# PriorityLRU: A Frequency and Recency Based Cache Replacement Policy

Hang Ruan, Zhiqian Wang, Shan Han
George Washington University
{ruanhang|sgshan3|wangzhiq}@gwu.edu

## ABSTRACT

In this paper we introduce the basic algorithms of different cache replacement policies(CRU), and then we extend a new algorithm named priorityLRU from the basic least recently used(LRU) algorithm. Finally, we use SimpleScalar and Instructor Benchmark to test the new algorithm and it shows an obvious improvement of the cache hit ratio.

### Keywords

priorityLRU, CRU, SimpleScalar, Benchmark, LRU, replacement policy

## INTRODUCTION

A memory system is a hierarchy of storage devices with different capacities, costs, and access times. Because the higher level storage is faster, a program tends to access the storage at any higher level more frequently than they access the storage at the next lower level. In order to do that, we prefer to move the data from lower level storage into the higher level storage so that the higher level storage is the cache of the lower one.

However, the capacity of higher level storage is limited. We can't store all the data in the higher level storage, so the ways have to be replaced when cache is full. CRP is widely used to manage full cache situation. CRP decides which cache page to be the victim and to be evicted while cache is full and new page is referenced. Frequency and recency are usually considered as criteria to design replacement policy. We will introduce the LRU and LFU in next part.

The standard to evaluate the performance of a CRP is hit ratio. When CPU tries to access data or instructions, it would check cache first. If the data or instructions are available, it is known as cache hit, otherwise, it is miss. Hit ratio is defined as the percentage of hit and total times of accessing cache.

The main contribution of this paper is to extend a new CRU based on LRU. Inspired from the 2-bits branch predictor, we add two bits for each way of cache to record the frequency, which combines the advantages of LRU and SCORE. We use SimpleScalar and Instructor Benchmark to test the performance of our new replacement policy with other replacement policies. An improvement of the hit ratio is expected.

## COMMON CRP

Based on our previous research, we studied the most mentioned and the most commonly used replacement policies. Here are the simple descriptions of those policies.

### FIFO & LIFO:

The FIFO replacement policy is always to evict the first element entered in the cache when the cache is filled. There are some problems with this policy. If the data were required by CPU periodically, cache miss will always occurs. The LIFO replacement policy is always to evict the last element entered in the cache when the cache is filled. There are also some problems with this policy. After the cache is filled, if there are two different elements required by CPU periodically, it will always be cache miss. As a result, most of the space in the cache was wasted.

**LRU:**
Least recently used (LRU) policy selects victim page and updates a new way according to the recency. Policy ranks ways in the cache. The most recently used page is head of the cache and the least recently used page is tail. Considering the most recently used way is more likely to be accessed in future, when cache is full, LRU way would be evicted. Then the updated way is ranked at head of cache. The time complexity of this policy is O (1).

**OPT:**
Optimal replacement policy is a theoretical policy with high hit ratio. When cache is full, the policy would evict way which will not be referred in the future and update the old way. For example, the policy evaluates all ways and find way which will not be used in 6 seconds and delete it from cache. Although the policy has positive hit ratio, predicting the future is almost impossible. The policy will never be implemented in real life. However, the policy can be used as standard for evaluating other policies.

**SCORE:**
SCORE replacement policy is a frequency based policy. It assigns every way an initial score when the way is referenced. Next, if way gets hit, its score will increase. Otherwise the score will decrease. The policy takes score as criteria for predicting the future. Way with lowest score has lowest possibility of being used again. Therefore, the way with lowest score is the victim way in SCORE policy.

**ARC:**
ARC is developed by IBM and it has been proved to have a better performance than LRU algorithm. Compared with the traditional LRU algorithm, the main idea of the ARC algorithm is to implement the reference frequency to LRU algorithm, which is designed by splitting the cache directory into 4 parts: T1, T2, B1, and B2.

1. T1 is designed for recent cache entries.
2. T2 is designed for frequent entries.
3. B1 is the ghost entries recently evicted from T1.
4. B2 is the ghost entries recently evicted from T2.

