2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

PAC-PLRU: A Cache Replacement Policy to Salvage Discarded Predictions from Hardware Prefetchers

Ke Zhang[1,2,*], Zhensong Wang[1,†], Yong Chen[3], Huaiyu Zhu[4], Xian-He Sun[5]
[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, P.R.China
[2] Graduate University of Chinese Academy of Sciences, Beijing, 100049, P.R.China
[3] Department of Computer Science, Texas Tech University, Lubbock, TX 79409, U.S.A.
[4] Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.
[5] Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, U.S.A.
zhangke@ict.ac.cn, zswang@ict.ac.cn, yong.chen@ttu.edu, hzhu10@illinois.edu, sun@iit.edu

***Abstract*—Cache replacement policy plays an important role in guaranteeing the availability of cache blocks, reducing miss rates, and improving applications' overall performance. However, recent research efforts on improving replacement policies require either significant additional hardware or major modifications to the organization of the existing cache. In this study, we propose the PAC-PLRU cache replacement policy. PAC-PLRU not only utilizes but also judiciously salvages the prediction information discarded from a widely-adopted stride prefetcher. The main idea behind PAC-PLRU is utilizing the prediction results generated by the existing stride prefetcher and preventing these predicted cache blocks from being replaced in the near future. Experimental results show that leveraging the PAC-PLRU with a stride prefetcher reduces the average L2 cache miss rate by 91% over a baseline system with only PLRU policy, and by 22% over a system using PLRU with an unconnected stride prefetcher. Most importantly, PAC-PLRU only requires minor modifications to existing cache architecture to get these benefits. The proposed PAC-PLRU policy is promising in fostering the connection between prefetching and replacement policies, and have a lasting impact on improving the overall cache performance.**

Keywords—cache replacement policy; high-performance processors; computer architecture; memory wall

I. INTRODUCTION

With the advancement of microarchitecture and semiconductor process technology, the performance gap between processor and memory has been significantly widened. To break this huge "Memory Wall" [28], a CPU cache is commonly used to reduce the average time of accessing the memory [9]. However, in modern microarchitecture design, a cache miss may cost several hundred CPU clock cycles to fetch the data from the off-chip memory [16]. Due to the limitation of cache capacity and in order to reduce the miss rate, cache replacement policies evict unnecessary cache blocks for the purpose of making good utilization of the silicon estate devoted to the caches and keep the frequently used blocks within caches as well. In short, cache replacement policies should kick out dead blocks and keep hot ones.

Least Recently Used (LRU), First In, First Out (FIFO) and Random (RAND) are the three most elementary and commonly used cache replacement policies [1]. During the last two decades, substantial variants based on these three basic policies and numerous other policies have been proposed by architecture researchers from both academia and industry. For example, Pseudo-LRU or Partial-LRU (PLRU) [9, 20], Most Recently Used (MRU) [23], Least Frequently Used (LFU) [17], Not Last Used (NLU) [9], Modified LRU [27], and Self-Correcting LRU [14]. Recently, some other advanced cache replacement policies incorporated prediction information. However, they needed a dedicated predictor or prefetcher to identify the dead blocks and evict them early [15, 11, 18], which would complicate this kind of policy to be implemented in real processors. Pseudo-LIFO [4] needs $2n\log_2(n)+2n$ bits per set for an n-way set associative cache. Re-Reference Interval Prediction (RRIP) [13] requires 2n bits per set and special logic circuits. In all of the above described work, these solutions require either significant additional hardware or major modifications to the organization of the existing cache or complex control logic. To the best of our knowledge, PLRU, LRU, RAND and FIFO are still the most favorable choices among modern processors due to their simplicity and acceptable performance.

In this paper, we propose a modified PLRU cache replacement policy. Our Prediction-Aware Confidence-based Pseudo LRU (PAC-PLRU) not only utilizes but also judiciously salvages the prediction information discarded from a widely-adopted stride prefetcher. The reason of recycling prediction results is that, if a block to be prefetched already lies in the cache, it should be avoided evicting from the cache. Experimental results show that leveraging the PAC-PLRU with a stride prefetcher reduces the average L2 cache miss rate by 91% over a baseline system with only PLRU policy, and by 22% over a system using PLRU with an unconnected stride prefetcher at the expense of increasing memory bus usage by only 7.9%. As a result, PAC-PLRU can benefit from the existing stride prefetcher without sacrificing performance or

[*] This work was primarily performed while he was a visiting student at Illinois Institute of Technology.
[†] To whom correspondence should be addressed.

978-0-7695-4395-6/11 $26.00 © 2011 IEEE
DOI 10.1109/CCGrid.2011.27

adding an extra predictor or prefetcher.

We make two primary contributions in this paper:

- We observe that more than three quarters of predicted blocks are discarded by the filtering mechanism inside a stride prefetcher (Section II). By salvaging these abandoned predictions, the basic PLRU replacement policy becomes prediction-aware (Section III).
- We propose to convert the confidence level of these discarded predictions into the priority of nodes in the binary tree used by the basic PLRU policy, so that this prediction information are not only incorporated into but also judiciously utilized by our proposed PAC-PLRU policy (Section IV). Therefore, PAC-PLRU is prediction-aware as well as confidence-based.

The rest part of this paper is organized as follows: Section V discusses our evaluation methodology and analyzes simulation results of PAC-PLRU policy. Section VI reviews and evaluates important related work from both academic research and real industry products. Finally, Section VII concludes this study and discusses potential future work.

II. Analysis of PLRU Replacement Policy and Stride Prefetcher

2.1. Basic PLRU Replacement Policy

LRU replacement policy evicts the cache block which has not been used for the longest time in a cache set. It requires a stack to store the accessing sequence. For high-associative caches, LRU is costly to implement in hardware because a lot of storage bits are needed to maintain this stack.

