LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance *

Song Jiang

Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795
sjiang@cs.wm.edu

Xiaodong Zhang

Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795
zhang@cs.wm.edu

ABSTRACT

Although LRU replacement policy has been commonly used in the buffer cache management, it is well known for its inability to cope with access patterns with weak locality. Previous work, such as LRU-K and 2Q, attempts to enhance LRU capacity by making use of additional history information of previous block references other than only the recency information used in LRU. These algorithms greatly increase complexity and/or can not consistently provide performance improvement. Many recently proposed policies, such as UBM and SEQ, improve replacement performance by exploiting access regularities in references. They only address LRU problems on certain specific and well-defined cases such as access patterns like sequences and loops. Motivated by the limits of previous studies, we propose an efficient buffer cache replacement policy, called *Low Inter-reference Recency Set* (LIRS). LIRS effectively addresses the limits of LRU by using recency to evaluate Inter-Reference Recency (IRR) for making a replacement decision. This is in contrast to what LRU does: directly using recency to predict next reference timing. At the same time, LIRS almost retains the same simple assumption of LRU to predict future access behavior of blocks. Our objectives are to effectively address the limits of LRU for a general purpose, to retain the low overhead merit of LRU, and to outperform those replacement policies relying on the access regularity detections. Conducting simulations with a variety of traces and 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 LRU.

*This work is supported in part by the U.S. National Science Foundation under grants CCR-9812187, EIA-9977030, and CCR-0098055.

1. INTRODUCTION

1.1 The Problems of LRU Replacement Policy

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 to manage 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 frequencies. Here are some representative examples reported in the research literature, to illustrate how LRU poorly behaves.

1. Under the LRU policy, a burst of references to infrequently used blocks, such as "sequential scans" through a large file, may cause replacement of commonly referenced blocks in the cache. This is a common complaint in many commercial systems: sequential scans can cause interactive response time to deteriorate noticeably [14]. A wise replacement policy should 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 soonest, because these blocks have not been accessed for the longest time [18]. A wise replacement policy should maintain a miss rate close to the buffer space shortage ratio.

3. In an example of multi-user database application [14], each record is associated with a B-tree index. There are 20,000 records. The index entries can be packed into 100 blocks, and 10,000 blocks are needed to hold records. We use $R(i)$ to represent an access to Record i, and $I(i)$ to Index i. The access pattern of the database application alternates references to random index blocks and record blocks by $I(1)$, $R(1)$, $I(2)$, $R(2)$, $I(3)$, $R(3)$, ... Thus, index blocks will be referenced with a probability of 0.005, and data blocks are with a probability of 0.00005. However, LRU will

keep an equal number of index and record blocks in the cache, and perhaps even more record blocks than index blocks. A wise replacement should select the resident blocks according to the reference probabilities of the blocks. Only those blocks with relatively high probabilities deserve to stay in the cache for a long period of time.

The reason for LRU to behave poorly in these situations is that LRU makes a bold assumption – a block that has not been accessed the longest would wait for relatively longest time to be accessed again. This assumption cannot capture the access patterns exhibited in these workloads with weak locality. Generally speaking, there is less locality in buffer caches than that in CPU caches or virtual memory systems [17].

However, LRU has its distinctive merits: simplicity and adaptability. It only samples and makes use of very limited information – recency. However, while addressing the weakness of LRU, existing policies either take more history information into consideration, such as LFU (Least Frequency Least)-like ones in the cost of simplicity and adaptability, or switch temporarily from LRU to other policies whenever regularities are detected. In the switch-based approach, these policies actually act as supplements of LRU in a case-by-case fashion. To make a prediction, these policies assume the existence of relationship between the future reference of a block with the behaviors of those blocks in its temporal or spatial locality scope, while LRU only associates the future behavior of a block with its own history references. This additional assumption increases the complexity of implementations, as well as their performance dependence on the specific characteristics of workloads. In contrast, our LIRS essentially only samples and makes use of the same history information as LRU does – recency, and almost retains the simple assumption of LRU. Thus it is simple and adaptive. In our design, LIRS is not directly targeted at specific LRU problems but fundamentally addresses the limits of LRU.

1.2 An Executive Summary of Our Policy

We use recent Inter-Reference Recency (IRR) as the recorded history information of each block, where IRR of a block refers to the number of other blocks accessed between two consecutive references to the block. Specifically, the recency refers to the number of other blocks accessed from last reference to the current time. We call IRR between last and penultimate (second-to-last) references of a block as recent IRR, and 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 again. Following this assumption, we select the blocks with large IRRs for replacement, because these blocks are highly possible to be evicted later by LRU before being referenced again under our assumption. It is noted that these evicted blocks may also have been recently accessed, i.e. each has a small recency.

In comparison with LRU, by adequately considering IRR in history information in our policy, we are able to eliminate negative effects caused by only considering recency, such as the problems presented in the above three examples. When deciding which block to evict, our policy utilizes the IRR information of blocks. 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 recencies of blocks are only used to help determine LIR or HIR statuses of 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 in the set can fit in the cache. The blocks in the LIR set are not chosen for replacement, and there are no misses with references to these blocks. Only a very small portion of the cache is assigned 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 point, 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 policy "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. The LIRS policy aims at addressing three issues in designing replacement policies: (1) how to effectively utilize multiple sources of access information; (2) how to dynamically and responsively distinguish blocks by comparing their possibilities to be referenced in the near future; and (3) how to minimize implementation overhead.

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

2. RELATED WORK

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

2.1 User-level Hints

Application-controlled file caching [3] and application-informed prefetching and caching [16] are the schemes based on user-level hints. These schemes identify blocks with low possibility to be accessed in the near future based on available hints provided by users. To provide appropriate hints, users need to understand the data access patterns, which adds to the programming burden. In [13], Mowry et. al. attempt to abstract hints by compilers to facilitate I/O prefetching. Although their methods are orthogonal to our LIRS replacement, the collected hints may help us ensure the existence of the correlation of consecutive IRRs. However, in most cases, the LIRS algorithm can adapt its behavior to different access patterns without explicit hints.

2.2 Tracing and Utilizing History Information

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

Robinson and Devarakonda propose a frequency-based replacement algorithm (FBR) by maintaining reference counts for the purpose to "factor out" locality [17]. However it is slow to react to reference popularity changes and some parameters have to be found by trial and error. Having analyzed the advantages and disadvantages of LRU and LFU, Lee et, al. combine them by weighing recency factor and frequency factor of a block [12]. The performance of the LRFU scheme largely depends on a parameter called λ, which decides the weight of LRU or LFU, and which has to be adjusted according to different system configurations, even according to different workloads.

