q, and EELRU. Here, we focus on performance difference between LIRS and UBM. UBM is a typical spatial regularity detection-based replacement algorithm that makes exhaustive reference pattern detections. UBM tries to identify sequential and looping patterns and applies MRU to the detected patterns. UBM further measures looping intervals and conducts period-based replacements. For those unidentified blocks without special patterns, LRU is applied. A scheme for dynamically allocating buffers among the blocks managed by different algorithms is employed. Without devoting specific efforts to specific regularities, LIRS outperforms UBM for all three mixed type workloads, which indicates that our assumption on IRR holds well and LIRS is able to cope with weak locality in the workloads with mixed type patterns.

4.3 LIRS versus Other Stack-Based Replacements

To get insights into the superiority of LIRS over other stack-based replacement algorithms, including LRU, 2Q, we plot a time-IRR graph to observe their actions on the blocks accessed at different recencies. In a time-IRR graph, the x axis represents virtual time, a reference in the access stream, the y axis represents IRR, the recency where the reference at a virtual time takes place. For first time accessed blocks, their IRRs are infinite, which we do not plot in the graph. We select two representative workloads, a non-LRU-friendly one, $postgres$, and an LRU-friendly one, $sprite$, for this study. Their IRRs are depicted in Fig. 7.

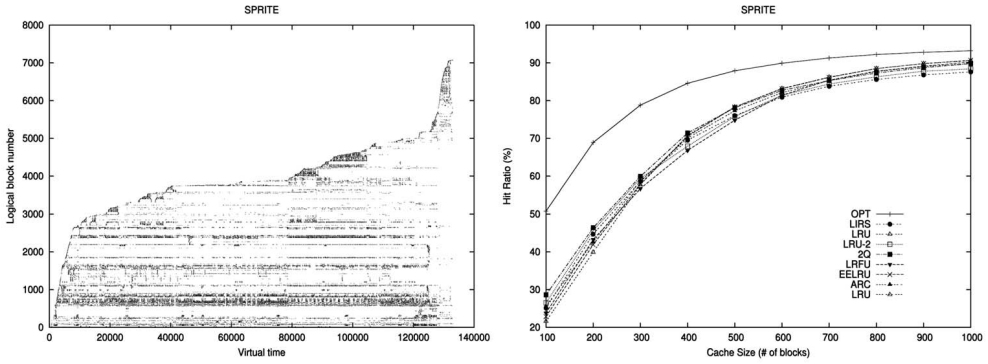

Fig. 5. The time-space map and the hit rate curve of **sprite** for the replacement algorithms.