Pseudo LRU (PLRU) is a tree-based approximation of the true LRU policy in that the block usage information is maintained in a binary tree, thus reducing the hardware overhead [9, 1]. For an N-way set associative cache, PLRU policy arranges the cache blocks at the leaves of a tree with (N-1) tree nodes pointing to the block to be replaced next. Each node of the tree has a one-bit flag denoting "go left to find a PLRU candidate" (flag bit = 0) or "go right to find a PLRU candidate" (flag bit = 1). On a cache miss, the binary tree of the relevant cache set is traversed to find a PLRU candidate based on the flag values. On a cache access, the tree is updated during the traversal: the node flags are set to denote the direction that is opposite to the direction taken.

Compared with the true LRU policy, *Pseudo* LRU does not always select the least-recently used block as the next one to replace. Consider the access sequence (in ways) A-B-C-D-A for a 4-way set associative cache, the block selected for replacement is Cache Block C, not Block B as is done in the true LRU algorithm. However, PLRU does ensure that the block selected for replacement is either the least-recently or the second least-recently used cache block.

Figure 1 illustrates PLRU behavior using a 4-way set associative cache as an example. In Figure 1 (a), the three flag bits Flag[2:0] form a decision binary tree. The Flag[0] bit indicates whether two lower blocks A and B (Flag[0] = 1), or two higher blocks C and D (Flag[0] = 0) have been recently used. The Flag[1] bit determines further which one of two blocks A (Flag[1] = 1) or B (Flag[1] = 0) has been recently used; Flag[2] keeps the access track between Block C and D. In Figure 1 (b), on a cache hit, the tree nodes are set according to Table 1. On a cache miss, Flag[0] determines where to look for the least recently block (two lower cache blocks or two higher cache blocks). Flag[1] or Flag[2] determines the least recently used block. For the same 4-way set associative cache, the truth table for selecting replacement candidates on cache misses is shown in Table 2.

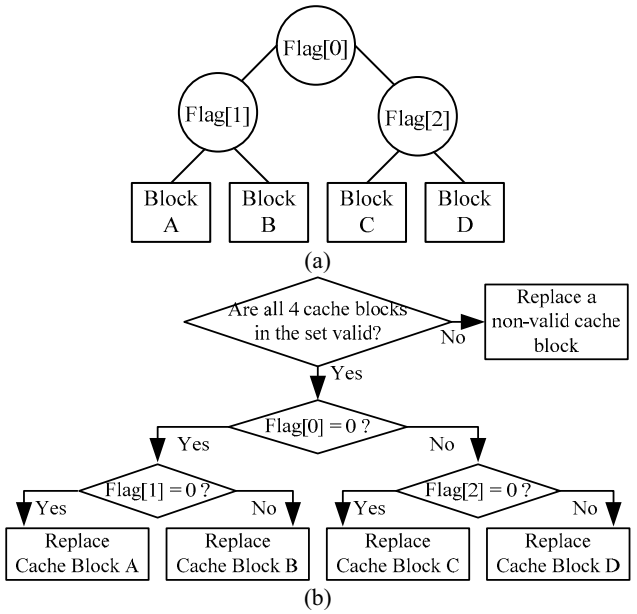

Figure 1: (a) Binary Tree-based PLRU replacement policy for one cache set in a 4-way set associative cache. (b) The process of searching a replacement candidate in the PLRU policy.

Table 1: Truth table for updating flag bits in the decision binary tree at cache hits.

Which cache block is hit?	Flag[0]	Flag[1]	Flag[2]
Cache Block A	1	1	no change
Cache Block B	1	0	no change
Cache Block C	0	no change	1
Cache Block D	0	no change	0

Table 2: Truth table for selecting replacement candidates based on flag bits in the decision binary tree at cache misses.

Flag[0]	Flag[1]	Flag[2]	Replacement candidate
0	0	0	Cache Block A
0	0	1	Cache Block A
0	1	0	Cache Block B
0	1	1	Cache Block B
1	0	0	Cache Block C
1	0	1	Cache Block D
1	1	0	Cache Block C
1	1	1	Cache Block D

2.2. Behavior Analysis of Stride Prefetcher

Stride prefetching detects the stride patterns originating from looping structures in data access streams [3]. This detection of stride is accomplished by comparing successive addresses used by load or store instructions. Most current prefetchers in commercial processors from Intel, IBM and AMD can predict stride pattern in data access streams [6, 10, 16, 24, 2]. For example, the Intel Core microarchitecture utilizes a Program Counter-indexed hardware stride prefetcher for L1 data cache [6].

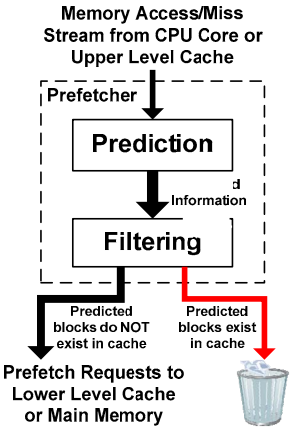

Figure 2: Process flowchart of a general stride prefetcher. Prefetcher utilizes the recent information of memory access or miss stream from CPU core or upper level cache to predict the future information by the Prediction procedure. Then, the process of Filtering discards the Predicted Information of the predicted blocks which already exist in cache, and sends the Prefetch Requests to lower lever cache or Main Memory if the predicted blocks are not present in cache. In this figure, the wider the arrows are, the larger the amount of information is.

Figure 2 illustrates the process flowchart of a general stride prefetcher with two cascaded procedures: prediction and filtering. First, based on the accessing history, predicted information of which cache blocks might be accessed in the near future is generated during the process of prediction. Then, if some predicted cache blocks are currently present in the cache, this part of information will be discarded by the filtering process because it is not necessary to fetch something that already exists, and the redundant prefetch requests can even increase the burden of memory bus. Although the discarded information is useless to the prefetcher, it could be the source of benefits of PLRU replacement policy because the replacement policy would harness this future information and keep the upcoming accessed blocks in the cache.