The LRU-K scheme [14] addresses the LRU problems presented in the Examples 1 and 3 in the previous section. LRU-K makes its replacement decision based on the time of the Kth-to-last reference to the block. After such a comparison, the oldest resident block is evicted. For simplicity, the authors recommended K = 2. By taking the time of the penultimate reference to a block as the basis for comparisons, 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, the overhead of 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 propose the 2Q scheme that has overhead of a constant time [10]. The authors claim that the scheme performs as well as LRU-2. The 2Q scheme can quickly remove sequentially-referenced blocks and cyclically-referenced blocks with long intervals from the cache. This is done by using a special buffer, called the *A1in queue*, in which all missed blocks are initially placed. When the blocks are replaced from the *A1in queue* in the FIFO order, the addresses of those replaced blocks are temporarily placed in a ghost buffer called *A1out queue*. When a block is re-referenced, if its address is in the *A1out queue*, it is promoted to a main buffer called *Am*, where frequently accessed blocks are stored. In this way they are able to distinguish between frequently and infrequently referenced blocks. By setting thresholds *Kin* and *Kout* for the sizes of *A1in* and *A1out*, respectively, 2Q provides a victim block either from *A1in* or *Am*. However, *Kin* and *Kout* are pre-determined parameters in 2Q, which need to be carefully tuned, and are sensitive to the types of workloads. Although both 2Q and LIRS have implementations with low overheads, our algorithm has overcome the drawbacks of 2Q by a properly updating of the LIR block set.

Inter-Reference Gap (IRG) for 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 consider the correlation between history IRGs and future IRG [15]. The past IRG string for each block is modeled by Markov chains to predict the next IRG. However, just as Smaragdakis et. al. indicate, replacement algorithms based on Markov models are not practical because they try to solve a much harder problem than the replacement problem itself [18]. An apparent difference in their scheme from our LIRS algorithm is on 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 an IRR is more relevant to the next IRR than an IRG to the next IRG. Moreover, it is the "recency" but not "gap" information that is used by LRU. An elaborate argument favoring IRR in the context of virtual memory page replacement can be found in [18]. Secondly, 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, researchers take 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 MRU, can be applied to the blocks with specific access regularities.

Glass and Cao propose adaptive replacement SEQ for page replacement in virtual memory management [9]. It detects sequential address reference patterns. If long sequences of page faults are found, MRU is applied to such sequences. If no sequences are detected, SEQ performs LRU replacement. Smaragdakis et. al. argue that the address-based detection lacks generality, and advocate using recency information to distinguish between pages [18]. Their EELRU examines aggregate recency distributions of referenced pages and changes the page eviction points using an on-line cost/benefit analysis by assuming the correlation among temporally contiguously referenced pages. With an aggregate analysis, EELRU can not quickly respond to the changing access patterns. Without spatial or temporal detections, our LIRS uses independent recency events of each block to effectively characterize their references.

Choi et. al. propose a new adaptive buffer management scheme called DEAR that automatically detects the block reference patterns of applications and applies different replacement policies to different applications based on the detected reference patterns [5]. Further, they propose an Application/File-level Characterization (AFC) scheme in [4]. The Unified Buffer Management (UBM) scheme by Kim et. al. also detects patterns in the recorded history [11]. Though their elaborate detections of block access patterns provide a large potential to high performance, they address the problems in a case-by-case fashion and have to cope with the allocation problem, which does not appear in LRU. To facilitate the on-line evaluation of buffer usage, certain pre-measurements are needed to set some pre-defined parameters used in the buffer allocation scheme [4, 5]. Our LIRS does not have the design challenge. Just like LRU, it chooses the victim block in the global stack. However, it can use the advantages provided by the detection-based schemes.

2.4 Working Set Models

Lastly, we would like to compare our work with the working set model, an early work by Denning [6]. A working set of a program is a set of its recently used pages. Specifically, at virtual time t, a program's working set $W_t(\theta)$ is the subset of all pages of the program, which have been referenced in the previous θ virtual time units (the working set window). A working set replacement algorithm is used to ensure that no pages in the working set of a running program are replaced [7]. Estimating the current memory demand of a running program in the system, the model does not incorporate the available cache size. When the working set is greater than the cache size, the working set replacement algorithm would not work properly. Another difficulty with the working set model is its weak ability to distinguish recently referenced "cold" blocks from "hot" blocks. Our LIRS algorithm ensures that LIR block set size is less than the available cache size and keeps the set in the cache. IRR helps to distinguish the "cold" blocks from "hot" ones: a recently referenced "cold" block could have a small recency, but would have a large IRR.

3. THE LIRS ALGORITHM

3.1 General Idea

We divide the referenced blocks into two sets: High Inter-reference Recency (HIR) block set and Low Inter-reference Recency (LIR) block set. Each block with history information in cache has a status – either LIR or HIR. Some HIR blocks may not reside in the cache, but have entries in the cache recording their status as HIR of non-residence. We also divide the cache, whose size in blocks is L, into a major part and a minor part in terms of the size. The major part with the size of L_{lirs} is used to store LIR blocks, and the minor part with the 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 free block is needed for replacement, we choose an HIR block that is resident in the cache. LIR block set always resides in the cache and there are no misses for the references to LIR blocks. However, a reference to an HIR block would likely to encounter a miss, because L_{hirs} is very small (its practical size can be as small as 1% of the cache size).

Table 1 gives a simple example to illustrate how a replaced block is selected by the LIRS algorithm and how LIR/HIR statuses are switched. In Table 1, symbol "X" represents that a block is accessed at a virtual time unit [1]. For example, block A is accessed at time units 1, 6, and 8. Based on the definition of recency and IRR in Section 1.2, at time unit 10, blocks A, B, C, D, E have their recency values of 1, 3, 4, 2, and 0, respectively, and have their IRR values of 1, 1, "infinite", 3, and "infinite", respectively. We assume $L_{lirs} = 2$ and $L_{hirs} = 1$, and at time 10 the LIRS algorithm leaves two blocks in LIR set = {A, B}. The rest of the blocks go to 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 just leave 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.

