Yet Another Implementation of Re-Reference Interval Prediction (RRIP) Cache Replacement

Yukun Zeng
Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77840
Email: yzeng@tamu.edu

***Abstract*—Cache replacement policies server on the purpose of predicting memory access patterns to make most of size limited cache space to reduce lags caused by too many memory accesses. Most early cache replacement policies like LRU simply predict the hit and miss blocks will be re-referenced immediately afterwards. Certainly applications with distant re-reference interval, either due to a large working set or frequent bursts of references to non-temporal data (namely, scans), performs very badly on such policies. To properly adapt to these applications, researchers from Intel proposed a Re-Reference Interval Prediction (RRIP) based cache replacement policy, which can learn cache block re-reference patterns on-the-fly thereby providing scan and thrash resistancy. The RRIP is shown in experiments to perform significantly better comparing to LRU, and the hardware budget it requires is practical (2X less than LRU, 2.5X less than LFU). In this paper, we present the design and our implementation of one NRU and two RRIP cache replacement policies. We also show a series of experiments carried out on the provided infrasctructure in comparison to other policies like LRU and NRU.**

I. Introduction

A. Cache and Replacement Policies

Cache is an indispensable intermediate level storage in the memory hierarchy of modern computer architecture. It fills the great speed gap between high-performance processors and the main memory. Cache size is always limited due to the fact that it's too expensive, thus it can only take a small, frequently-used portion from main memory to feed the processor in a much quicker manner. Therefore, there must be some times that the processor want to access the portion of memory that is not currently available in cache (or a certain cache set it maps to), namely, this is a cache miss. Capacity miss is the case that at the meantime, the cache (or cache set) is full, which means we have to drop one block in the cache so that the new block can come in. Among all the miss circumstances, capacity miss is the biggest bottleneck that influences cache performance. Upon a capacity miss, we ideally want new blocks of memory to get into the cache by replacing old blocks that won't be accessed in the distant future. To achieve that, we need a cache replacement policy that makes choices on which block to abandon for new blocks.

B. Motivation

Practical replacement policies are not aware of the future re-reference sequence of memory blocks, thus they usually do replacement decisions based on prediction of which memory block (currently in cache) will be accessed again in the furthest future so that they could evict it for new block with minimum price. Specifically, the most widely-used replacement policy, namely Least Recently Used (LRU), maintains a chain (basically a timeline of memory blocks reference operations) and assumes the block that is accessed the earliest in the chain will be most unlikely to be re-reference again in the near-immediate. Apparently, such a policy performs good in applications with high data locality, but it cannot extend well to thrashing applications or applications with large working sets described before. Some other policies based on LRU have been proposed recently to achieve better miss rate or deal with different applications in [1] [2] [3] [4].

In last-level cache (LLC), the accuracy of a replacement policy is crucial since a miss in the last-level means an access to main memory. As we mentioned before, pure LRU policy has several disadvantages and many studies like [5] [6] [2] [4] have illustrated that temporal locality filtering by small front-level caches cause lots of blocks that will never be re-referenced again evicted to LLC. We now study the worst cases of LRU in the following memory access patterns:

- Recency-friendly Access Patterns
- Thrashing Access Patterns
- Streaming Access Patterns
- Mixed Access Patterns

[7] has illustrated that for both thrashing and mixed access patterns, there is room to improve LRU. The key to this improvement shall rely on a better prediction on re-reference interval of different memory blocks based on historical reference information.

Though designed to describe the recency of block references, we can also regard LRU timeline as a RRIP chain in the sense that the LRU policy is actually assuming the head block of its chain will be re-referenced near-immediate while the tail will have a distant re-reference interval. According the relations between RRIP and LRU above, we can adapt LRU to more kinds of applications by applying RRIP policy into it to learn and predict the re-reference patterns of blocks on-the-fly.