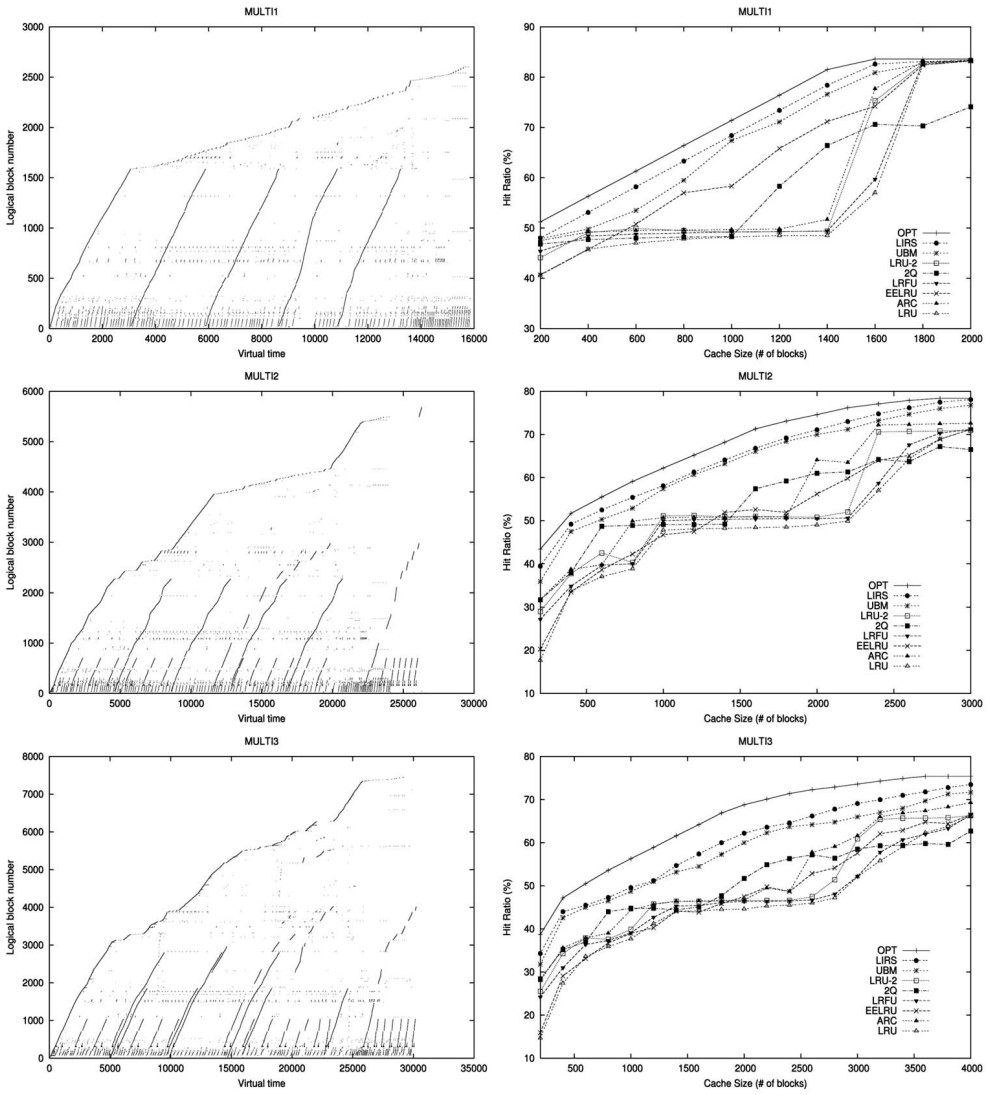

Fig. 6. The time-space maps and the hit rate curves of **multi1**, **multi2**, and **multi3** for the replacement algorithms.

The stack size of LRU, which is determined by the cache size in blocks, is fixed. If the stack size is L, all the references shown in the graphs with their IRRs less than L are hits and those with IRRs larger than L are misses in LRU. Thus, the hit rates of LRU are determined by the IRR distribution. If most of the IRRs are concentrated in the low recency area, such as what is shown in the graph for $sprite$, LRU will get a high hit rate. For workloads with dispersed recency distributions, LRU is incompetent in achieving high hit rates. For example, in $postgres$, there are two IRR concentrations at around IRRs 350, 1150, and 1950. In corresponding to the IRR distribution, there are some apparent "lift ups" in the LRU hit rate curve when the cache size reaches these values (see Fig. 3). If there are a large number of

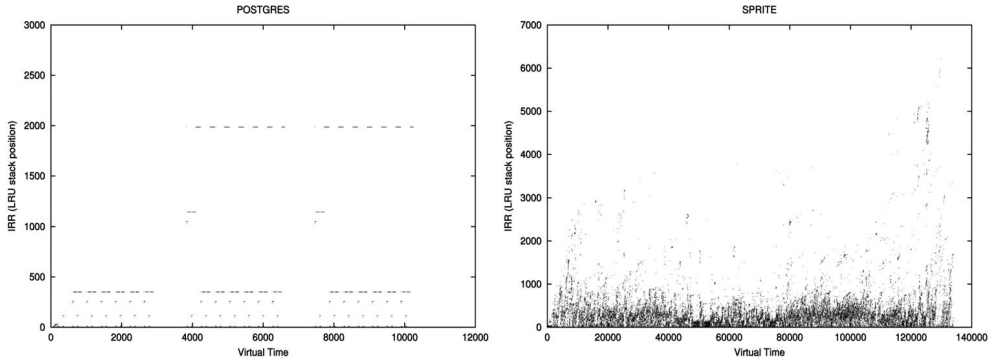

Fig. 7. The IRRs of the references in **postgres** and **sprite**.