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

The key to successfully make the LIRS idea work in practice rests on whether we can dynamically and responsively maintain the LIR block set and HIR block set. When an HIR block is referenced at a recency, it gets a new IRR, which is equal to its recency. Then we determine whether the newly born IRR is small enough compared with some current reference statistics of existing LIR blocks, so that we can decide whether we need to change its status to LIR. Here we have two options: to compare it either with the IRRs or recencies of the LIR blocks. We choose the recency information of LIR blocks for the comparison. There are two reasons for this: (1) The IRRs are generated before their respective recencies, and they are outdated. So they are 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 the recent activities of other blocks. The result of the comparisons between the new IRR and recencies of the LIR blocks determines the eligibility of the HIR block to be considered as a "hot block". Though we claim that IRRs are used to determine which block should be replaced, it is the newly born 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 coming IRR of the LIR block. This is because the recency of the LIR block is a portion of its coming 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 LIR block set in a timely way by removing an LIR block from the set; (2) keep the size of LIR block set no larger than L_{lirs}, thus the entire set can reside in the cache.

Again in the example of Table 1, if there is a reference to block D at time 10, then a miss occurs. LIRS algorithm evicts resident HIR block E, instead of block B, which would be evicted by LRU due to its largest recency. Furthermore, because block D is referenced, its new IRR becomes 2, which is smaller than the recency of LIR block B (=3), indicating that coming 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 4, rather than block D with its recency 2, gets referenced, there will be no status switching. Then block C becomes a resident HIR block, though the replaced block is still E at virtual time 10. The LIR block set and HIR block set are formed and dynamically maintained in this way.

3.2 An Implementation Using the 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 is a cache storage containing L entries, each of which represents a block[2]. In practice, L is the cache size in blocks. LIRS algorithm makes use of the stack to record

[2]For simplicity, in the rest of the paper we just say without ambiguity "a block in the stack" instead of "the entry of a block in the stack" .

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

Table 1: An example to explain how a victim block is selected by the LIRS algorithm and how LIR/HIR statuses are switched. An "X" means the block of the row is referenced at the virtual time of the column. The recency and IRR columns represent the corresponding values at virtual time 10 for each block. We assume $L_{lirs} = 2$ and $L_{hirs} = 1$. At time 10 LIR set = {A, B}, HIR set = {C, D, E}, and the only resident HIR block is E.

the recency, and to dynamically maintain the LIR block set and HIR block set. Not adopting the LRU stack, where only resident blocks are managed by LRU replacement in the stack, we store LIR blocks, and HIR blocks with their recency less than the maximum recency of LIR blocks in a stack called LIRS stack S, which follows the LRU stack operation principle but has a varying size. With this implementation, we are able to remove the possible burden of explicitly keeping track of the IRR and recency values and searching the maximum recency value. Each entry in the stack records the LIR/HIR status and residence status indicating whether or not the block resides in the cache. To facilitate the search of resident HIR blocks, we link all these blocks into a small list Q with its maximum size L_{hirs}. Once a free block is needed, the LIRS algorithm removes a resident HIR block from the front of the list for replacement. However, the replaced HIR block remains in the stack S with its residence status changed to non-residence, if it is originally in the stack. We ensure the block in the bottom of the stack S is an LIR block by removing HIR blocks after it. Once an HIR block in the LIRS stack gets referenced, which means there is at least one LIR block, such as the one in the bottom, whose coming IRR will be greater than the new IRR of the HIR block, we switch the LIR/HIR statuses of the two blocks. The LIR block in the bottom is evicted from the stack S and goes to the end of the list Q as a resident HIR block. This block is going to be replaced out of the cache soon due to the small size of the list Q (at most L_{hirs}).

Such a scheme is also intuitive from the perspective of LRU replacement behavior: if a block gets evicted in the bottom of LRU stack, it means it holds a buffer during the period of time when it moves from the top to the bottom of the stack without being referenced. Why should we afford a buffer for another long idle period when the block is loaded again into the cache? The rationale behind this is the assumption that temporal IRR locality holds for block references.

3.3 A Detailed Implementation Description

We define an operation called "stack pruning" on the LIRS stack S, which removes the HIR blocks in the bottom of the stack until an LIR block sits in the stack bottom. This operation serves for two purposes: (1) We ensure the block in the bottom of the stack 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 chances to change their status from HIR to LIR, because their recencies are larger than the new maximum recency of LIR blocks.

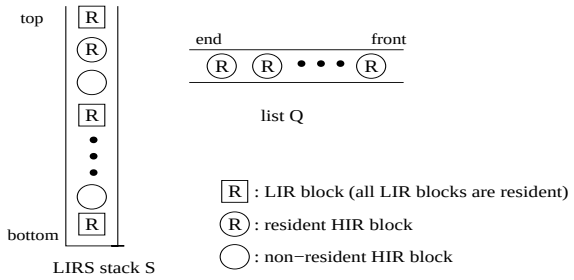

Figure 1: The LIRS stack S holds LIR blocks as well as HIRS blocks with or without resident status, and a list Q holds all the resident HIR blocks.

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

Figure 1 describes a scenario where stack S holds three kinds of blocks: LIR blocks, resident HIR blocks, non-resident HIR blocks, and a list Q holds all of the resident HIR blocks. Each HIR block could either be in stack S or not. Figure 1 does not depict non-resident HIR blocks that are not in stack S. There are three cases with various references to these 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 in the bottom of the stack, we conduct a stack pruning.

2. **Upon accessing an HIR resident block X:** This is a hit in the cache. We move it to the top of stack S. There are two cases for block X: (1) If X is in the stack S, we change its status to LIR. This block is also removed from list Q. The LIR block in the bottom of S is moved to the end of list Q with its status changed to HIR. A stack pruning is then conducted. (2) If X is not in stack S, we leave its status in HIR and move it to the end of list Q.

3. **Upon accessing an HIR non-resident block X:** This is a miss. We remove the HIR resident block at the front of list Q (it then becomes a non-resident block), and replace it out of the cache. Then we load the requested block X into the freed buffer and place it on the top of stack S. There are two cases for block X: (1) If X is in stack S, we change its status to LIR and move the LIR block in the bottom of stack S to the end of list Q with its status changed to HIR. A stack pruning is then conducted. (2) If X is not in stack S, we leave its status in HIR and place it in the end of list Q.