The replacement of ARC will work as following:

New entry will be recorded at the top of T1 and eventually be evicted into the top of B1. When the B1 gets full, older entry will be evicted first. Any entry that is referenced again in T1 will be moved to the top of T2 and eventually be evicted into the top of B2. Any entry that is referenced again in T2 will be moved to the top of T2. When the B2 gets full, the older entry will be replaced. When B1 gets a hit, the size of T1 will be increased, the last entry in T2 will be evicted into B2. When B2 gets a hit, the size of T2 will be increase and the last of entry of T1 will be evicted into B1.

Our research on the other policies showed that most complex replacement policies are designed based on LRU. So, we also tried to design a replacement policy based on LRU. Unlike the complex policy ARC which modify the LRU by partition the cache, we modify the LRU by adding frequency priority rule to make the victim process more efficient and accurate.

## PROPOSED POLICY
Existing replacement policies try to predict possibility of reusability of ways and evict way with the lowest possibility. For example, OPT policy results high hit ratio because it predicts the possibility accurately. Increasing accuracy is the key idea of designing new policy.

The new policy combines recency and frequency. Similar to LRU policy, each set of cache contains a stack which called recency stack. This stack is used to record recency of all ways within the set. The top of stack means the most recently used way, the bottom of stack stores the least recently used way. Every time a way is accessed, it is moved to the top.

In addition to, an extra stack which is called priority stack is implemented. It stores 2 bits which mean the priority of the way. An abstract cache controller monitors two stacks. When cache set is full and new block is going to be cached, controller would evict the way which located at the bottom of recency stack and which has the lowest priority.

### 1. Initialisation of recency and priority

When an incoming block is placed in cache, it is treated as the most recently used. Corresponding way would be at the top of recency stack. As time goes, the least recently used way would be put at the bottom. Furthermore, default priority of way is '01' in binary. It means that the way is cached for the first time.

### 2. Changes of recency and priority

While there is a cache hit, the corresponding way would be moved to the top of recency stack. Meanwhile, its priority is increased by 1. Besides, the maximum and the minimum priority are 3(11) and 0(00). However, priority might be dropped in particular situation, details will be listed in next part.

### 3. Selection of victim way

Victim way would be selected when cache or set is full. Cache controller evaluates four ways at the bottom of recency stack, and concoller selects way with the lowest priority and evict it. If priorities are equal, LRU way would be selected as victim. In addition priorities of rest of ways are decreased by one, because they were considered as victims and gaven one more chance to stay. On the other hand, many ways with 3 priority may be placed at the bottom of stack. Otherwise, priority could lose effect without decreasing and the policy would be same as LRU policy.

| | |
|---|---|
| Space requirement[bit] | sets*ways*(log2(ways)+2) |
| Action on cache hit | Update LRU and priority stack |
| Action on cache miss | Update LRU and priority stack |

### 4. Extension

The maximum value of priority can be changed. It means that more bits would be stored at priority stack. Then more ways would be considered as victims. For example, maximum priority is $2^n - 1$ and each way has n bits priority, furthermore $2^n$ ways are evaluated for selecting victim. However, there is a tradeoff. Although more bits may reduce more miss rate, it also leads to the cost of more space and more time of calculation.

At implementation, we used 2 bits priority.

## SIMULATION TOOLS & BENCHMARKS

To simulate the cache structures and the replacement policies, we choose SimpleScalar as our simulation tool. To test how our replacement policy perform compared with other replacement policies, we choose the Instructor Benchmarks that is provided on the SimpleScalar Benchmark website.

### SimpleScalar:

SimpleScalar is a system software infrastructure that is built fo4r modeling application performance, micro-architectural modeling, and hardware-software co-verification. SimpleScalar includes sample simulator such as dynamically scheduled processor model, non-block caches simulation, and state-of-the-art branch prediction. SimpleScalar can emulate the PISA, ARM, Alpha, and x86 instruction sets.