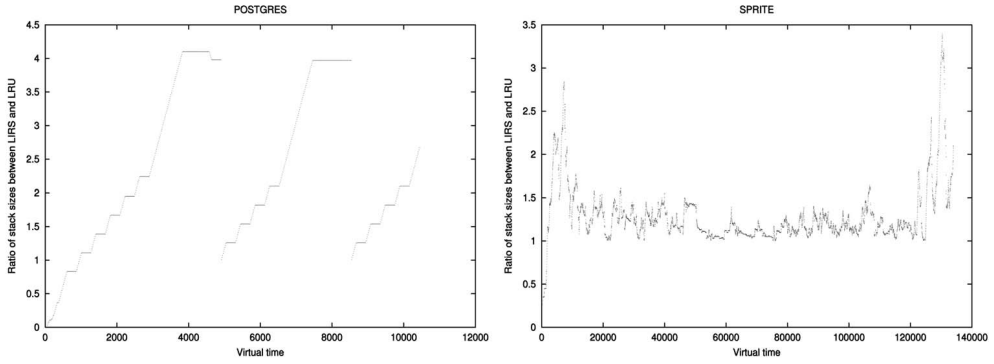

Fig. 8. The ratios of LIRS stack size and LRU stack size for **postgres** and **sprite**. Cache size is 500.

references with their IRRs larger than the LRU stack size, many blocks with their low recencies but high IRRs would hold the stack spaces (residing in the cache) without being accessed before being replaced from the stack. The occupied buffers do not contribute to the hit rate. Thus, what really matters is IRR, not recency. To improve LRU, the criterion to determine which accessed blocks are to be cached should be the L blocks with the smallest IRRs, rather than the L blocks with their recencies no more than L (L is the cache size). Following this criterion, the LIRS algorithm uses the LIRS stack to dynamically predict the L blocks that will have the smallest IRRs. The LIRS stack serves two purposes: 1) providing a threshold for being a LIR block and 2) holding the L blocks with the smallest IRRs (i.e., LIR blocks). In the LIRS algorithm, the threshold is $Rmax$, the recency of the LIR block at the LIRS stack bottom. The threshold is also the LIRS stack size.

4.3.1 The Relationship between LIRS Stack Size and Access Characteristics

To get insights into the relationship of the LIRS stack size and workload access characteristics, we plot the ratio of the LIRS stack size and the LRU stack size for two workloads, $postgres$ and $sprite$, in Fig. 8, where we fix the cache size at 500 blocks. We find that the LIRS stack size is an inherent reflection of the LRU capability to exploit locality. If the references have a strong locality, most of the references are to the blocks with small recencies. Thus, the LRU stack still holds these blocks while they get reaccessed and LRU achieves a high hit rate. At the same time, these blocks are low IRR blocks, i.e., most of the references go to the LIR blocks, which would leave only a small number of HIR blocks in the LIRS stack. So, the LIRS stack size is small and close to the LRU stack size. This is the case for workload $sprite$. With 500 buffer blocks, the LRU stack is able to hold the most frequently referenced blocks. On the other hand, LIRS can find enough low IRR blocks within the recency range covered by the LRU stack. So, there is no need for LIRS to significantly raise its stack size to hold a large number of blocks with high recencies in the cache. This is evidenced in Fig. 8 right, where the ratios of the LIRS and LRU stack sizes are not far from 1 for most of the period of time. However, once LIRS cannot find enough low IRR blocks within the size of the LRU stack, it will raise its size accordingly. We observe that the LIRS stack size of $postgres$ is significantly increased in several phases during the periods when more references go to the blocks with high recencies than to those with low recencies. With a cache size of 500 and a fixed stack size, LRU cannot make the locality distinction among the blocks with high recencies and causes their references to all miss. By increasing the stack size according to the current access characteristics, LIRS can make the distinction among blocks with weak locality and make a decision to replace the blocks with a weak locality. The experiments also hint that the LIRS stack size is a good indicator of the LRU-friendliness of a workload.