4. PERFORMANCE EVALUATION

4.1 Experimental Settings

To validate our LIRS algorithm and to demonstrate its strength, we use trace-driven simulations with various types of workloads to evaluate and compare it with other algorithms. We have adopted many application workload traces used in previous literature aiming at addressing the limits of LRU. 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 [12], multi1, multi2, multi3 are used in [11]. We briefly describe the workload traces here.

1. **2-pools** is a synthetic trace, which simulates application behavior of the example 3 in Section 1.1 with 100,000 references.
2. **cpp** is a GNU C compiler pre-processor 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 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.

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

- Sequential references: all blocks are accessed one after another, and never re-accessed;
- 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 the associated probabilities.

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

Policies LRU, LRU-2, 2Q, and LRFU belong to the same category of replacement policies as LIRS. In other words, these policies take the same technical direction — predicting the access possibility of a block through its own history access information. Thus, we focus our performance comparisons between ours and these policies. As representative policies in the category of regularity detections, we choose two schemes for comparisons: UBM for its spatial regularity detection, and EELRU for its temporal regularity detection. UBM simulation asks for file IDs, offsets, and process IDs of a reference. However, some traces available to us only consists of logical block numbers, which are unique numbers for accessed blocks. Thus, we only include the UBM experimental results for the traces used in paper [11], which are multi1, multi2, multi3. We also include the results of OPT, an optimal, off-line replacement algorithm [2] for comparisons.

4.2 Performance Evaluation Results

We divide the 9 traces into 4 groups based on their access patterns. Traces cs, postgres, and glimpse 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. For the policies with pre-determined parameters, we used the parameters presented in their related papers. The only parameter of the LIRS algorithm, L_{hirs}, is set as 1% of the cache size, or $L_{lirs} = 99\%$ of the cache size. This selection results from a sensitivity analysis to L_{hirs}, which is described in Section 5.1. Figure 2 shows hit rates for each workload as the cache size increases for various replacement policies.

4.2.1 Replacement Performance on Looping Patterns

Traces cs, glimpse, and postgres have looping patterns with long intervals. As expected, LRU performs poorly for these workloads with the lowest hit rates among the policies. Let us take cs as an example, which has a pure looping pattern. Each of its blocks is accessed almost with the same interval.

Since all blocks with looping accesses 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 just what LIRS does: the same LIR blocks are fixed in the cache because HIR blocks do not have IRRs small enough to change their status. In the looping pattern, recency predicts the opposite of the future reference time of a block: the larger the recency of a block is, the sooner the block will be re-referenced. The hit rate of LRU for cs is almost 0% until the cache size approaches 1,400 blocks, which can hold all the blocks referenced in the loop. It is interesting to see that the hit rate curve of LRU-2 overlaps with the LRU curve. This is because LRU-2 chooses the same victim block as the one chosen by LRU for replacement. When making a decision, LRU-2 compares the penultimate reference time, which is the recency plus the recent IRG. However, the IRGs are of the same value 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 penultimate reference time, LRU-2 behaves similarly to LRU.

Except for cs, the other two workloads have mixed looping patterns with different intervals. LRU presents stair-step curves to increase the hit rates for those 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 of postgres have a sharp increase from 16.3% to 48.5%. Because LRU-2 considers the last IRG in addition to the recency, it is easier for it to distinguish blocks in the loops with different intervals than LRU does. However, LRU-2 lacks the capability to deal with these blocks when varying recency is involved. Our experiments show that the achieved performance improvements by LRU-2 over LRU are 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 re-referenced blocks deserve to be kept in the cache for a long period of time. We can see that the hit rates of 2Q are significantly lower than those of LIRS for all the three workloads. As the cache size increases, 2Q even performs worse than LRU for workloads glimpse and postgres. Another observation of 2Q on glimpse and postgres is a serious "Belady's anomaly"[1]: increasing the cache size may increase the number of misses. The reason is as follows: 2Q relies on queue $A1out$ to decide whether a block is qualified to promote to stack Am so that it can stay in the cache for long time, or consequently to decide whether a block in Am should be demoted out of Am. Actually if the blocks in Am are relatively frequently referenced, it should be hard for the blocks out of Am to join Am. Otherwise, it should become easy for these blocks to join Am. On the other hand, the larger $A1out$ is, the more easily a block is promoted into Am; and the smaller, the more difficult. So $A1out$ should vary its size threshold $Kout$ dynamically reflecting the relative frequencies in the access patterns of blocks in and out of Am. However, in 2Q it is a fixed pre-determined parameter. LIRS policy can be viewed to integrate $A1in$, $A1out$ and Am into an LIRS stack, whose size varies according to the current access patterns. We provide an effective criterion – only when an HIR block gets accessed and generates a new IRR less than the recency of an LIR block, do we promote it to be an LIR block and demote an LIR block to be an HIR one.

LRFU, which combines LRU and LFU, is not effective on a workload with a looping pattern. The LRFU and LRU hit rate curves for workload cs are overlapped.

Our trace-driven simulation results show LIRS significantly outperforms all of the other policies, and its hit rate curves are very close to that of OPT. LIRS can make a more accurate prediction on the future LIR/HIR status of each block for cs and postgres than glimpse, because the intervals of loops in cs and postgres are of less variance, and the consecutive IRRs are of less variance (See the performance difference between cs, postgres and glimpse in Figure 2). However, the LIRS algorithm is not sensitive to the variance of IRRs, which is reflected by its good performance on workload glimpse. We explain it as follows.

We denote the recency of the LIR block in the bottom of LIRS stack S as $Rmax$. When there are no free block buffers, $Rmax$ is larger than the cache size in blocks. Only when the two consecutive IRRs of references to a block vary across value $Rmax$, is the status prediction of the LIRS algorithm based on the last IRR wrong, including two cases: (1) an IRR less than $Rmax$ is succeeded by another IRR greater than $Rmax$, and (2) an IRR greater than $Rmax$ is succeeded by another IRR less than $Rmax$. All other IRR variances, no matter how much they are, would impose no mishandling of the LIRS replacement. Let us take a close look at the penalty from a wrong LIR/HIR status decision: (1) If a block is mistakenly labeled as LIR due to previous small IRR, LIRS will evict the block in the bottom of the stack without being referenced. However, if it would be in its status HIR, it was evicted much earlier after being loaded into the cache to produce a free block. Keeping a block in the cache until it is evicted in the bottom is just what LRU does to every block. Thus such a mis-labeled block incurs a comparable performance loss with that of LRU. (2) If a block is mistakenly labeled as HIR due to previous large IRR, LIRS will evict the block far before its reaching the bottom. However, if it would be in its status LIR, it was referenced before being evicted from the cache. Thus LIRS would incur an extra miss if the block had been evicted from HIR resident list Q. However, because the number of block buffers assigned to list Q (L_{hirs}) is very small, which is only 1% of total cache size in our experiments, HIR blocks would be replaced very soon, which reduces the chance for the replaced block to be re-referenced shortly after its eviction. The free block buffer for the period between the early eviction and its next reference helps to reduce the penalty from the extra misses.