II. Related Works

In computer architecture literature, cache replacement policies have been studied intensively. For the fact that our policy aims at improving LLC performance, we present a

brief review on prior art on this topic. Problems of dead blocks (blocks that are never or distantly accessed after brought into cache) is commonly solved by introducing block access frequency to predict re-reference pattern. Recency and frequency are two factors mostly used to determine the re-reference pattern of a block. Leveraging counters to record access frequencies of blocks, Least Frequently Used (LFU) [8] predicts that the block being accessed most frequently will be referenced earliest in the future. However, LFU only considers the frequency factors, it's easy to imagine that when some applications have finished and new applications come along, it will lead to continuous cache misses for a considerably long time due to the unawareness of recency. To address the problems with high recency applications, studies in [8] [9] [10] combined both factors of recency and frequency but they require parameter tuning beforehand when dealing with different workloads. Self-tuning policies proposed in [11] [12] [13], though effective and adaptive to various workloads, they significantly increased the hardware complexity and overhead. On the other hand, [14] proposed to combine dead blocks predictions with replacement policies at LLC. As [2] showed that the occurence of dead blocks often happens in the case when working set is larger that cache size, Dynamic Insertion Policy (DIP) is also proposed to dynamically change insertion policy while the workloads vary. Different from using a common recency stack, pseudo-LIFO [15] uses fill stack to learn the re-reference probabilities of a block but it requires additional hardware to track fill stack positions. Similarly to pseudo-LIFO, other solutions in [16] [17] [18] [19] [20] requires significant changes or additions on current cache hardware implementation.

III. Methodology

As discussed before, in the cases of mixed access pattern and thrashing accesses, LRU's performance is greatly undermined. To improve the replacement policy in such circumstances, [RRIP] mainly proposed two Static RRIP (SRRIP) policies, one uses hit promotion strategy which predicts every hit block will be accessed in the near-immediate, the other maintains a multi-bit counter to exploit block access frequency. The original paper also has proposed a Bimodal RRIP (BRRIP) that is suitable for predicting with thrash-resistancy, it was also combined with SRRIP using Set Dueling [2] so that we can dynamically determine the type of current workloads and switch between two policies to achieve a better overall performance. However, this paper didn't go into details on BRRIP and DRRIP, thus we omit their implementation here and present the hit promotion and frequency priority strategy here, as well as the NRU policy, which serves as the similar role of benchmark as LRU.

A. Not Recently Used (NRU) Replacement

NRU replacement policy is an approximation of LRU which yields similar performance while is much simpler and requires less on hardware. Different from LRU, who maintains a chain (which is actually a linked-list) and keeps doing complex operations like insertion and switching, NRU only requires single bit for predicting the future re-reference of cache blocks. A block with nru-bit of '0' means it will be accessed in near-immediate, while nru-bit of '1' implies the block will be re-referenced after a distant interval. The detailed initialization, update and victim selection algorithm of NRU is provided in Algo 1.

Algorithm 1 NRU Replacement Policy

```
function Initialize
    setCount = number of sets in the cache
    assoc = cache associativity
    init replInfo[setCount][assoc]
    for setIndex = 0 → (setCount − 1) do
        for blockIndex = 0 → assoc do
            replInfo[setIndex][blockIndex].nru_bit = 1
        end for
    end for
end function
function VictimSelection(setIndex)
    lineReplInfo = replInfo[setIndex]
    for blockIndex = 0 → assoc do
        if lineReplInfo[blockIndex].nru_bit == 1 then
            lineReplInfo[blockIndex].nru_bit = 0
            return blockIndex
        end if
    end for
    for blockIndex = 1 → assoc do
        lineReplInfo[blockIndex].nru_bit = 1
    end for
    return 0
end function
function HitUpdate(setIndex, blockIndex, cacheHit)
    if cacheHit then
        replInfo[setIndex][blockIndex] = 0
    end if
    return 0
end function
```

B. Static RRIP (SRRIP)

While we enjoy the simplicity of NRU replacement policy based on single bit hardware, the discreteness of NRU (predicts either near-immediate or distant re-reference) greatly limits its performance for mixed access patterns because scan blocks will always be predicted that they will be re-referenced near-immediate while in fact it's not ture. To correctly identify such scans and improve scan-resistancy, the authors proposed RRIP cache replacement policy that leverages M bits to store 2^M Re-reference Prediction Values (RRPV). With decreasing RRPV values, the possibility of a block to be re-referenced in near-immediate gets bigger. Since prediction of either near-immediate or distant re-reference on a block is not robust shown in the NRU, RRIP inserts new blocks with a prediction that it would be accessed in a long re-reference interval. A long re-reference interval is defined to be $2^M - 2$. The idea is

through a long re-reference setting we can prevent distant re-reference block from getting into the cache and it also buys the policy more time to learn re-reference intervals for different blocks.

Upon a cache replacement, the victim selection algorithm of RRIP policy iterate all the blocks in order to find the first block with a distant re-reference interval, i.e., whose RRPV equals distant re-reference interval ($2^M - 1$). If such block exists, then we simply replace it and set its RRPV to long re-reference value ($2^M - 2$). If the RRPV of all blocks is below distant re-reference interval, we iteratively increase the RRPV of all blocks and search again after each increment operation.