The 2Q Replacement algorithm also tries to identify blocks of small IRRs and to hold them in cache. It relies on queue $A1out$ to decide whether a block is qualified to be promoted to stack Am so that it can be cached for a long time or, consequently, to decide whether a block in Am

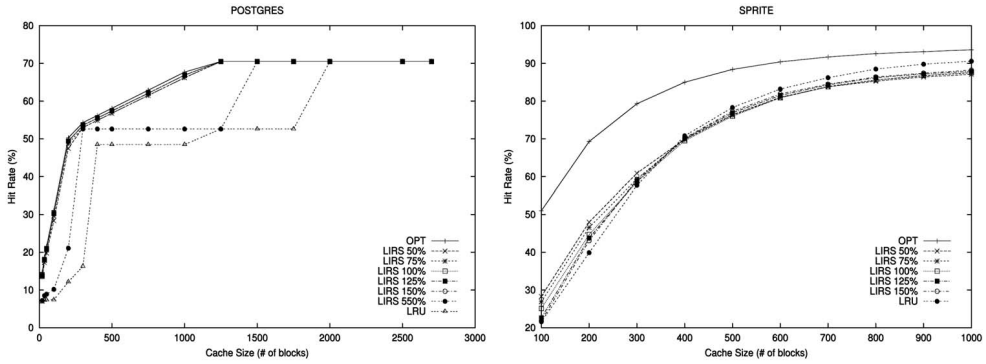

Fig. 9. The hit rate curves of **postgres** and **sprite** by varying the ratio of the status switching threshold and $Rmax$ in LIRS, as well as the curves for OPT and LRU.

should be demoted out of Am. In 2Q, the size of $A1out$ serves as a threshold to identify the blocks of small IRRs and Am holds these blocks. Because the threshold is intended to predict the blocks with the L smallest IRRs among all accessed blocks, 2Q should also consider the access characteristics of blocks in Am. Unfortunately, it does not and only the blocks in $A1out$ are used for setting the threshold. The recommended size of $A1out$ in paper [10] is 50 percent of the cache size. With a fixed threshold, 2Q could make it either too easy or too difficult for the blocks to join in Am with the varying access patterns, This explains why 2Q cannot provide a consistent performance improvement over LRU.

4.3.2 LRU as a Special Member of the LIRS Family

In the LIRS algorithm, the largest recency of the LIR blocks, $Rmax$, serves as a threshold for status switching. An HIR block with a new IRR smaller than the LIRS threshold can change into LIR status and may demote an LIR block into HIR status. The threshold controls how easily an HIR block may become an LIR block or how difficult it is for an LIR block to become an HIR one. We scale the threshold by a weight factor to get insights into the relationship of LRU and LIRS. A weight factor defines a particular LIRS alternative. So, with the scaling, we have a family of LIRS algorithms with various thresholds. Lowering the threshold value, we are able to strengthen the stability of the LIR block set by making it more difficult for HIR blocks to switch their status into LIR. It also prevents the LIRS algorithm from responding to the relatively small IRR variance. Increasing the threshold value, we go in the opposite direction. In this way, LRU becomes a special member of the LIRS family—an LIRS algorithm with an indefinitely large threshold, which always gives any accessed block an LIR status and keeps it in the cache until it is evicted from the stack bottom.

Fig. 9 presents the results of a sensitivity study of the threshold value. We again use workloads $postgres$ and $sprite$ to observe the effects of changing the threshold values from 50 percent, 75 percent, 100 percent, 125 percent to 150 percent of $Rmax$. For $postgres$, we include a very large threshold value—550 percent of $Rmax$ to highlight the relationship between LIRS and LRU. We have two observations. First, LIRS is not sensitive to the threshold value across a large range. In $postgres$, the curves for the threshold values of 100 percent, 125 percent, 150 percent of $Rmax$ are overlapped and the curves for 50 percent, 75 percent of $Rmax$ are slightly lower than the curve for 100 percent of the $Rmax$ threshold. Second, the LIRS algorithm can simulate LRU behavior by significantly increasing the threshold. As the threshold value increases to 550 percent of $Rmax$, the LIRS curve of $postgres$ is very similar to that of LRU in its shape and is close to the LRU curve. Further increasing the threshold value makes the LIRS curve overlaps with the LRU curve. For $sprite$, an LRU-friendly workload, increasing the threshold value makes the LIRS hit rate curve move slowly to the LRU curve.