4.2.2 Replacement Performance on Probabilistic Patterns

According to the detection results in [4], workload cpp exhibits probabilistic reference pattern. The cpp hit rate in Figure 2 shows that before the cache size increases to 100 blocks, the hit rate of LRU is much lower than that of LIRS for cpp. For example, when the cache size is 50 blocks, hit rate of LRU is 9.3%, while hit rate of LIRS is 55.0%. This is because holding a major reference locality needs about

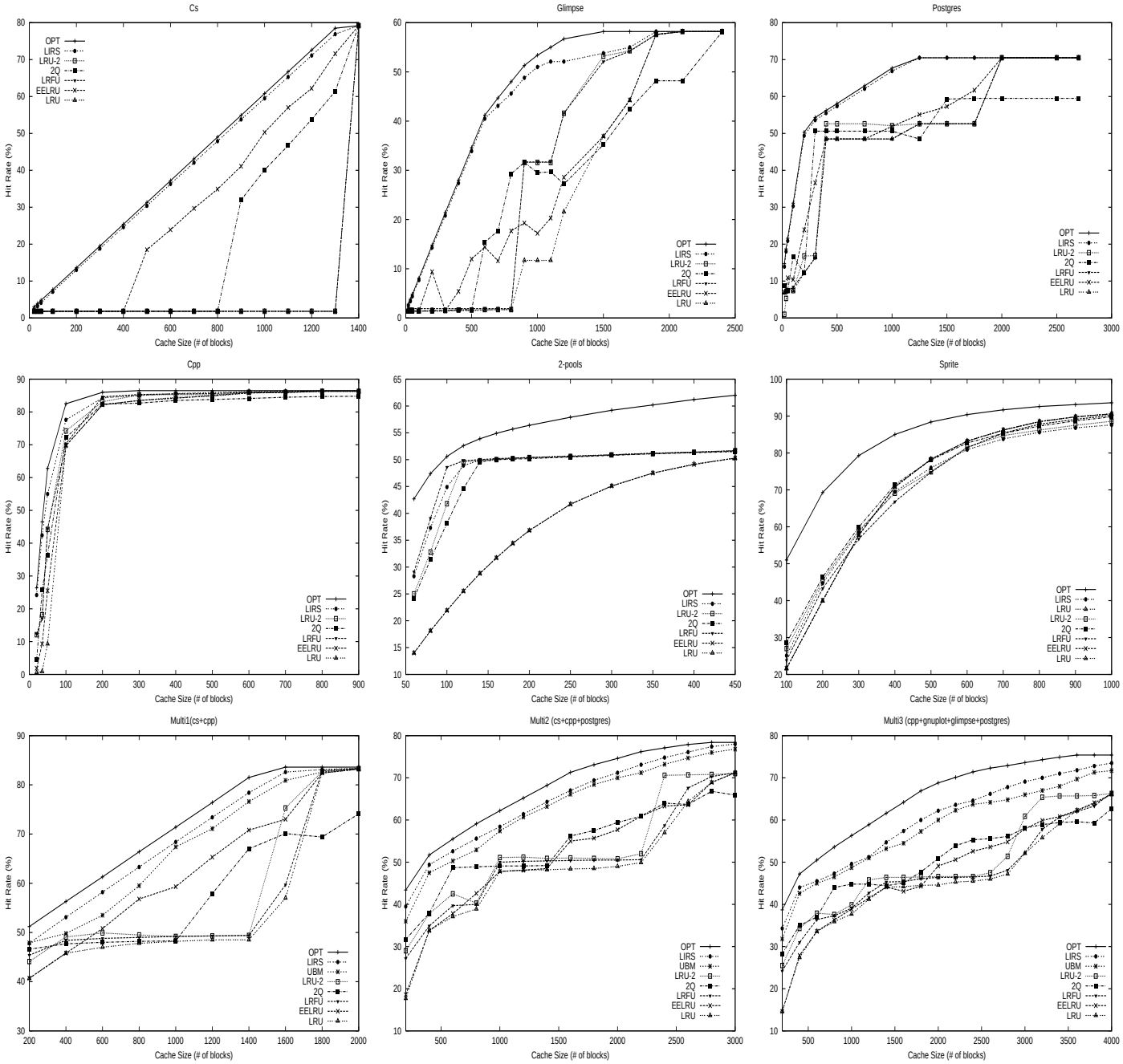

Figure 2: Hit rate curves by various replacement policies on various workloads. Workloads *cs*, *postgres*, and *glimpse* belong to the looping type, workloads *cpp* and $2-pools$ belong to the probabilistic type, workload *sprite* belongs to the temporally-clustered type, and workloads *multi*1, *multi*2, and *multi*3 belong to the mixed type.

100 blocks. LRU can not exploit locality until enough cache space is available to hold all the recently referenced blocks. However, the capability for LIRS to exploit locality does not depend on the cache size – when it is identifying the LIR set to keep them in the cache, it always let set size match the cache size. Workload 2-pools is generated to evaluate replacement policies on their abilities to recognize the long-term reference behaviors. Though the reference frequencies are largely different between record blocks and index blocks, it is hard for LRU to distinguish them when the cache size is relatively small compared with the number of referenced blocks. This is because LRU takes only recency into consideration. LRU-2, 2Q, and LIRS algorithms take one more previous references into consideration — the time for the penultimate reference to a block is involved. Even though the reference events to a block are randomized (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. Thus, the constant long-term frequency becomes more visible, and is ready to be utilized by the LFU-like scheme. The performance of LRFU is a slightly better than that of LIRS. It is not surprising to see the hit rate curve of EELRU performs poorly and overlaps with that of LRU, because EELRU relies on an analysis of a temporal recency distribution to decide whether to conduct an early point eviction. In workload 2-pools, the blocks with high access frequency and blocks with low access frequency are alternatively referenced, thus no sign of an early point eviction can be detected.

