High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)

Guanxiong Liu, Xiaomeng Chen, Yuhui Zheng

Abstraction

Practical cache replacement policies attempt to emulate optimal replacement by predicting the re-reference interval of a cache block. We compared LRU, SRRIP, and DRRIP in our work, and shown that RRIP may not be a general solution to all benchmarks, which have workset larger than cache size.

Keywords: Cache Replacement Policy, RRIP, LRU

I Introduction

The goal of this project is to implement RRIP(Re-Reference Interval Prediction) cache replacement policy in LLC(Last Level Cache), and verify results in [1]. RRIP is designed to gain high cache replacement performance on workload, whose working-set is larger than the cache size, or frequent referencing to non-temporal data. This kind of workload, common in real world applications, has two types of cache access pattern: Thrashing Access Pattern, and Mixed Access Pattern.

Comparing to LRU and DIP, RRIP is supposed to solve the problem by dynamically learning re-reference information for each block in the cache access pattern. Traditionally, LRU [3] replacement policy has limited performance when re-references only occur in the distant future in Thrashing Access Patterns. Meanwhile, Dynamic Insertion Policy (DIP) [2] improves performance of LRU when re-reference interval is in the distant future by dynamically changing the re-reference prediction from a near-immediate re-reference interval to a distant re-reference interval, but it still makes the same prediction for all references of a workload.

In this paper, we first introduce RRIP algorithms in section II. Then we specify our experimental methodology in section III. The evaluation results on static RRIP with different re-reference prediction value(M) are in section IV. The results are compared to LRU, with the same CPU configuration. Meanwhile, we proceed to dynamic RRIP, and got partial simulation results by the deadline of this paper. In section V, we conclude our simulation results. For more details about implementation, one could refer Appendix.

II RRIP Algorithm

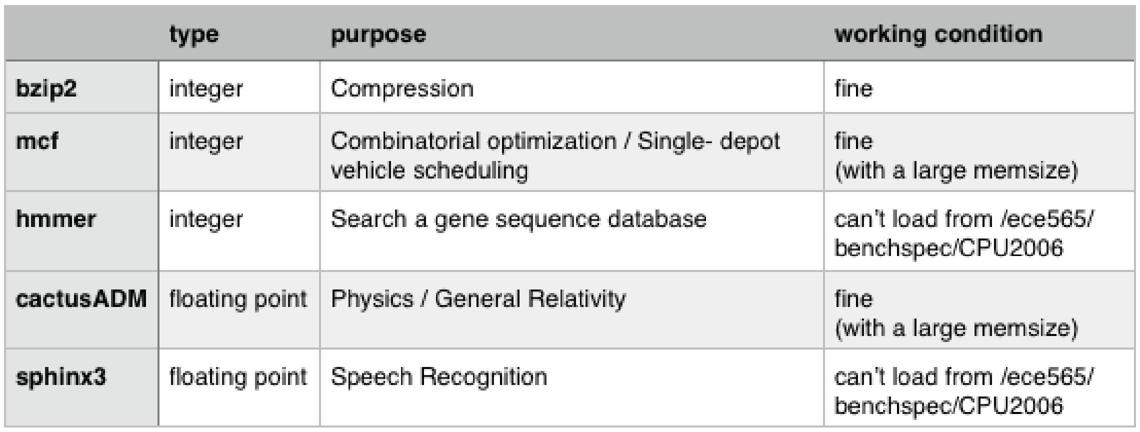
There are two kinds of RRIP algorithms: static RRIP and dynamic RRIP. SRRIP uses M bits per cache block to store one of 2^M possible Re-reference Prediction Values. Compared to NRU, SRRIP enables intermediate re-reference intervals that are greater than a near-immediate re-reference interval but less than a distant re-reference interval. SRRIP also implements a hit promotion policy, either with hit priority or frequency priority, to dynamically improve the accuracy of the predicted re-reference interval of cache blocks. Since SRRIP inefficiently utilizes the cache when the re-reference interval of all blocks is larger than the available cache, Dynamic Re-reference Interval Prediction is proposed. DRRIP uses Set Dueling to choose between scan-resistant SRRIP and thrash-resistant BRRIP.