The authors further proposed two different policies for updating RRPVs when cache hits, one named Hit Promotion while the other is based on Frequency Priority. In the following sections we briefly go through mechanisms in these two policies.

1) RRIP-Hit Promotion: RRIP hit promotion policy, abbreviated as RRIP-HP, prioritizes to replace blocks that are not receiving hits over any block that is re-referenced. When a cache hit happens, it predicts the block associated with the hit will be re-referenced in the near-immediate by setting RRPV of that block to be 0. The algorithms for RRIP-HP are given in Algo 2. Though RRIP-HP performs well in most workloads by predicting re-reference interval, it can potentially degrade performance in the case that a cache block is only referenced once after insertion (like what happens in a scan or thrash) since it will direcly set RRPV to be 0 which would make it a long time for this distant re-referenced block to be evicted.

2) RRIP-Frequency Priority: To properly address the disadvantages of RRIP-HP, we further implemented the RRIP frequency priority (RRIP-FP) policy. Since there are only some trivial changes between RRIP-HP and RRIP-FP, we omitted the detailed algorithm part of RRIP-FP. The biggest difference between RRIP-FP and RRIP-HP lies in that RRIP-FP decrements the RRPV values of hit blocks instead of setting them directly to zero (near-immediate). Apparently by applying such update policy, we will learn more information on the re-reference frequency of blocks thereby improving the performance based on RRIP-HP.

Since all the RRIP policies presented are static in the sense that they are not adaptive to different workloads, we refer them as Static Re-Reference Interval Prediction (SRRIP). A simple example provided in [7] that illustrates the behavior of RRIP in comparison with LRU and NRU policy is also given in Figure 1.

Algorithm 2 RRIP-HP REPLACEMENT POLICY

function INITIALIZE(M)
  $max\_rrpv = pow(2, M) - 1$
  $setCount$ = number of sets in the cache
  $assoc$ = cache associativity
  init $replInfo[setCount][assoc]$
  for $sIndex = 0 \rightarrow (setCount - 1)$ **do**
    for $bIndex = 0 \rightarrow assoc$ **do**
      $replInfo[sIndex][bIndex].rrpv = max\_rrpv$
    end for
  end for
end function
function VICTIMSELECTION(sIndex)
  $lineReplInfo = replInfo[sIndex]$
  $maxRrpvIndex = assoc$
  for $blockIndex = 0 \rightarrow assoc$ **do**
    if $lineReplInfo[bIndex].rrpv = max\_rrpv$ **then**
      $lineReplInfo[bIndex].rrpv = max\_rrpv - 1$
      return $bIndex$
    end if
    $curRrpv = lineReplInfo[bIndex].rrpv$
    $maxRrpv = lineReplInfo[maxRrpvIndex].rrpv$
    if $curRrpv > maxRrpv$ **then**
      $maxRrpvIndex = bIndex$
    end if
  end for
  $diff = max\_rrpv - lineReplInfo[maxRrpvIndex]$
  for $bIndex = 0 \rightarrow assoc$ **do**
    $lineReplInfo[bIndex].rrpv+ = diff$
  end for
  $lineReplInfo[maxRrpvIndex] = max\_rrpv - 2$
  return maxRrpvIndex
end function
function HITUPDATE(sIndex, bIndex, cacheHit)
  if $cacheHit$ **then**
    $replInfo[sIndex][bIndex] = 0$
  end if
  return 0
end function

IV. EXPERIMENTS

In the experiment part, we carried out a comprehensive analysis on the performance of both HP and FP RRIP policies and normalized their results based on LRU performance, we also added NRU into consideration. First, we tune the M parameter by running a series workloads on varying M values, recall that M is the number of bits we use for each cache block to store their re-reference frequency. Then we use the best fitted M value and run benchmarks to get normalized performance improvements based on LRU (in some cases the improvement might be negative which means LRU outperforms new strategies). The metrics we used for evaluation includes both MPKI and IPC.