To install the SimpleScalar, we installed a Ubuntu system on the Oracle Virtualbox. Using SimpleScalar we were able to build a cache structure and write our designed CRP into the simulator by modifying files cache.h and cache.c.

### Instructor Benchmarks:

To test the performance of our replacement policy, we research all the commonly used benchmarks for example SPEC CPU series. However most of the benchmarks need to be purchased for a price over $1000. So, we end up using the Instructor Benchmark set which is included in the SimpleScalar website, which is a custom designed benchmark set for education purpose from SompleScalar LLC.

This Instructor Benchamark is derived from SPEC2000 for computer architecture course instruction. It includes binaries for Alpha and

PISA instructions sets, inputs, and reference outputs. This benchmark suite will simulate a wide range of instructions with a short running experiment from 100 to 250 million instructions. In our test, we only used Alpha instructions sets.

With our SimpleScalar and Instructor Benchmarks, we were able to simulate our priorityLRU policy and other replacement policy such as LRU, FIFO, and random replacement policy. The results and analysis are provided in next section.

## RESULTS & ANALYSIS

SimpleScalar provides 2 levels cache hierarchy structure. Each level includes data cache and instruction cache.
SimpleScalar supports shared cache structure, data and instructions can be located at same level 2 cache.

For analysing performance of new police in data cache and instruction cache, we used private cache which means that data and instructions are cached at different level 2 caches.

For simplifying test, we chose 128 sets for level 2 cache and 64 sets for level 1 cache. Because more sets costs more time to execute.

In addition to, a significant number of ways could result a miss rate that is too small, even close to constant, which is too difficult to analyse. Besides, we found that miss rate of priorityLRU and LRU are the same when associate is greater than 64. It means that 64 can be the threshold. Then 1, 4, 8, 16, 32, 64 are chosen as associates. SimpleScalar has FIFO, LRU and Random policies and all of them are compared with priorityLRU.

Due to PriorityLRU equals to LRU when associate is less than or equal to 4, we ignored those part of data. Furthermore, algorithms of LRU and FIFO are the same besides FIFO do not rank ways in SimpleScalar and they had same test results, we omit FIFO from test results.

Figure 1 and 2 show miss rate of data cache in level 1(DL1) and level 2(DL2). PriorityLRU resulted less miss rate than LRU in DL1. When associate was 8, improvement was 4.9% which is highest. Average improvement was 1.9%. However, miss rate of priorityLRU in DL2 increased.

Figure 3 and 4 display miss rate of instruction cache in level 1(IL1) and level 2(IL2). Similar to DL1 and DL2, priorityLRU provided lower miss rate in level 1 but higher miss rate in level 2. The highest improvement was 2% when associate was 8. Average improvement was 2%.

In miss rate aspect, although new policy does improve hit ratio in level 1 cache, it failed in level 2 cache. The possible explanation is less access times in level 2 cache. Miss rate is calculated by miss frequency divided by access frequency.

$$missRate = Frequence(miss)/Frequence(access)$$

In priorityLRU policy, because miss rate is decreased in level 1 cache, frequency of accessing level 2 cache is also decreased. Level 2 cache has more set, it can store more data, then change of miss frequency is small, it means that new policy can not improve level 2 cache effectively.

$$missRate = (Frequence(miss) - a)/(Frequence(access) - b)$$
$$a << b$$

Hence, decrease ratio of miss is less than access, it leads to higher miss rate in level 2 cache.

Figure 5 illustrates time of execution. Execution time of priorityLRU is greater than LRU at first because priorityLRU chooses victim by comparing 4 ways. However, access frequency of level 2 cache drop, then cost of execution time decreases.