Further analysis in Figure 3 shows that the prefetcher discards a large amount of information by filtering. For all 29 SPEC CPU2006 benchmarks, an average of 76% of predicted blocks already exist in L2 cache using a traditional stride prefetcher. We believe that this discarded information could be transformed into certain potential benefits of replacement policy and further potential improvement in cache performance. Notice that our observation is that a large fraction of predictions hit in the cache, not a large fraction of prefetches. The blocks that are requested by the prefetches are fresh to the cache, and all the prefetches should not hit in the cache; however, a part of predictions are filtered before the prefetcher issues requests, which is indicated by the fact that three quarters of predicted blocks hit in L2 cache for most benchmarks in Figure 3. As a result, a very large amount of information that is potentially useful for replacement policy has been filtered. Moreover, the larger the amount of discarded information, the more potential benefits the PLRU replacement policy could gain.

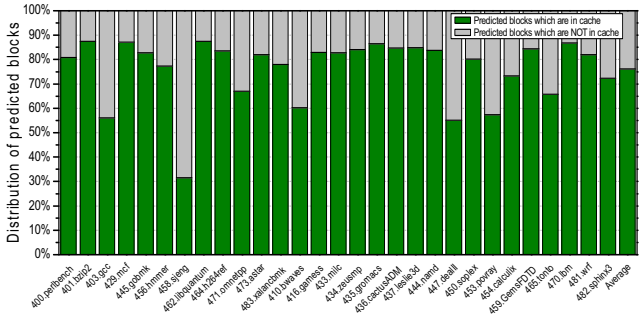

Figure 3: Distribution of predicted blocks before filtered for SPEC CPU2006 benchmarks

III. Prediction-Aware PLRU Replacement Policy

"One Man's Junk is Another Man's Treasure."

Drawing on the wisdom in the above old saying, we propose to incorporate the discarded prediction information from a stride prefetcher into the PLRU replacement policy.

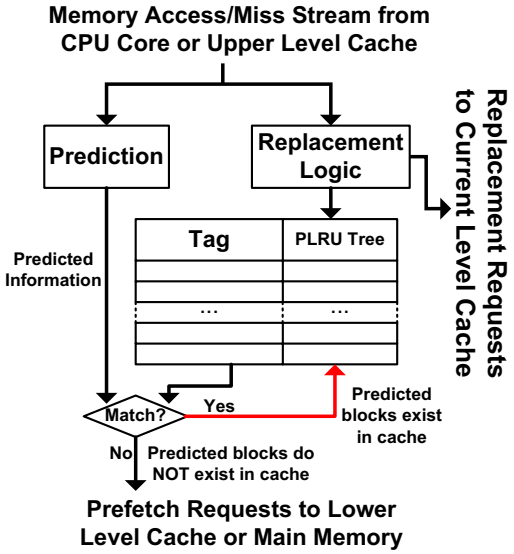

Figure 4: The Structure of our proposed Prediction-Aware PLRU Replacement Policy. The filtering mechanism in Figure 2 is shown as a Match operation in this figure. If the predicted blocks do not exist in cache, Prefetch Requests are issued as the same as that in Figure 2; if there is a match, our method will modify the corresponding PLRU binary tree as the red arrow indicates. The Replacement Logic examines the PLRU trees during cache accesses. For simplicity, other main components in the cache controller are omitted.

Figure 4 shows the structure of our proposed Prediction-Aware PLRU Replacement Policy which is implemented in a cache controller. The cache controller determines whether or not a predicted address generated from the predictor in a stride prefetcher exists in the cache by simply looking it up in the cache tags just as a normal access. The filtering mechanism in Figure 2 is shown as a match operation in Figure 4. If the predicted blocks do not exist in cache, prefetch requests are issued as the same as that in Figure 2. When a match occurs, our proposal makes changes to the node value of the cache block in its corresponding PLRU binary tree during the process of lookup. Thus, the PLRU binary tree turns into the medium of information transmission over the connection between the stride prefetcher and PLRU cache replacement policy. As shown in Figure 4, our proposal does not require any additional hardware overhead, but just some operations on PLRU binary trees in this scenario.

Since the lookup of existence of a predicted address is unavoidable, it is trivial to make changes to the nodes' value of PLRU binary trees during the process of lookup. Therefore, the inspection of predicted addresses and the change of PLRU binary trees can overlap and the time overhead of our proposed prediction-aware PLRU replacement policy is negligible.

IV. PAC-PLRU Replacement Policy

Previous section shows how to incorporate prediction information into the PLRU policy, and the PLRU tree is modified during the inspection of predicted addresses. However, it is not wise to boldly change the binary trees because the predictions are not real accessing information. In this section, we will tackle the problem of how to judiciously salvage the prediction information discarded from a stride prefetcher.

4.1. Priority in Binary Tree-based PLRU

As mentioned in Figure 1 (a), for a 4-way set associative cache, PLRU policy has a two-level binary tree and three nodes to represent the LRU status for 4 cache blocks in each cache set. Based on the operation process of PLRU, we find that different layers in the binary tree of PLRU have different priorities. The higher level the nodes stay in, the larger coverage they can control. For example, the root node has the highest priority because modifying the value of the root node can move the replacement candidate to the other half of the cache blocks in the cache set. In contrast, leaf nodes have the lowest priority. Toggling the value of a leaf node can only move the replacement candidate to the adjacent cache block in the cache set. It should be noted that the basic PLRU replacement policy modifies every level in the binary tree when there is an update due to a memory access (hit or miss) that just happened. This is understandable because for something that already took place, we are 100% sure about its possibility, and we have full confidence to proceed to the following steps. However, what would happen if we did not have 100% confidence under certain circumstances? For example, how should we update the binary trees when these predicted blocks will be accessed in the near future with a level of confidence? Our solution to this problem is called Prediction-Aware Confidence-based Pseudo LRU (PAC-PLRU) replacement policy.