4.2.3 Replacement Performance on Temporally-Clustered Patterns

Workload sprite exhibits temporally-clustered reference patterns. The sprite result in Figure 2 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, the slope of the LRU is close to that of OPT. Sprite is a so called LRU-friendly workload [18], which seldom accesses more blocks than the cache size over a fairly long period of time. For this type of workload, the behavior of all the other policies should be similar to that of LRU, so that their hit rates could be close to that of LRU. Before the cache size reaches 350 blocks, the hit rates of LIRS are higher than those of LRU. After this point, the hit rates of LRU are slightly higher. Here is a reason for the slight performance degradation of LIRS beyond that cache size: whenever there is a locality scope shift or transition, i.e. some HIR blocks get referenced, one more miss than would occur in LRU may be experienced by each 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, it does not have frequent locality scope changes, and the negative effect of the extra misses is very limited.

4.2.4 Replacement Performance on Mixed Patterns

Multi1, multi2, and multi3 are the traces with mixed access patterns. The authors in [11] provide a detailed discussion why their UBM shows the best performance among the polices they have considered – UBM, SEQ, 2Q, EELRU, and LRU. Here we focus on performance differences between LIRS and UBM. UBM is a typical spatial regularity detection-based replacement policy that conducts 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 unidentified blocks, LRU is applied. A dynamical buffer allocation among blocks managed by different policies is employed. Without devoting specific effort to specific regularities, LIRS outperforms UBM for all the three mixed type workloads, which shows that our assumption on IRR well holds and LIRS is able to cope with weak locality reference in the workloads with mixed type patterns.

5. SENSITIVITY/OVERHEAD ANALYSIS

5.1 Size Selections of L_{hirs}

L_{hirs} is the only parameter in the LIRS algorithm, which is the maximum size of list Q holding resident HIR blocks. The blocks in the LIR block set can stay in the cache for longer time than those in the HIR block set and experience less misses. 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% of the cache size, L_{hirs} to be 1% of the cache size in our experiments, and achieve expected performance. From the other perspective, an increased L_{hirs} may be beneficial to the performance: it reduces the first time reference misses. For a longer list Q (larger L_{hirs}), it is more possible that an HIR block will be re-accessed before it is evicted from the list, which can help such HIR blocks change into the LIR status without experiencing extra misses. However, the benefit of large L_{hirs} is very limited, because the number of this kind of hits is small.

We select two workloads, a non-LRU-friendly workload, postgres, and an LRU-friendly workload, sprite, to observe the effects of changing L_{hirs}. We change L_{hirs} from 2 blocks, to 1%, 10%, 20%, and 30% of the cache size. Figure 3 presents the results of the sensitivity study. For each workload, we measure the hit rates of OPT, LRU, and LIRS with different L_{hirs} sizes by increasing the cache size. We have following 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} that is not in favor of LIRS, the performance of LIRS with different cache sizes is still quite acceptable. With the increase of L_{hirs}, the hit rates of LIRS approach that of LRU. Secondly, our experiments indicate that increasing L_{hirs} lowers the performance of LIRS for workload postgres, but slightly improves the performance of LIRS for workload sprite.

5.2 Thresholds for LIR/HIR Switching

In the LIRS algorithm, we use maximum recency among those of the LIR blocks, $Rmax$, as a threshold value for LIR/HIR status switching. Any HIR block has a new IRR smaller than the threshold will change into LIR status, and 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 would like to vary the threshold value to get insights into the relationship of LRU and LIRS. 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 LIRS from responding to the relatively small IRR variance. Increasing the threshold value, we go in the opposite direction. In fact, LRU can be considered as a special case of LIRS algorithm with a sufficiently large threshold value, which always gives a block LIR status and keeps it in the cache until it is evicted in the bottom of stack, no matter how large the block's recency is when the block gets accessed.

Figure 4 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%, 75%, 100%, 125% to 150% of $Rmax$. For postgres, we include a huge threshold value – 550% of $Rmax$ to hightlight the relationship between LIRS and LRU. We have two observations. First, LIRS is not sensitive to the threshold values across a large range. In postgres, curves for the threshold values of 100%, 125%, 150% of $Rmax$ are almost overlapped,

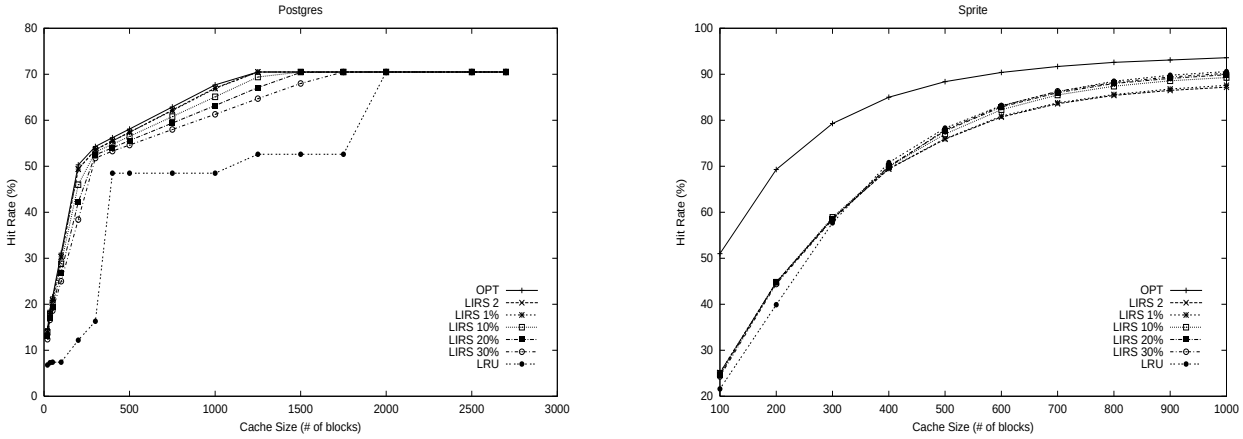

Figure 3: The hit rate curves of workload postgres (left figure) and workload sprite (right figure) by varying the size of list Q (L_{hirs}, the number of cache buffers assigned to HIR block set) of LIRS algorithm, as well as the curves for OPT and LRU. "LIRS 2" means size of Q is 2, "LIRS x%" means size of Q is x% of the cache size in blocks.