III Experimental Methodology

>> baseline configuration

In order to produce similar results in [1], we keep strictly the same CPU configurations as [1]. For core side, we used 1 core, 128-entry ROB, and 4-way issue. For cache side, we used L1($I and $D) (4way / 64B / 32KB), L2(8way / 64B / 256KB), and L3(16way / 64B / 2MB). Also each layer of cache has 32 mshrs. However, the major difference of our baseline from [1] is, that we are using ALPHA, while [1] is using X86 ISA. This may cause a different miss/hit number, but qualitatively X86 and ALPHA should have a similar trend of changing miss/hit number as replacement policy changes.

>> benchmark selection
Since we only have spec2006 to work with, our benchmarks are limited to bzip2, mcf, cactusADM, hmmer, sphinx3. These workloads were chosen in [1], because they are sensitive to memory latency on the baseline processor configuration and there is an opportunity to improve their performance through enhanced replacement decisions. According to Table 1, bzip2, mcf, and cactusADM are chosen to evaluate our RRIP.



	type	purpose	working condition
bzip2	integer	Compression	fine
mcf	integer	Combinatorial optimization / Single- depot vehicle scheduling	fine (with a large memsize)
hmmer	integer	Search a gene sequence database	can't load from /ece565/ benchspec/CPU2006
cactusADM	floating point	Physics / General Relativity	fine (with a large memsize)
sphinx3	floating point	Speech Recognition	can't load from /ece565/ benchspec/CPU2006

Table 1. benchmarks

>> simulation environment
As RRIP cache replacement policy is mainly targeting at decreasing conflict misses in large-workset programs, there needs to warmup caches before collecting data, to avoid counting compulsory misses into total misses. Also should notice that, every replacement policy could be affected by cache configuration, for large cache size could avoid capacity misses in caches. Since we are only interested in performance improvement brought by RRIP, we fixed cache configuration(cache associativity/block size/capacity/mshr) throughout the experiment, and only changed re-reference prediction value(M) accordingly.