4.2. Confidence Level of Prediction

It is well known that the further the predictor forecasts, the less accurate the predicted information is. For example, if cache block A was just accessed 3 clock cycles ago, block B is predicted to be accessed in 30 clock cycles, and block C is predicted to be referenced in 4,000 clock cycles, then we have full confidence in block A and less than 100% confidence in the accuracy of predictions about block B and C. It is also obvious that the prediction confidence in block B is much stronger than that in block C because the rule of thumb is that more recent behavior predicts the future better. As a result, the confidence level is related and inversely proportional to the degree of prediction, which is defined as the position of a particular predicted block in one prediction sequence. The predicted blocks with more confidence should be retained in the cache for a longer time than those with less confidence. In other words, on a cache miss, the predicted blocks with higher confidence levels should be exempt as much as possible from being selected as the replacement candidates.

4.3. PAC-PLRU Replacement Policy

The basic idea of PAC-PLRU is converting the confidence level of predictions into the priority of the nodes in the binary tree. As explained in the previous two subsections, the information of prediction degree can be converted into the number of levels that should be modified in the binary tree. The mechanism of PAC-PLRU policy is designed as follows. For the original data access information, the behavior of PAC-PLRU is the same as the basic PLRU policy due to full confidence in the referencing information. For the predicted future information, PAC-PLRU assumes that these predicted blocks are normal cache accesses, but with a certain confidence level. As a result, for these predicted blocks, PAC-PLRU only modifies part of the binary tree, from leaf nodes to upper levels, based on how much the confidence is, from weak to medium to strong.

A hash function uses the number of the prediction degree of the predicted block to form the number of levels that should be modified in the binary tree. We define K as the maximum prediction degree, and k as the position of a particular predicted block in this prediction sequence, $k \in \{1,\ 2,\ 3,\ \cdots,\ K\}$. The equation $L_i = \log_2(N) - i$ calculates the number of modified tree levels (L_i) based on the parameter of position k for the i th segment, where i is an integer from 1 to $\log_2(N)$ that satisfies the following formula:

$$\left(\left\lceil\frac{(i-1)\cdot K}{\log_2(N)}\right\rceil+1\right)<k<\left\lceil\frac{i\cdot K}{\log_2(N)}\right\rceil.$$

For an *N*-way set associative cache, the whole prediction sequence K is divided into the number of $\log_2(N)$ segments, and i is the index number of a segment. Each segment has a corresponding total number L_i of modified tree levels based on the above equation. We can see that $\log_2(N)$ is also the total number of levels in the binary tree. Therefore, the lower the segment index number is, the more levels PAC-PLRU modifies in the binary tree. Notice that the modification to the binary tree is made from leaf nodes to upper levels, except the root node. The root node can only be toggled by an actual memory access. An example of the behavior of PAC-PLRU is given in the next subsection.

This approach statically calculates the number L_i for each segment in advance. As a result, it can be easily implemented in hardware. The PAC-PLRU replacement logic converts i into L_i based on a small hash table generated by the hash function beforehand.

According to this method, we statically connect the degree and confidence level of prediction with the structure of the binary tree. Although the confidence level can also be changed dynamically, under the current architecture, experimental results in Section V show that it is sufficient to statically define the confidence level. How to dynamically change the confidence level will be our future work.

4.4. Example of the Hash Function in PAC-PLRU

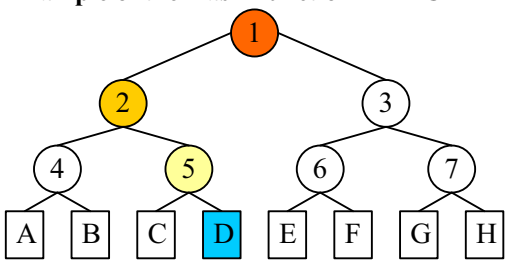

Figure 5: An example of behavior of the hash function for an 8-way set associative cache. The circles represent the nodes in the binary tree which has 3 levels and 7 nodes (No. 1 to 7). The rectangles represent the cache blocks in one cache set which has 8 blocks (A to H).

To help readers better understand the PAC-PLRU policy, a simple example of the behavior of the hash function mentioned in previous subsection is shown here. Take an 8-way set associative cache as an example in Figure 5. Assume a specific case: cache block D in one cache set was just accessed, and then based on the history information, the predictor sends out a 4-degree prediction sequence, which consists of block F, B, C and G. It should be noticed that these blocks, D, F, B, C and G, are not necessarily in one cache set; in most cases, they belong to different cache sets, so they have different binary trees. But here, we use Figure 5 below to show the relative position of the nodes in these trees.

According to the condition in L_i equation, block F and B are in segment 1, block C is in segment 2, and block G is in the last segment. Based on the previous description, block D belongs to the real cache referencing information. In consequence, the number of tree levels that should be modified could be calculated by L_i equation. For block D, node 5, 2 and 1 need to be updated with 100% confidence level. For block F and B, node 6 and 3, node 4 and 2 should be toggled with strong confidence. For block C, only node 5 needs to be changed. Lastly, nothing will be modified for block G because it has the weakest confidence level.

V. Evaluation and Analysis

5.1. Experimental Methodology

In this study, an instrumentation-driven simulator CMP$im [12] with Pin tools [19] is used to collect memory traces of the SPEC CPU2006 benchmark suite [25] on a real machine based on the representative simulation points generated by SimPoint [22]. Then we use a prefetching kit called PREF_KIT [7] to get both predicted trace and uncompressed data access trace. Finally, these traces are imported to a trace-driven cache simulator Dinero [8] to verify our proposed PAC-PLRU.