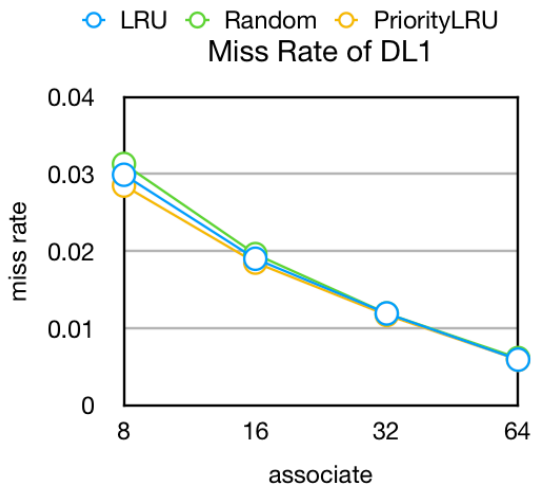


**Figure 1**

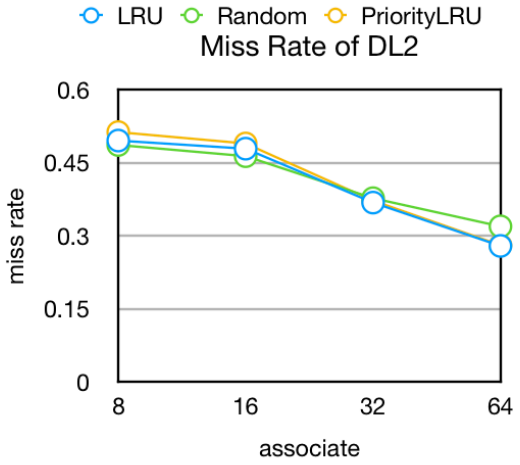


**Figure 2**

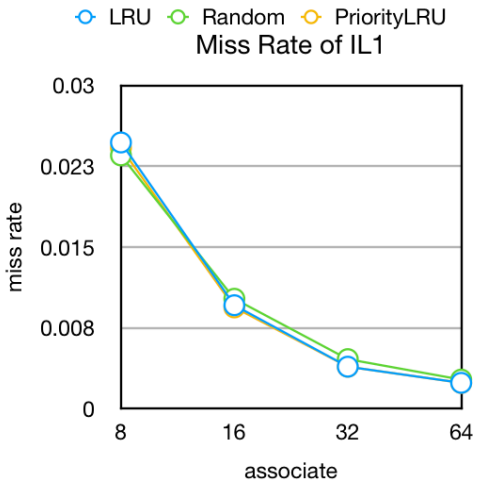


**Figure 3**

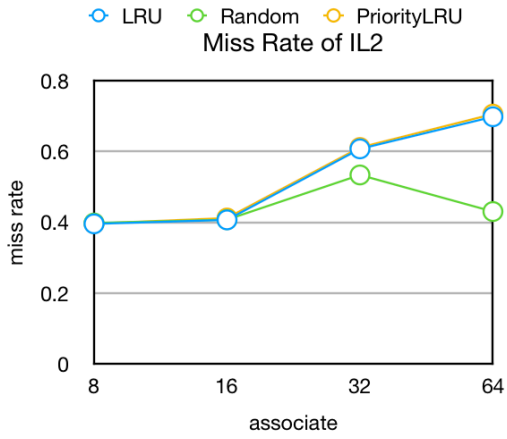


**Figure 4**

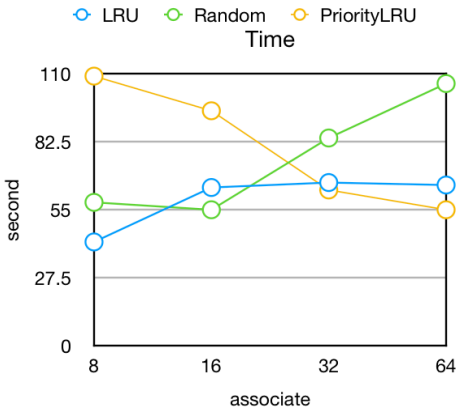


**Figure 5**

## CONCLUSION

Our algorithm combined the LRU and SCORE, which means it consider both the recency and frequency.

By adding 2-bits to record the frequency of a way were used within a period, we significantly decreased the miss rate of priorityLRU compared with LRU. The optimal result was reached when the associate was 8 and the improvement of the hit ratio was 4.9%. Because when the cache is full, priorityLRU will consider four ways of the least recent used and to compare their 2-bits priority and select the smaller one as the victim to evict it.