A. RRIP Tuning

We first tune the RRIP replacement policy by setting M from 2 to 4 and comparing the results to NRU. Both the MPKI and IPC performance results of RRIP-HP and RRIP-FP are show in Figure 2. From the figure we learn that by increasing M, for some special workloads we can greatly improve the policy performance. However, for most workloads the influence is negligent and we might want a second thought since it's requesting far more hardware budget. Also we found that the simplicity of RRIP-HP policy helps it outperform

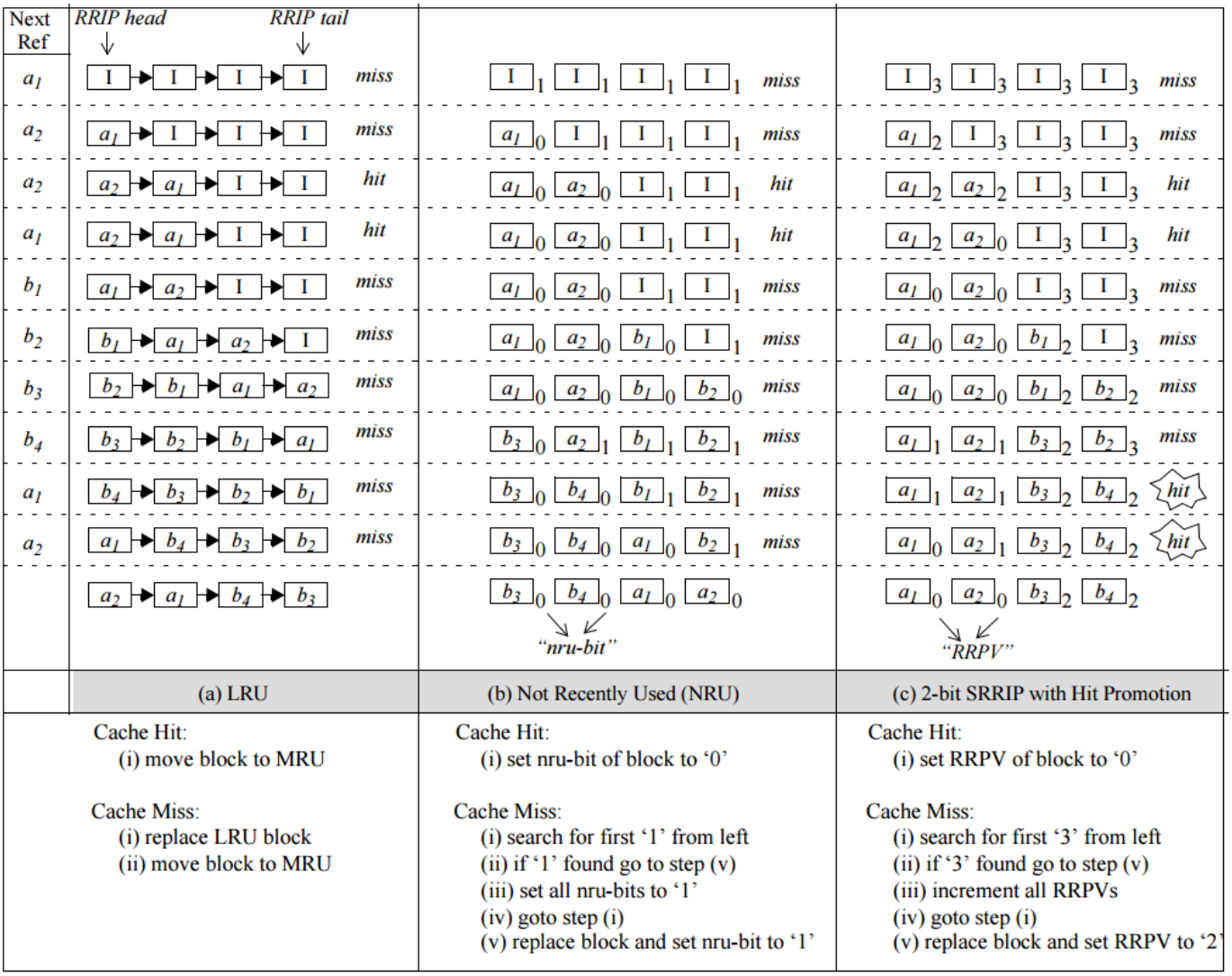


Fig. 1. Behavior comparisons between LRU, NRU and SRRIP

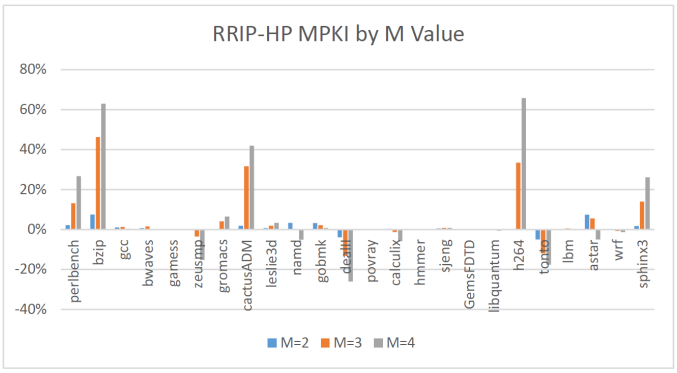
RRIP-FP in most cases, thus we do need a dynamic switching strategy like the one mentioned in the original paper to make our strategy adaptive to various workloads. Also be noticed that we didn't provide the results under $M = 1$ since in that case the RRIP strategies are basically the same with NRU and we've already normalized our results using NRU performance. Finally, we determine that the best M value that both fits well in terms of hardware budget and also provides considerably good results shall be $M = 4$ and thus we'll be using RRIP-HP with $M = 4$ in the following evaluation.

B. Comprehensive Comparison

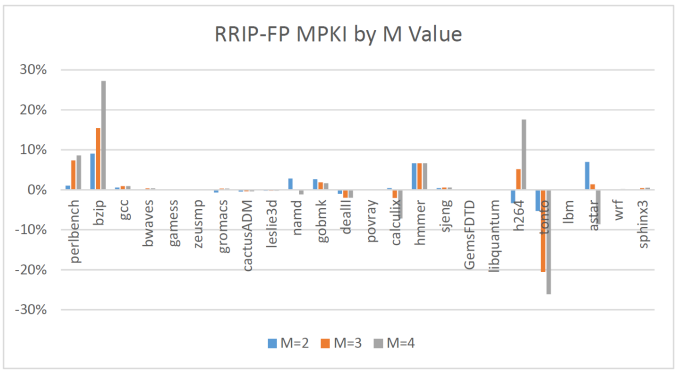