5.2. Simulation Environment

Table 3: Architectural Configurations

Item	Parameter
Processor Pipeline	4-wide, 15-stage, OoO
Instruction Window	128-entry
L1 cache organization	32K/64K, 4/8-way
L2 cache organization	256K/512K, 8/16-way
Cache Block Size	64B for L1&L2
Default Replacement Policy	Basic PLRU for L1&L2
L2 Cache Latency	20 cycles
Memory Latency	220 cycles
L2 Bandwidth	1 cycle/access
Memory Bandwidth	16 cycles/burst
L2 MSHRs	32-entry
Prefetch Degree	8
Prefetch Distance	0
Stride Prefetcher Table Size	1024-entry

As shown in Table 3, the simulator was configured as an out-of-order processor with a 15-stage, 4-wide pipeline and perfect branch prediction. L1 cache is 32KB/64KB and 4/8-way set associative. L2 cache is 8/16-way with a capacity of 256KB/512KB. The default configuration for cache replacement policy follows basic PLRU policy. The stride prefetcher used in our experiments is similar to that used in the Intel Core microarchitecture which utilizes a Program Counter-indexed hardware table to store the stride

information [6]. The simulation testing was conducted with the complete 29 benchmarks from the SPEC CPU2006 suite. The benchmarks were compiled using GCC 4.2.4 with -O3 -funroll-all-loops -ffast-math optimization and -m32 option. We collected traces for all benchmarks by fast forwarding to the representative points and then running 200 million instructions. The ref input size was used for all benchmarks.

5.3. Misses per Kilo-Instructions (MPKI) of PAC-PLRU

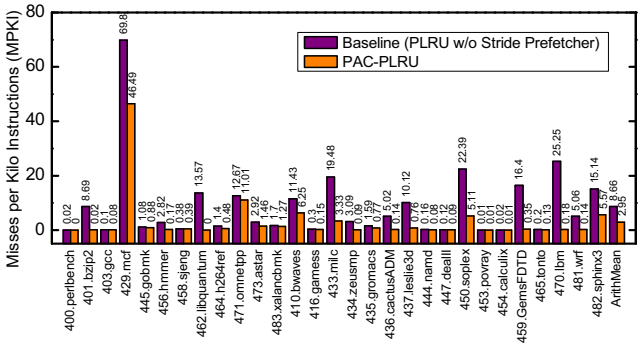

Figure 6: Misses per Kilo-Instructions (MPKI) of PAC-PLRU policy compared with a baseline system. L1 32K 4-way, L2 256K 8-way, MPKI results on L2 cache.

Figure 6 shows the changes of Misses per Kilo-Instructions (MPKI) using PAC-PLRU policy. For a baseline system only with the basic PLRU replacement policy, the average MPKI for all 29 SPEC CPU2006 benchmarks is 8.66. After judiciously salvaging the prediction information discarded from a stride prefetcher, our proposal reduces the average value of MPKI to 2.95. However, detailed analysis shows that about one third benchmarks in the whole SPEC CPU 2006 suite have very low miss rate (MPKI < 1) for the representative simulation points we select. Thus, these benchmarks cannot get a lot of benefits from any improvement on the baseline system. In consequence, we will only use the benchmarks with high miss rate (MPKI > 1) in the following experiments.

5.4. Cache Performance of PAC-PLRU on L2 Cache

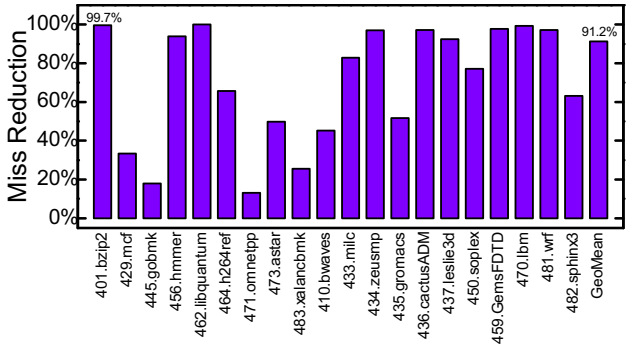

Figure 7: Improvement in L2 miss reduction by PAC-PLRU with stride prefetcher over the baseline system for 20 high-miss-rate SPEC CPU2006 benchmarks.

In this section, we use the miss reduction, which means the total number of reduction in cache misses, to show the improvement of cache performance. Figure 7 shows the L2 cache miss reduction by using our proposed PAC-PLRU replacement policy with a widely-adopted stride, compared to the baseline system with only the basic PLRU policy and without any prefetchers. It can be clearly observed that 13 out of 20 benchmarks in SPEC CPU2006 suite gained significant performance improvement (over 50%) leveraging the PAC-PLRU policy. On average, PAC-PLRU with the stride prefetcher reduced average L2 cache miss rate by 91% over the baseline system with only PLRU. The reason of this huge improvement is because these programs show strong repeated stride pattern of memory access that can be correctly predicted by a stride prefetcher, and a large amount of predictions already exist in L2 cache.

The simulation results reported in Figure 7 show that, for several benchmarks, PAC-PLRU largely reduced cache misses and achieved nearly 100% (99.x% actually; for instance, 99.7% for *401.bzip2*) L2 miss reduction. Note that the cold misses are included in the gap between 99.x% and 100%. Therefore, like any other replacement policies, PAC-PLRU still suffers from L2 cache misses, even it largely reduced misses and achieved near-optimal replacement results for the simulation of 200 million instructions in several benchmarks.