IV Simulation Result

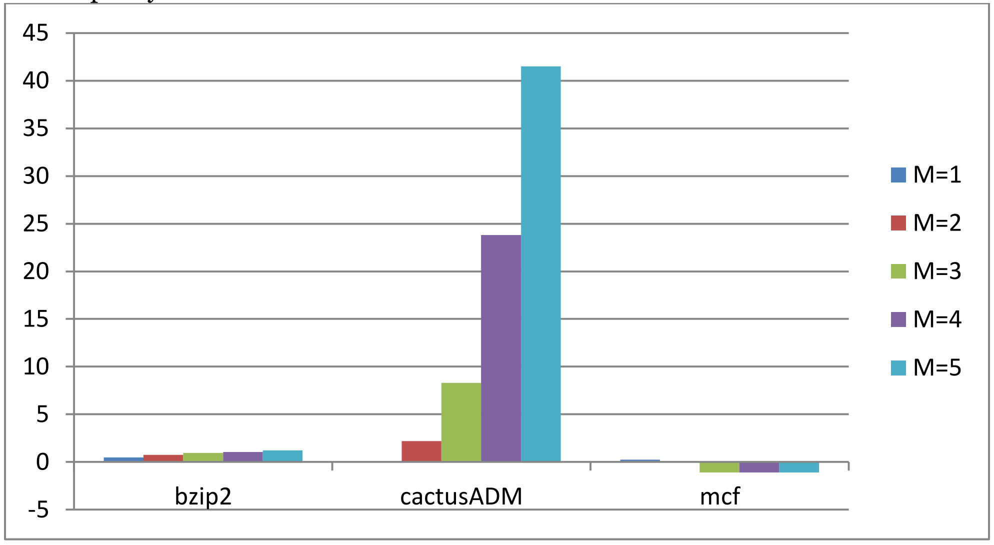


Figure 1 CPI increase (%) for SRRIP compared to LRU

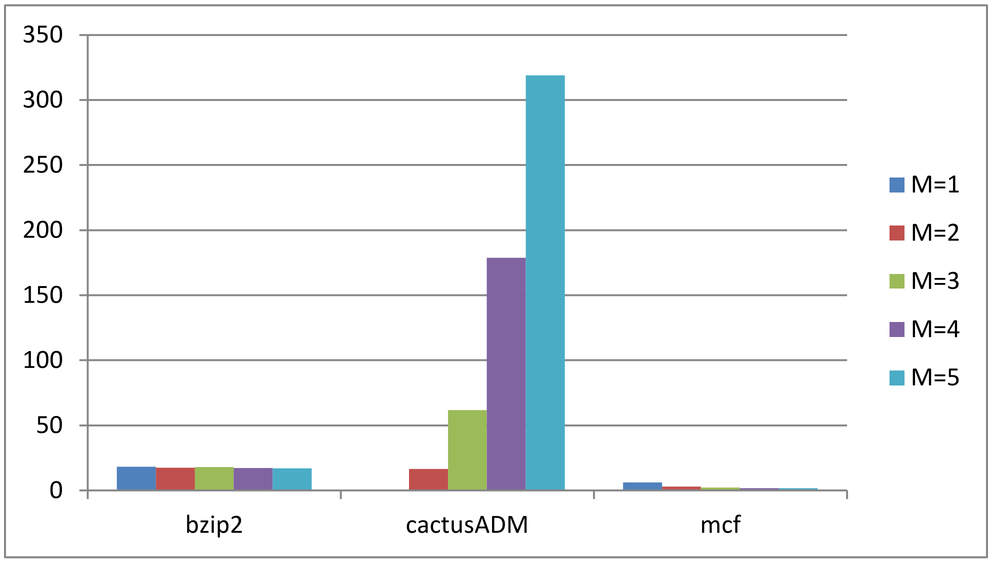


Figure 2 miss rate increase (%) for SRRIP compared to LRU

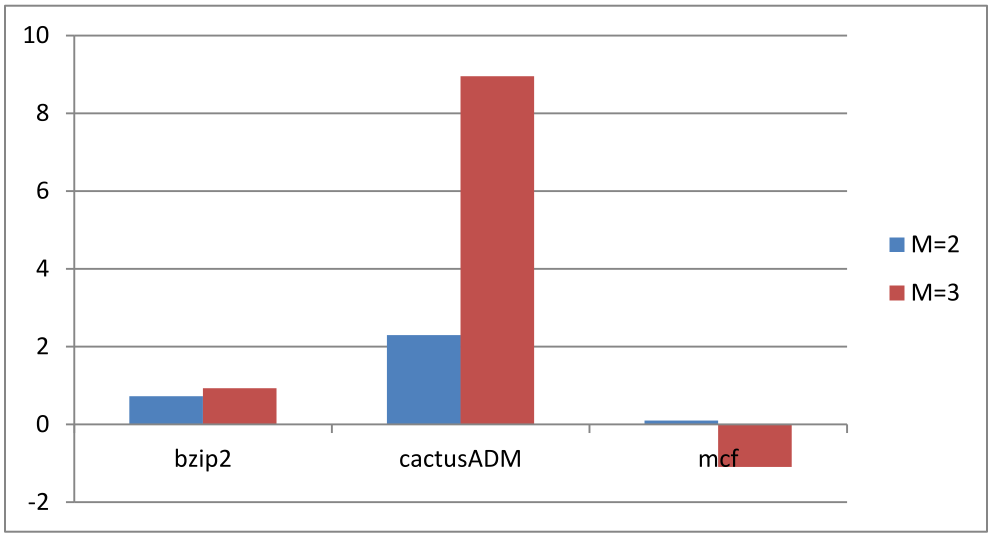


Figure 3 CPI increase (%) for DRRIP compared to LRU

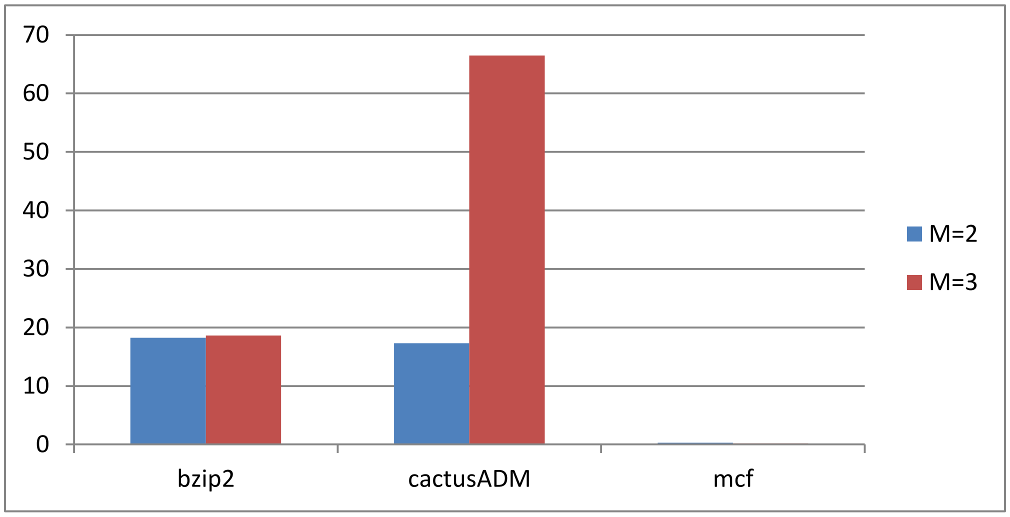
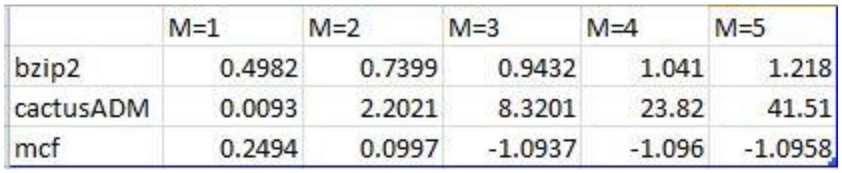


Figure 4 miss rate increase (%) for DRRIP compared to LRU

$$CPI\ increase\ (\%) = \frac{new\ CPI - CPI\ \ for\ LRU}{CPI\ \ for\ LRU} \times 100\%$$

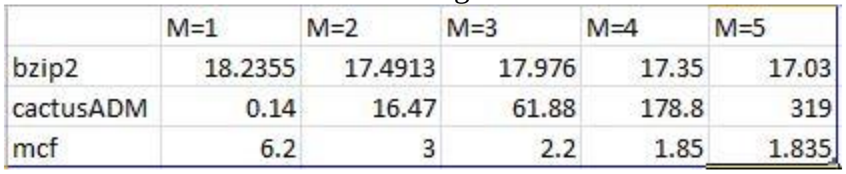
$$miss\ rate\ increase\ (\%) = \frac{new\ miss\ rate - miss\ rate\ \ for\ LRU}{miss\ rate\ \ for\ LRU} \times 100\%$$

Since some data in the graph is not easy to read, here is the data corresponding to above graphs:



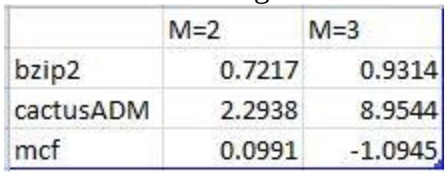
	M=1	M=2	M=3	M=4	M=5
bzip2	0.4982	0.7399	0.9432	1.041	1.218
cactusADM	0.0093	2.2021	8.3201	23.82	41.51
mcf	0.2494	0.0997	-1.0937	-1.096	-1.0958

Data for Figure 1



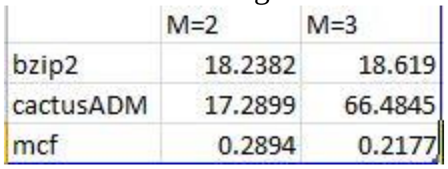
	M=1	M=2	M=3	M=4	M=5
bzip2	18.2355	17.4913	17.976	17.35	17.03
cactusADM	0.14	16.47	61.88	178.8	319
mcf	6.2	3	2.2	1.85	1.835

Data for Figure 2



	M=2	M=3
bzip2	0.7217	0.9314
cactusADM	2.2938	8.9544
mcf	0.0991	-1.0945

Data for Figure 3



	M=2	M=3
bzip2	18.2382	18.619
cactusADM	17.2899	66.4845
mcf	0.2894	0.2177

Data for Figure 4

V Conclusion

Practical cache replacement policies attempt to emulate optimal replacement by predicting the re-reference interval of a cache block. Theoretically, RRIP should generally outperform other replacement policies in miss-rate, while adding very little cache tag overhead.

Yet in our work, we've shown that:

a, RRIP may affect little to integer programs.

b, RRIP may hurt performance of some floating point programs, in a certain configuration of caches.

Both results may be correct due to the different of ISA, or some other leading facts, which may not be captured by RRIP replacement policy. This needs further simulation on various benchmarks, and also on various configuration of caches.