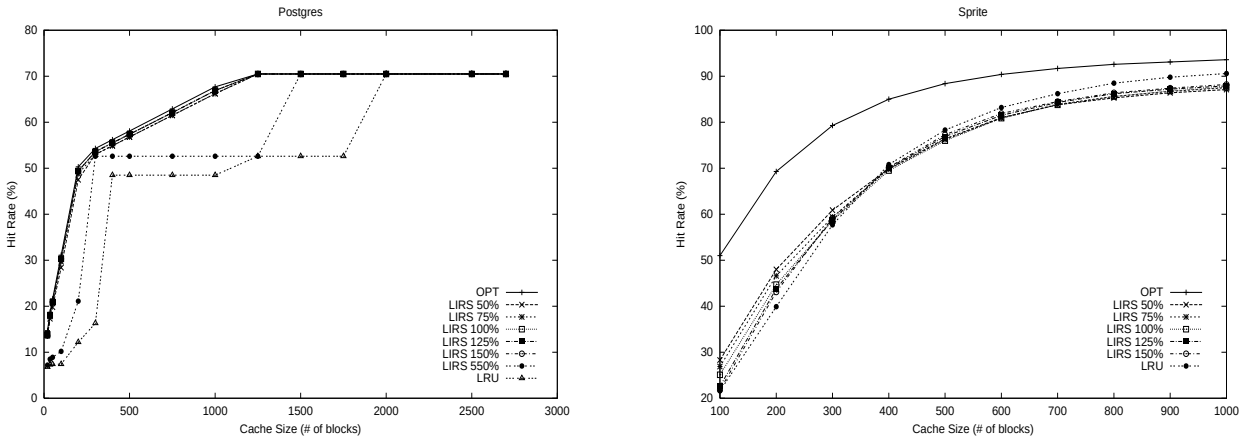

Figure 4: The hit rate curves of workload postgres (left figure) and workload sprite (right figure) by varying the ratios between threshold values for LIR/HIR status switching and $Rmax$ in LIRS, as well as the curves for OPT and LRU.

and curves for 50%, 75% of $Rmax$ are slightly lower than the curve with 100% of $Rmax$ threshold. Specifically for sprite, an LRU-friendly workload, increasing the threshold value, the LIRS hit rate curves move very slowly close to that of LRU. Secondly, the LIRS algorithm can simulate LRU behavior by largely increasing the threshold value. As the threshold value increases to 550% of $Rmax$, LIRS curve of workload postgres is very similar to that of LRU in its shape, and close to it in its values. (See the left figure of Figure 4.) Further increasing the threshold value, LIRS becomes LRU.

5.3 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 that of LRU with a few additional operations such as those on the list 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 entries for the blocks with their recencies less than $Rmax$. We have conducted experiments on all the traces to observe the variance of the ratios between the LIRS stack size, $Rmax$, and the size of the LRU stack in the LRU algorithm, L (or the cache size in blocks). Figure 5 shows when we fix the cache size at 500 blocks, the varying ratios between $Rmax$ and L for workloads postgres and sprite with the virtual time. We find that the ratio is an inherent reflection of the LRU capability to exploit locality of a workload. References to blocks with strong locality leave only a small number of HIR blocks in stack S and the size of stack S is shrinked, because most of references are to LIR blocks. If the size of stack S is close to that of LRU stack, LRU stack can hold sufficient history reference information, thus LRU has a good performance. For example, we observe the ratios of postgres are close to 1 between virtual times 1000 and 1300, between virtual times 4900 and 5300, and between virtual times 8500 and 8900. Analyzing its reference

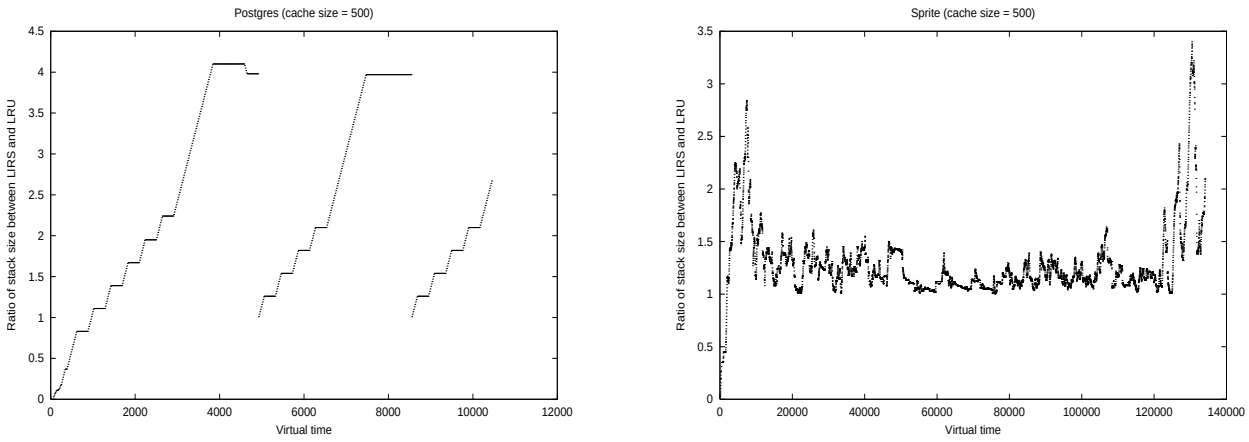

Figure 5: The ratios between $Rmax$ and cache size in blocks (L) for workload postgres (left figure) and workload sprite (right figure). $Rmax$ is the size of LIRS stack, which changes with the virtual time. Cache size is 500 blocks.

steam in the trace, we know that these are the periods when postgres has just experienced relatively strong locality. For other periods with large ratios, LRU could not exploit the weak locality. In contrast, Figure 5 shows that for sprite with the cache size 500, the ratios are not far from 1 for the most of period of time, which implies that it has strong locality, and the LRU stack is able to keep the frequently referenced blocks in the cache. Thus LRU is performance effective. It can be seen that the ratio is a good indicator of the LRU-friendliness of a workload.