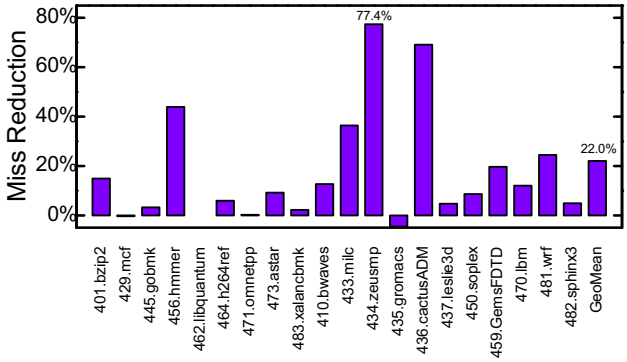

Figure 8: Improvement in L2 miss reduction by PAC-PLRU over the baseline system with a disconnected stride prefetcher for 20 high-miss-rate SPEC CPU2006 benchmarks.

Figure 8 shows the L2 cache miss reduction by using our proposed PAC-PLRU replacement policy, compared to the baseline system with the basic PLRU policy and independent stride prefetcher. We would like to show the miss ratio reduction of PAC-PLRU itself when ignoring the effect of prefetching by this comparison. The statistics report that, on average, PAC-PLRU with the stride prefetcher reduced average L2 cache miss rate by 22% over the baseline system with PLRU and a stride prefetcher.

In Figure 8, several benchmarks had noticeable improvement, such as *434.zeusmp*, *456.hmmer*, *433.milc* and *436.cactusADM*. This is because the PAC-PLRU indeed makes good utilization of the discarded information from the stride prefetcher. However, two applications, *435.gromacs* and *429.mcf*, had negative performance improvement over the baseline system with an unconnected

stride prefetcher. Our analysis of the trace segment of these benchmarks we have reveals that, the stride prefetcher fails to detect any useful pattern, so the prediction information is not accurate enough for PAC-PLRU to retain useful cache blocks. Therefore, the cache is polluted by those unwanted blocks.

Moreover, PAC-PLRU has a significant role here because it increases the life-time of those in-accurately predicted blocks. How to avoid completely relying on the prefetcher is a future study for enhancing the PAC-PLRU.

5.5. Cache Performance of PAC-PLRU on L1 Cache

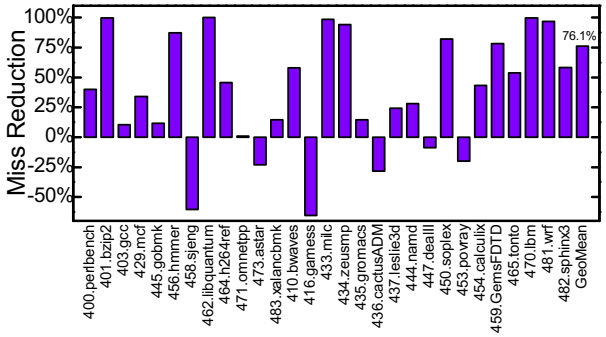

Figure 9: Improvement in L1 miss reduction by PAC-PLRU over the baseline system for 29 SPEC CPU2006 benchmarks.

Compared to the baseline system with only the basic PLRU policy, Figure 9 demonstrates the L1 cache miss reduction by using our proposed PAC-PLRU replacement policy with a stride prefetcher. On average, PAC-PLRU reduced the L1 cache miss rate by 76% over the baseline system with only PLRU. According to Figure 9, PAC-PLRU could improve L1 cache, but not as much as that in L2, even some applications show large negative performance improvement, such as *458.sjeng* and *416.gamess*. There are three reasons for this. Substantial L1 cache accesses are demanded by the CPU core compared to the number of L2 accesses. Also, a considerable amount of information from the L1 cache that fed to the prefetcher constrains its effectiveness by either rendering cache pollution or poor timeliness. Third, the MPKI for L1 cache is much lower than that for L2. These are the main reasons why we highly suggest implementing our mechanism on the last level cache.

5.6. Sensitivity to Cache Configuration

Figure 10 shows the L2 cache miss reduction of PAC-PLRU replacement policy with a stride prefetcher for four different configurations of L1 and L2 cache organization. The PAC-PLRU gains substantial and stable performance improvement regardless of different cache configurations in most benchmarks. However, *416.gamess* is an exception. In normal cases, the larger the cache is, the more miss reduction is. But *416.gamess* has higher miss rate when the cache capacity or associativity increases. Moreover, the gap is significant. We can see the effect of this application's sensitivity in the next section about memory bus usage.

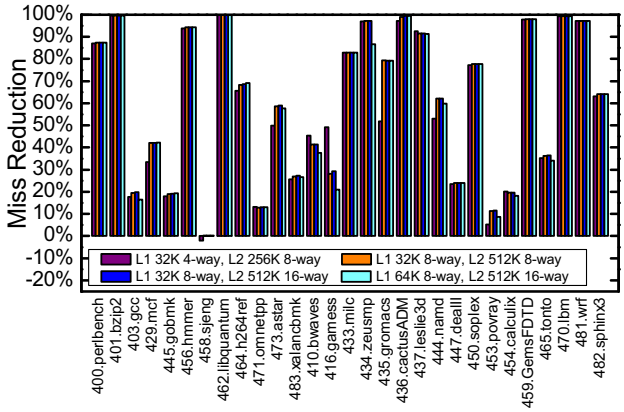

Figure 10: Improvement in L2 cache miss reduction by PAC-PLRU with a stride prefetcher over the baseline system for four different configurations of L1 and L2 cache organization.