5 Sensitivity and Overhead Analysis

5.1 Cache Allocation for Resident HIR Blocks

L_{hirs} is the only parameter in the LIRS algorithm. The blocks in the LIR block set can stay in the cache for a longer time than those in the HIR block set and experience fewer page faults. A sufficiently large L_{lirs} (the cache size for LIR blocks) ensures there are a large number of LIR blocks. For this purpose, we set L_{lirs} to be 99 percent of the cache size, L_{hirs} to be 1 percent of the cache size in our experiments, and achieve expected performance. From the other perspective, an increased L_{hirs} may also be beneficial to the performance in some cases: It reduces the first time reference misses. For a large size of stack Q (large L_{hirs}), it is more likely that an HIR will be reaccessed before it is evicted from the stack, which can help the HIR block change into LIR status without experiencing an extra miss. However, the benefit of large L_{hirs} is limited because the number of such kind of misses is small.

We use two workloads, $postgres$ and $sprite$, to observe the effect of changing the size. We change L_{hirs} from two blocks, to 1 percent, 10 percent, 20 percent, and 30 percent of the cache size. Fig. 10 shows the results of the sensitivity study on L_{hirs} for $postgres$ and $sprite$. For each workload, we measure the hit rates of OPT, LRU, and LIRS with different L_{hirs} sizes with increasing cache sizes. We have two observations. First, for both workloads, we find that LIRS is not sensitive to the increase of L_{hirs}. Even for a very large L_{hirs}, which is not in favor of LIRS, the performance of LIRS with different cache sizes is still acceptable. With the increase of L_{hirs}, the hit rates

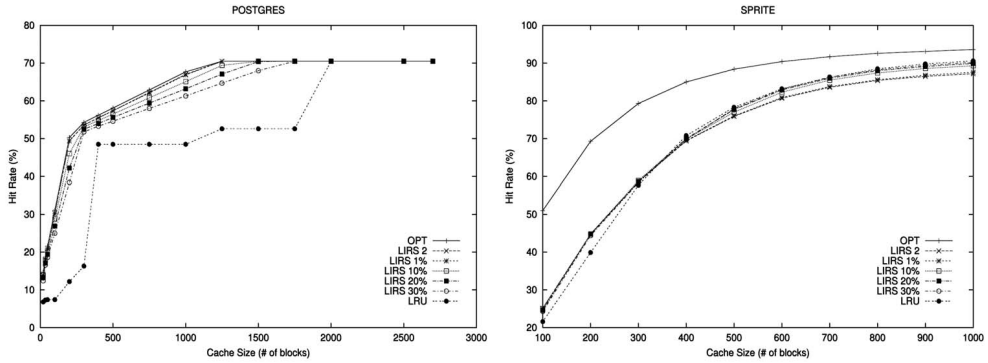

Fig. 10. The hit rate curves of **postgres** and **sprite** by varying the size of stack Q (L_{hirs}) of the LIRS algorithm, as well as the curves for OPT and LRU. "LIRS 2" means the size of Q is 2, "LIRS x%" means the size of Q is x percent of the cache size in blocks.