Here in this section we forget all about varying M value and different RRIP policies. We conducted experiments on the best policy and parameter settings currently known (i.e., RRIP-HP using $M = 4$), we compared the results of such a RRIP setting with the results of NRU and we choose the normalize the performance based on LRU. The MPKI and IPC figures we got are shown in Figure 3. By doing some further statistic analysis, we can further know that: comparing to LRU, the average MPKI improvements of NRU, RRIP-HP are 1% and 8% respectively, the average IPC improvement of NRU is negligent but it's approximately 2% when talking about RRIP-HP.

V. Conclusion

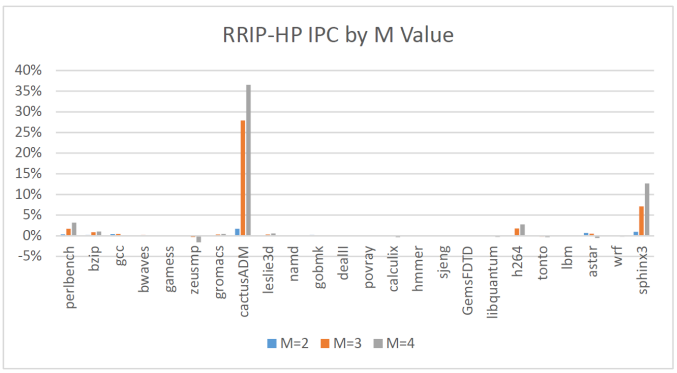
In practical cache replacement practice, the variety of applications cast a lot of different cache replacement challenges on our policies. Scan and thrashing are two typical examples where a portion of memory that will hardly be accessed again are referenced continuously. Under LRU policy, applications with scan and thrashing memory access patterns will degrade the cache efficiency much for it might take too many blocks that are useless in the long-term into cache, swapping blocks with a closer re-reference interval back to memory. To properly address this problem, Intel researchers proposed RRIP replace-