5.7. Memory Bus Usage

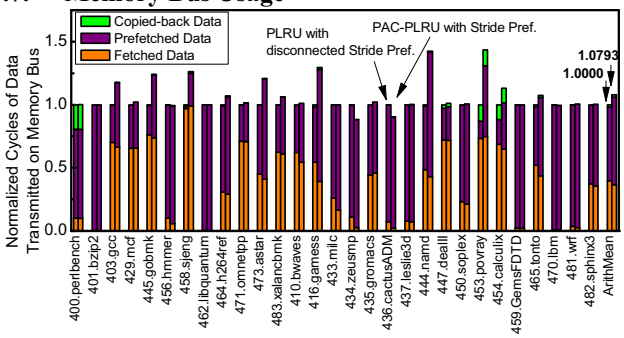

Figure 11: Normalized cycles of data transmitted on memory bus for PAC-PLRU with stride prefetcher.

Figure 11 shows the normalized clock cycles of data transmitted on the L2-to-main memory bus. We classify the data into three types: (1) copied-back data - the data transferred back to the next level storage, (2) prefetched data - the data fetched by prefetching requests, and (3) fetched data - the data issued by normal memory accessing instructions. The latter two categories dominate the memory bus in our experiments. This analysis of memory bus usage reflects some impact of PAC-PLRU on the programs' performances.

On average, PAC-PLRU with a stride prefetcher increased memory bus usage by only 7.9%, compared to the baseline system with PLRU and a disconnected prefetcher. About half of the benchmarks performed well, which means no significant increase in memory bus contention. Moreover, for several applications, such as *434.zeusmp*, *436.cactusADM*, *456.hmmer* and *470.lbm*, the total number of clock cycles used for transmitting data are limited to a certain degree. However, *416.gamess* along with several other applications was the worst case due to its high sensitivity to different cache configurations. Its high demand of memory bandwidth is because the working set of this application cannot be effectively fetched or prefetched into L2 cache.

The stride prefetcher in our experiments is not very bandwidth-efficient. However, it is widely accepted by both academia and industry due to its simplicity and high performance. Thus, we use it to show our PAC-PLRU idea is universal and simple. We believe that future bandwidth-efficient and advanced prefetchers, such as [5, 29], can also benefit from PAC-PLRU.

5.8. Hardware Cost of PAC-PLRU

To implement the concept of PAC-PLRU, several modifications need to be made on current existing cache hardware, especially in the cache controller, but the implementation is straightforward and cost-efficient. The cache controller determines whether or not a predicted address generated from the prefetcher exists in the cache by looking it up in the cache tags just as a normal access. Meanwhile, if it is a hit, PAC-PLRU makes changes to the node value of the cache block in its corresponding PLRU binary tree during the process of examination. The binary tree becomes the medium of information transmission over the connection between the prefetching mechanism and cache replacement policy. Therefore, PAC-PLRU does not require any additional hardware overhead, but just some operations on PLRU binary trees. This additional control logic is feasible for modern IC design utilizing the vast silicon estate available on chip. Moreover, the inspection of predicted addresses and the change of PLRU binary trees can overlap and the time overhead of PAC-PLRU is negligible.

VI. Related Work

LRU, Random, FIFO, and PLRU are the four major and widely used cache replacement policies in commercial processors. We have focused on the existing technologies in real processors in order to make our new proposal as practicable as possible. We surveyed 35 modern representative commercial processors. Table 4 summarizes the distribution of usage for each cache replacement policy used in these real commercial processors, in terms of different associativity. Based on Table 4, we can draw a conclusion that PLRU is the most widely used cache replacement policy in real-world processors, especially for cache associativity between 4-way and 16-way. This is the reason why we use the basic PLRU replacement policy in our baseline experimental system.

Table 4: Distribution of replacement policies usage based on the statistics of 35 modern representative processors. (86 caches in all)

Replacement Policy Category	Cache Associativity (N-way)							
	2-way	4-way	8-way	10-way	12-way	16-way	32-way	128-way
Random	4	15	1					
LRU	10	10	2	1	1			
PLRU		13	11		1	3		
FIFO	3	7					2	2

Notes:
1. The category of LRU policy includes LRU and true LRU. Some caches use LRU policy as described in their processor datasheets, but the implementation might be PLRU or some other variants. However, without loss of generality, we still count this case in the category of LRU policy.
2. The category of PLRU policy includes Pseudo LRU, Approx. LRU, Quasi LRU and NRU.
3. Some caches can be configured using more than one replacement policy by a hardware-dependent register. In this case, we count every possible policy into corresponding category in this table.

In Table 5 (a) and (b), we give the comparison of different cache replacement policies in terms of hardware storage and operation complexity.

Table 5: Complexity Comparison of Different Cache Replacement Policies [1, 9]

(a) Storage

Policies	Storage requirements (bits)
LRU	$S \cdot \lceil \log_2(N!) \rceil$
FIFO	$S \cdot \log_2(N)$
Random (LFSR)	$\log_2(N)$
PLRU (tree-based)	$S \cdot (N-1)$
PAC-PLRU (Our Proposal)	$S \cdot (N-1)$

(b) Operation

Policies	Action on cache hits	Action on cache misses
LRU	Update the LRU stack (Read Op. + Write Op.)	Update the LRU stack (Read Op. + Write Op.)
FIFO	No Operation	Increment FIFO counter
Random (LFSR)	No Operation	Update LFSR register
PLRU (tree-based)	Update the tree bit(s) (Write Op.)	Update the tree bit(s) (Read Op. + Write Op.)
PAC-PLRU (Our Proposal)	Update the tree bit(s) on real hits and prediction hits (Write Op.)	Update the tree bit(s) (Read Op. + Write Op.)

Notes:
1. S is the number of Cache Sets. N is the number of ways.
2. "Op." stands for operation.

Based on the LRU entry in Table 5 (a), the statement we made in Subsection 2.1 that LRU is costly to implement in hardware for high-associative cache can be further proved. A 4-way set associative cache must have five storage bits for each cache set to represent the 24 (= 4!) possible states of the cache blocks usage, since 24 states require five bits to encode. Similarly, an 8-way cache would require 16 bits to store the LRU status for each cache set; a 16-way cache

requires 45 bits; a 32-way cache needs 118 bits. In real processors, such as UltraSPARC T2 [26] and MIPS32 1004K [21], even six LRU status bits are maintained for each cache set in a 4-way data cache. However, compared with LRU, a 32-way cache using PLRU policy need only 31 bits for each cache set.