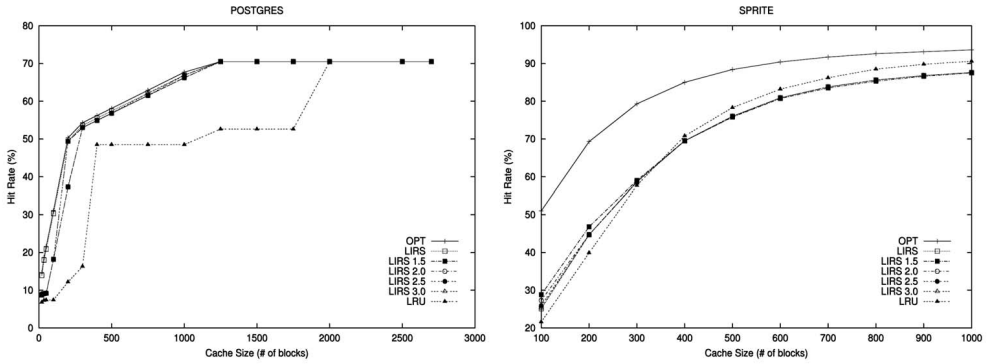

Fig. 11. The hit rate curves of **postgres** and **sprite** by varying the LIRS stack size limit, as well as the curves for OPT and LRU. Limits are represented by ratios of LIRS stack size limit and cache size in blocks.

of LIRS approach those of LRU. Second, our experiments indicate that increasing L_{hirs} reduces the performance benefits of LIRS to workload $postgres$, but slightly improves performance of workload $sprite$.

5.2 Overhead Analysis

LRU is known for its simplicity and efficiency. Comparing the time and space overhead of LIRS and LRU, we show that LIRS keeps the LRU merit of low overhead. The time overhead of LIRS algorithm is $O(1)$, which is almost the same as LRU with a few additional operations such as those on stack Q for resident HIR blocks. The extended portion of the LIRS stack S is the additional space overhead of the LIRS algorithm.

The stack S contains metadata for the blocks with their recency less than $Rmax$. When there is a burst of first-time block references, the LIRS stack could grow to be unacceptably large. Imposing a size limit is a practical issue in the implementation of the LIRS algorithm. In an updated version of LIRS, the LIRS stack has a size limit that is larger than L, and we remove the HIR blocks close to the bottom from the stack once the LIRS stack size exceeds the limit. We have tested a range of small stack size limits, from 1.5 times to 3.0 times of L. From Fig. 11, we can observe that, even with these strict space restrictions, LIRS retains its desirable performance. The effect of limiting LIRS stack size is equivalent to reducing the threshold values in Section 4.3.2. As expected, the results are consistent with the ones presented there. In addition, since a stack entry consists of only several bytes, it is easily affordable to have an LIRS stack size limit much more than three times LRU stack size. There would be little negative effect on LIRS performance by enforcing the limit of such a large size.

6 Conclusions

Replacement algorithms play important roles in the buffer cache management and their effectiveness and efficiency are crucial to the performance of file systems, databases, and other data management systems. We make two contributions in this paper by proposing the LIRS algorithm: 1) We show that LRU limitations with weak locality workloads can be successfully addressed without relying on the explicit access pattern detections. 2) We show earlier work on improving LRU such as LRU-K or 2Q can evolve into one algorithm with consistently superior performance, without tuning or adaptation of sensitive parameters. The effort of these algorithms, which only trace their own history information of each referenced block, is promising to produce an algorithm that is simple and low overhead yet effective for weak locality access patterns. We have shown the LIRS algorithm accomplishes this goal.

As a general-purpose replacement algorithm, the LIRS algorithm also has its potential to be applied in the virtual memory management for its simplicity and its LRU-like assumption on workload characteristics. Because virtual memory system cannot afford an overhead proportional to the number of memory accesses, neither LRU nor LIRS can be directly used there. We have designed an LIRS approximation, called CLOCK-Pro, with a reduced overhead comparable to that of the CLOCK replacement policy

[12]. The results of an implementation of the LIRS approximation in a Linux kernel have shown its significant performance advantages in terms of hit rates and program run times.

ACKNOWLEDGMENTS

This work is supported in part by the US National Science Foundation under grants CCR-9812187 and CCR-0098055. The authors are grateful to Dr. Sam H. Noh at Hong-IK University, Drs. Jong Min Kim, Donghee Lee, and Jongmoo Choi at the Seoul National University for providing us with their traces and simulators. They are also grateful to Dr. Scott Kaplan at Amherst College and Dr. Yannis Smaragdakis at the Georgia Institute of Technology, who provided them with the latest version of their EELRU simulator and traces. The preliminary results of this work were presented in [11].