As the associate increase, the time of the priorityLRU tends to decrease and the time of LRU tends to increase. At certain time, the time of priorityLRU will be less than the time of LRU.

## REFERENCE


1. S. Jiang and X. Zhang, “LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance,” in Proc. ACM SIGMETRICS Conf., 2002

2. N. Duong., R. Cammarota., D. Zhao., T. Kim., A. Veridenbaum, “SCORE: A Score-Based Memory Cache Replacement Policy.” In: Emer, J. (ed.) JWAC 2010 - 1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship, Saint Malo, France 2010

3. R. Butt., C. Gniady and Y. C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms," in IEEE Transactions on Computers, vol. 56, no. 7, pp. 889-908, July 2007.

4. N. Megiddo., D. S. Modha, “ARC: A Self-Tuning, Low Overhead Replacement Cache,” Proceedings of the 2nd USENIX Conference on File and Storage Technologies, March 31-31, 2003, San Francisco, CA

5. Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing"

6. Moinuddin K. Qureshi, Yale N. Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches”.

7. Dybdahl, H., Stenström, P. and Natvig, L. (2007). An LRU-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. ACM SIGARCH Computer Architecture News, 35(4), p.45.

8. Gu, X. and Ding, C. (2011). On the theory and potential of LRU-MRU collaborative cache management. ACM SIGPLAN Notices, 46(11), p.43.

9. D. Burger., T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", http://www.simplescalar.com/docs/users_guide_v2.pdf

10. D. Burger., T. M. Austin, “The SimpleScalar Tool Set, Version 3.0,” University of Wisconsin-Madison Technical Report #1342, June 1997.

11. G Yadgar, M Factor, A Schuster, "Karma: know-it-all replacement for a multilevel cache[C]", *Usenix Conference on File and Storage Technologies FAST 2007*, pp. 169-184, February 13–16, 2007.

12. H. Al-Zoubi, A. Milenkovic, M. Milenkovic, "Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite", *Proceeding ACM-SE 42 Proceedings of the 42nd annual Southeast regional conference*, pp. 267-272.


## APPENDIX

How to install and run SimpleScalar and Instructor Benchmark:

1. Download SimpleScalar and Instructor Benchmark at www.simplescalar.com
Files:
simplesim-3v0e.tgz
simpletools-2v0.tgz
simpleutils-2v0.tgz
Instructor benchmarks

2. decompress those files.

3. Use command:

cd $IDIR/binutils-2.5.2

```
        configure --host=$HOST --
target=ssbig-na-sstrix --with-gnu-as
                --with-gnu-ld --prefix=$IDIR
        make
        make install
```
$HOST should be “i386-*-gnu/linux” and $DIR should be  “../”

4. Use command:

```
cd $IDIR/simplesim-3.0
        make
```

5. Use command:

```
        cd $IDIR/gcc-2.6.3
        configure --host=$HOST --
target=ssbig-na-sstrix --with-gnu-as
                --with-gnu-ld --prefix=$IDIR
        make LANGUAGES=c
        ../simplesim-3.0/sim-safe ./enquire -f
>! float.h-cross
        make install
```

HOST should be “i386-*-gnu/linux” and $DIR should be  “../”

6. Replace “cache.c” and “cache.h” file in simplesim-3.0 directory.

7. Use command:

```
cd simplesim-3.0
./sim-cache -cache:dl1 dl1:64:32:64:l -
cache:il1 il1:64:32:64:l -cache:dl2
dl2:128:32:64:l -cache:il2 il2:128:32:64:l
benchmarks/cc1.alpha -O benchmarks/1stmt.i
```

For cache configure “-cache”, there are 5 parameter:
-cache <name> <nsets> <sbloc> <assoc> <repl>
<name>: cache name
<nsets> sets#
<sbloc> size of block
<assoc> associativity
<repl> replacement policy:
l - LRU, f - FIFO, r - RANDOM, p - priorityLRU