VI References

[1] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., Joel Emer. "High Performance Cache Replacement Using Re-Reference Interval Prediction(RRIP)". ISCA'10, June19-23, 2010
[2] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, J. Emer. "Adaptive Insertion Policies for High Performance Caching". ISCA-34, 2007
[3] E. J. O'Neil, P. E. O'Neil, G. Weikum. "The LRU-K page replacement algorithm for database disk buffering", in Proc. ACM SIGMOD Conf., pp. 297-306, 1993
[4] "SPEC CPU2006 Benchmark Descriptions"

VII Appendix -- Implementation Details

Our work on RRIP consists of 2 parts:
1, Modify original implementation
2, Implement RRIP tags(SRRIP/DRRIP)
Details are as below.

1, Modification of original implementation

a) Add variable *rrpv* in CacheBlk class.

b) Add variable *policy* in CacheSet class labeling what replacement policy this cache set uses. For SRRIP, *policy* can only be SRRIP. For DRRIP, *policy* can SRRIP, BRRIP and Follower. The cache sets labeled with SRRIP and BRRIP are sampled sets implementing SRRIP and BRRIP replacement policy respectively. Their performance decide the replacement policy of the cache sets labeled by Follower according to the Set Dueling mechanism in [2].

c) Add variable psel (Policy Selector) for the whole cache system, which is a saturating counter, keeps track of which of the two sampled sets mentioned in 2) incurs fewer misses.

d) Add two functions CachBlk* findDistantRFBlk() and CachBlk* findLongRFBlk() in CachSet class which respectively find the distant-reference block and long-reference block in one cache set.

e) Add RRIP class, implementing the similar functions as that in URL class. The important functions include accessBlock() to access block and update replacement, findVictim() to find a block to evict for the addresss provided and inserBlock() to insert the new block into the cache. We changed accessBlock() and findVictim() to implement RRIP algorithm.

f) Add three constants rrpvSize, pselSize, and epsilon. rrpvSize if the number of possible rrpv value, equal to 2^M in [1]. pselSize is the maximum value of psel. epsilon is the probability to use BRRIP in DRRIP alorithm

2. Implementation of SRRIP *

```
accessBlock()
//input: address // address of the memory address to be accessed
//input: sets // cache set array
//output: The matched cache block if found, or Null otherwise
tag = extractTag(address)
set = extractSet(address)
```

```
block = sets[set].findBlock(tag)
if(block != NULL) // cache hit
    block->rrpv = 0 //set rrpv of the hitted block to 0.
else // cache miss
    do nothing
return block
```

```
findVictim()
//input: address // address of the memory address to be accessed
//input: sets // cache set array
//output: The victim cache block
set = extractSet(address)
block = sets[set].findDistantFRBlk() // Replace the block with distant-reference.
block - > rrpv = rrpvSize - 2 // Make the victim block as a long-reference block.
return block
```

3.Implementation of DRRIP

```
Initialize() // Initialize caches sets with assign replacement policies to each set
//input: sdmSets = 32 // Sample set number for each policy in set dueling mechanism
//input: numSets // Number of the total sets
//input: sets // cache set array
sampleInterval = numSets / sdmSets
for i : 0 -> numSets
    if(i % sampleInterval == 0)
        sets[i].policy = SRRIP
    else if(i % sampleInterval == 1)
        sets[i].policy = BRRIP
    else
        sets[i].policy = Follower
```

```
accessBlock()
//Same as that in SRRIP
```

```
findVictim()
//input: address // address of the memory address to be accessed
//input: sets // cache set array
//output: The victim cache block
set = extractSet(address)
if(sets[set].policy == SRRIP)
    block = sets[set].findDistantFRBlk()
    psel ++
if(sets[set].policy == DRRIP)
    if(rand() < epsilon) // Use rand() returns a random value in [0,1)
        block = sets[set].findLongFRBlk()
    else
        block = sets[set].findDistantFRBlk()
    psel --
if(sets[set].policy == Follower)
    if(psel < pselSize/2)
        block = sets[set].findDistantFRBlk()
    else
        if(rand() < epsilon)
            block = sets[set].findLongFRBlk()
        else
            block = sets[set].findDistantFRBlk()
block - > rrpv = rrpvSize - 2 // Make the victim block as a long-reference block.
return block
```