REFERENCES

[1] L.A. Belady, R.A. Nelson, and G.S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," *Comm. ACM,* vol. 12, pp. 349-353, 1969.
[2] E.G. Coffman and P.J. Denning, *Operating Systems Theory.* Prentice-Hall, 1973.
[3] P. Cao, E.W. Felten, and K. Li, "Application-Controlled File Caching Policies," *Proc. USENIX Summer 1994 Technical Conf.,* pp. 171-182, June 1994.
[4] J. Choi, S. Noh, S. Min, and Y. Cho, "Towards Application/File-Level Characterization of Block References: A Case for Fine-Grained Buffer Management," *Proc. 2000 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 286-295, June 2000.
[5] J. Choi, S. Noh, S. Min, and Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme," *Proc. 1999 Ann. USENIX Technical Conf.,* pp. 239-252, June 1999.
[6] C. Ding and Y. Zhong, "Predicting Whole-Program Locality through Reuse-Distance Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 245-257, June 2003.
[7] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Trans. Database Systems,* pp. 560-595, Dec. 1984.
[8] C. Gniady, A.R. Butt, and Y.C. Hu, "Program Counter Based Pattern Classification in Buffer Caching," *Proc. Sixth Symp. Operating Systems Design and Implementation,* pp. 395-408, Dec. 2004.
[9] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *Proc. 1997 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 115-126, May 1997.
[10] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th Int'l Conf. Very Large Data Bases,* pp. 439-450, Sept. 1994.
[11] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. 2002 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 31-42, June 2002.
[12] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," *Proc. 2005 Ann. USENIX Technical Conf.,* pp. 323-336, Apr. 2005.
[13] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, "A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," *Proc. Fourth Symp. Operating System Design and Implementation,* pp. 119-134, Oct. 2000.
[14] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. 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 Conf. Measuring and Modeling of Computer Systems,* pp. 134-143, May 1999.
[15] T.C. Mowry, A.K. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Application," *Proc. Second USENIX Symp. Operating Systems Design and Implementation,* pp. 3-17, Oct. 1996.
[16] N. Megiddo and D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proc. Second USENIX Conf. File and Storage Technologies,* pp. 115-130, Mar. 2003.
[17] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proc. 1993 ACM SIGMOD Int'l Conf. Management of Data,* pp. 297-306, May 1993.
[18] V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," *Proc. 1995 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 291-300, May 1995.
[19] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc. 15th Symp. Operating System Principles,* pp. 1-16, Dec. 1995.
[20] J.T. Robinson and N.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. 1990 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 134-142, May 1990.
[21] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proc. Usenix Winter 1993 Technical Conf.,* pp. 405-420, Jan. 1993.
[22] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," *Proc. 1999 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 122-133, May 1999.
[23] Y. Zhou, J.F. Philbin, and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," *Proc. 2001 Ann. USENIX Technical Conf.,* pp. 91-104, June 2001.

Song Jiang received the BS and MS degrees in computer science from the University of Science and Technology of China in 1993 and 1996, respectively, and received the PhD degree in computer science from the College of William and Mary in 2004. He is a postdoctoral research associate at the Los Alamos National Laboratory, developing next generation operating systems for high-end systems. He received the S. Park Graduate Research Award from the College of William and Mary in 2003. His research interests are in the areas of operating systems, computer architecture, and distributed systems.

Xiaodong Zhang received the BS degree in electrical engineering from Beijing Polytechnic University in 1982 and the MS and PhD degrees in computer science from the University of Colorado at Boulder in 1985 and 1989, respectively. He is the Lettie Pate Evans Professor of computer science and the department chair at the College of William and Mary. He was the program director of Advanced Computational Research at the US National Science Foundation from 2001 to 2003. He is a past editorial board member of the *IEEE Transactions on Parallel and Distributed Systems* and currently serves as an editorial board member for the *IEEE Transactions on Computers* and an associate editor of *IEEE Micro*. His research interests are in the areas of parallel and distributed computing and systems and computer architecture. He is a senior member of the IEEE.