Another problem of both LRU and PLRU policy is that they have to update storage information (Read and/or Write Op) on both cache hits and misses, while FIFO and Random policy needs no operation on cache hits. Furthermore, LRU policy needs a read operation and a write operation on both cache hits and misses. But PLRU needs only one write operation on cache hits.

Our proposed PAC-PLRU has the same hardware storage cost as the basic PLRU, but needs some operations on PLRU binary trees when the predicted addresses hit in the cache.

VII. Conclusion and Future Work

In this study, we propose the PAC-PLRU cache replacement policy. PAC-PLRU not only utilizes but also judiciously salvages the prediction information discarded from a widely-adopted stride prefetcher. The main idea behind PAC-PLRU is utilizing the prediction results generated by the existing stride prefetcher and preventing these predicted cache blocks from being replaced in the near future if they are already in cache. Changing the PLRU binary tree of these predicted blocks is required to implement PAC-PLRU.

Extensive simulation results show that the proposed PAC-PLRU replacement policy is very promising in fostering the connection between PLRU and stride prefetcher, and it will have a lasting impact on improving overall cache performance. In addition, PAC-PLRU utilizes the prefetcher already available on processor chips to enhance the cache replacement policy, so that any concerns about the budget and cost of a dedicated prefetcher can be ignored. Last but not least, this design is independent of the prefetcher, which means it can benefit from any prefetcher including future advanced ones. Our future work includes the exploration of dynamically changing the confidence level in PAC-PLRU, and adaptively turning off PAC-PLRU, depending on specific access patterns.

Acknowledgment

We are thankful to the anonymous reviewers for their valuable suggestions to further improve this work. We thank Dr. Zhiling Lan for providing the benchmark suite, Dr. Hassan Ghasemzadeh for contributions to the cache simulator. We also thank Dr. Lixin Zhang and Dr. Mingyu Chen at Institute of Computing Technology of CAS, and SCS Group members at Illinois Institute of Technology for their insightful comments. This research was supported in part by the China Scholarship Council under CSC grant 2008100631, and by the Illinois Institute of Technology Graduate Dean Scholarship.

References

[1] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. in *Proceedings of the 42nd Annual Southeast Regional Conference*, April 2004.

[2] AMD Inc. Software Optimization Guide for AMD Family 10h and 12h Processors, Revision 3.12. http://support.amd.com/us/Processor_TechDocs/40546.pdf, pp. 81-89, December 2010.

[3] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC '91)*, November 1991.

[4] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*, December 2009.

[5] Y. Chen, H. Zhu, and X.-H. Sun. An Adaptive Data Prefetcher for High-Performance Processors. in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid-2010)*, May 2010.

[6] J. Doweck. Intel Smart Memory Access and the Energy-Efficient Performance of the Intel Core Microarchitecture. http://download.intel.com/technology/architecture/sma.pdf, 2006.

[7] DPC committee. The 1st JILP Data Prefetching Championship (DPC-1). http://www.jilp.org/dpc/, 2009.

[8] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://pages.cs.wisc.edu/~markhill/DineroIV/.

[9] J. Handy, The Cache Memory Book, Academic Press, San Diego, 1998.

[10] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium® 4 Processor. *Intel Technology Journal*, (First Quarter) 2001.

[11] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*, May 2002.

[12] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. in *Proceedings of Fourth Annual Workshop on Modeling, Benchmarking and Simulation*, June 2008.

[13] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*, June 2010.

[14] M. Kampe, P. Stenström, and M. Dubois. Self-correcting LRU replacement policies. in *Proceedings of the 1st Conference on Computing Frontiers (CF '04)*, April 2004.

[15] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, June/July 2001.

[16] H. Q. Le, W. J. Starke, J. S. Fields, J. S. O'Connell, D. Q. Nguyen, B. J. Ronchetti, B. J. Sauer, E. M. Schwarz, et al. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639-662, November 2007.

[17] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352-1361, December 2001.

[18] H. Liu, M. Ferdman, H. Jaehyuk, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. in *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture (MICRO '08)*, November 2008.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, et al. Pin: building customized program analysis tools with dynamic instrumentation. in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[20] A. Malamy, R. N. Patel, and N. M. Hayes. Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature. United States Patent US5353425, October 1994.

[21] MIPS. MIPS32® 1004K® CPU Family Software User's Manual, Revision 01.10. https://www.mips.com/application/login/login.dot?product_name=/auth/MD00622-2B-CMP-SUM-01.10.pdf, pp. 265-273, July 2009.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, October 2002.

[23] K. So and R. N. Rechtschaffen. Cache Operations by MRU Change. *IEEE Transactions on Computers*, 37(6):700-709, June 1988.

[24] W. E. Speight and L. Zhang. Cache Directed Sequential Prefetch. (Application Publication No.: US 2010/0030973 A1), Feburary 2010.

[25] C. D. Spradling. SPEC CPU2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 35(1):130-134, March 2007.

[26] Sun Microsystems, Inc. UltraSPARC T2 supplement to UltraSPARC Architecture 2007 Specification, Hyperprivileged, Draft D1.4.3. http://opensparc-t2.sunsource.net/specs/UST2-UASuppl-current-draft-HP-EXT.pdf, pp. 937-940, September 2007.

[27] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. in *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA '00)*, January 2000.

[28] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20-24, March 1995.

[29] H. Zhu, Y. Chen, and X.-H. Sun. Timing local streams: improving timeliness in data prefetching. in *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*, June 2010.