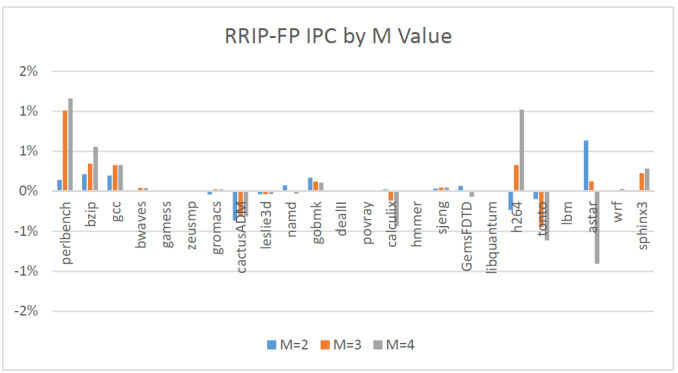
(a) RRIP-HP MPKI



(b) RRIP-FP MPKI

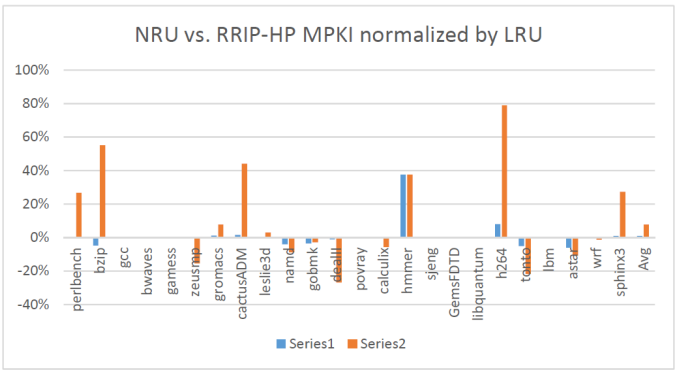


(c) RRIP-HP IPC

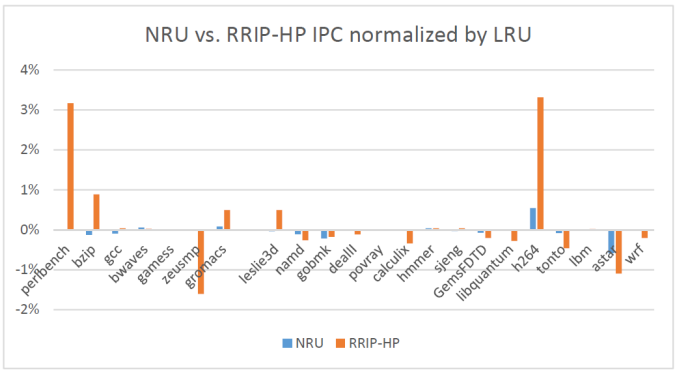


(d) RRIP-FP IPC

Fig. 2. RRIP performance normalized by NRU



(a) NRU vs. RRIP-HP MPKI



(b) NRU vs. RRIP-HP IPC

Fig. 3. NRU vs. RRIP-HP (M=4) normalized by NRU

ment policies which leverages historical block reference data to predict its future re-reference interval. In this paper, our work can be mainly summarized as follows:

- We implemented three cache replacement policies NRU, RRIP-HP and RRIP-FP based on the descriptions in [7].
- We tested the RRIP strategies using different lengths of replacement information bits and find $M = 3$ to be averagely the best bit length for both RRIP-HP and RRIP-FP.
- We carried out a series of benchmarks and compared performance of policies in this paper with LRU and noticed a maximum MPKI improvement about 80% and a maximum IPC improvement approximately 3.5%.

However, there are certain cases in which the new policies suffered a worse performance than LRU or NRU, it might due to certain workload features that make it not suitable to use our new policies. Also, we didn't do much on improving those strategies since the comprehensive experiment with over 20 workloads took too much time, so we might leave those as the future work of this paper.

REFERENCES

[1] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 208–219.

[2] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 381–391.

[3] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 63–74.

[4] Y. Xie and G. H. Loh, "Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 174–183.

[5] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 2, pp. 240–251, 2001.

[6] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. IEEE, 2001, pp. 144–154.

[7] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[8] 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*, vol. 50, no. 12, pp. 1352–1361, 2001.

[9] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 297–306, 1993.

[10] J. T. Robinson and M. V. Devarakonda, *Data cache management using frequency-based replacement*. ACM, 1990, vol. 18, no. 1.

[11] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, 2003, pp. 115–130.

[12] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement." in *FAST*, vol. 4, 2004, pp. 187–200.

[13] R. Subramanian, Y. Smaragdakis, and G. H. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 385–396.

[14] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008, pp. 222–233.

[15] 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*. ACM, 2009, pp. 401–412.

[16] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Scavenger: A new last level cache architecture with global block priority," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 421–432.

[17] T. Johnson and D. Shasha, "X3: A low overhead high performance buffer management replacement algorithm," 1994.

[18] G. H. Loh, "Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2009, pp. 201–212.

[19] K. Rajan and G. Ramaswamy, "Emulating optimal replacement with a shepherd cache," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 445–454.

[20] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 91–104.