When there is a burst of first-time (or "fresh") block references, the LIRS stack could be extended to be unacceptably large. To give 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 out of 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 Figure 6, we can observe that even with these strict space restrictions, LIRS retains its desired performance. As expected, the results are consistent with the ones when we reduce the threshold values in Section 5.2. In addition, a stack entry only consists of several bytes, it is quite affordable to have LIRS stack size limit much more than three times of LRU stack size. With such large limits, there is little negative effect on LIRS performance by removing HIR block entries close to the stack bottom due to the size limit.

While LRU can be seen as a special case of LIRS by limiting the LIRS stack size as L, moderately extending the LRU stack size makes a large difference on its performance. This is because our solution effectively addresses the critical limits of LRU.

6. CONCLUSIONS AND FUTURE WORK

We make two contributions in this paper by proposing the LIRS algorithm: (1) We show that LRU limits with weak locality workloads can be successfully addressed without relying on the explicit regularity detections. Not depending on the detectable pre-defined regularities in the reference of workloads, our LIRS catches more opportunities to improve LRU performance. (2) We show earlier work on improving LRU such as LRU-K, 2Q can be evolved into one algorithm with consistently superior performance, without tuning or adapting sensitive parameters. The efforts of these algorithms, which only trace their own history information of each referenced block, are promising because it is very likely to produce a simple and low overhead algorithm just like LRU. We have shown the LIRS algorithm accomplishes this goal.

In summary, our LIRS algorithm assumes that there exists stability on the IRR of a block over certain time. This certainly holds for the access patterns with strong locality, where LRU works well. The strong point of LIRS lies on its ability to deal with access patterns with weak locality. By extending LRU stack into LIRS stack, we successfully make LRU a special case of LIRS. LRU uses the LRU stack to capture the working set and then keep it in the cache. Weak locality actually enlarges the working set. Once the LRU stack cannot hold it, the performance of LRU could be much worse than expected. However, LIRS uses the LIRS stack to capture the working set, and uses the LIR block set to hold its portion most deserved to be in the cache under the assumption. Algorithms like LRU-K, and 2Q make their efforts for the similar purpose. However, without an effective mechanism to identify the most deserved portion matching the currently available cache size, they either significantly increase the cost and/or introduce workload sensitive parameters.

Our LIRS algorithm can be effectively applied in the virtual memory management for its simplicity and its LRU-like assumption on workload characteristics. Currently we are designing and implementing an LIRS approximation with reduced overhead comparable to that of LRU approximations, such as the clock, second chance algorithms.

Acknowledgments: We are grateful to Dr. Sam H. Noh at Hong-Ik University, Drs. Jong M. Kim, Donghee Lee, Jongmoo Choi, Sang L. Min, Yookun Cho, and Chong S. Kim at the Seoul National University, to provide us with their

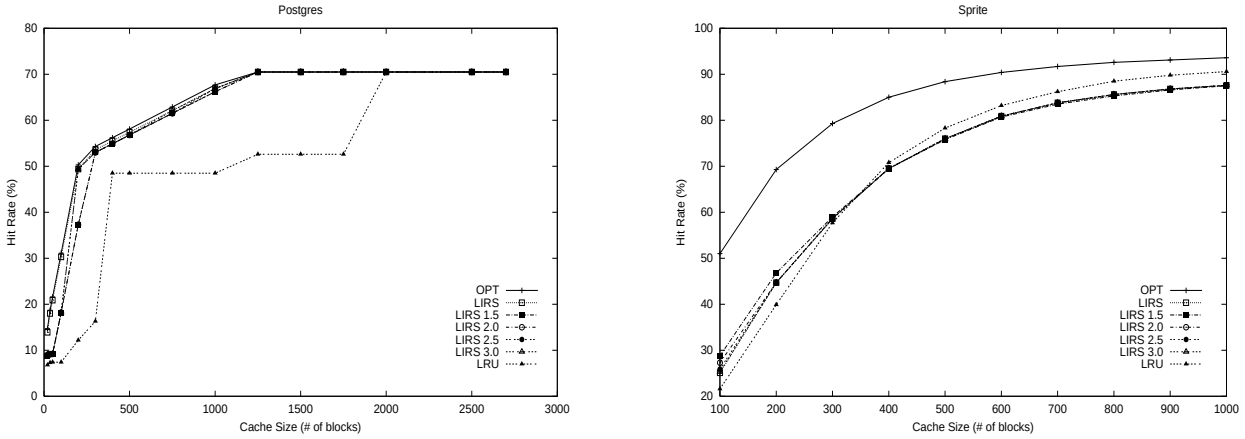

Figure 6: The hit rate curves of workload postgres (left figure) and workload sprite (right figure) by varying the LIRS stack size limits, as well as the curves for OPT and LRU. The limits are represented by the ratios between the LIRS stack size limit in blocks and the cache size in blocks (L).

traces and simulators used in their papers [4], [11], and [12]. The comments from the anonymous referees are constructive and helpful. We thank our colleague Bill Bynum to read the paper and his comments. Finally, this work is also a part of an independent research project sponsored by the National Science Foundation for its program directors and visiting scientists.

7. 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", *Communication of the ACM*, Vol. 12, June 1969, pp. 349-353.

[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", *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994, pp. 171-182.

[4] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "Towards Application/File-Level Characterization of Block References: A Case for Fine-Grained Buffer Management", *Proceedings of 2000 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, June 2000, pp. 286-295.

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

[6] P. J. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, Vol. 11, No. 5, May, 1968, pp. 323-333.

[7] P. J. Denning, "Virtual Memory", *Computer Survey* Vol. 2, No. 3, 1970, pp. 153-189.

[8] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management", *ACM Transaction on Database Systems*, Dec, 1984, pp. 560-595.

[9] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior", *Proceedings of 1997 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1997, pp. 115-126.

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

[11] 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", *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, October 2000, pp. 119-134.

[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", *Proceeding of 1999 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1999, pp. 134-143.

[13] T. C. Mowry, A. K. Demke and O.Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Application", *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 1993, pp. 297-306.

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

[15] V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior", *Proceeding of 1995 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, 1995, pp. 291-300.

[16] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, "Informed Prefetching and Caching", *Proceedings of the 15th Symposium on Operating System Principles*, 1995, pp. 79-95.

[17] J. T. Robinson and M. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement", *Proceeding of 1990 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1990, pp. 134-142.

[18] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement", *Proceedings of 1999 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1999, pp. 122-133.

[19] J. R. Spirn, "Distance String Models for Program Behavior", *IEEE Computer*, Vol. 9, 1976, pp.14-20.