Name: Rohan Seth(927001590) & Saloni Raina(426009634)

Project Title: Literature Survey of "An Adaptive Demotion Policy for High-Associativity Caches"

Web Link for the Paper

Date and Time of Submission: 12/10/2018, 22:00

Evaluation	MAX. Score	Your Score
Overall Organization Title, abstract (problem attempted, outline of your results, improvements) table of contents, introduction (general description of the problem, motivation, related work, and goals), description of the problem and related definitions and background, description of your work, experimental results, conclusions, appendices (your code may be included here), tables, and figures.	10	
In depth description of the problem and its significance	10	
Techniques used and definitions (should be fairly self-sufficient)	20	
Description of your work and results	40	
Technical evaluation (soundness of approach and depth) of results	60	
Conclusions, summary of work, and directions for further work	15	
Appendices (if applicable)	5	
References	10	
Overall quality of report	30	
Total	200	

Literature Survey - "An Adaptive Demotion Policy for High-Associativity Caches"

Rohan Seth - 927001590 and Saloni Raina- 426009634
CEEN
Texas A&M University

***Abstract*—The RRIP policy in general suffers a limitation in performance when it is employed in higher-level caches, and in fact, the RRIP policy causes a significant performance degradation in the execution of several applications. This is because the RRIP policy controls the priority of the block without considering the priorities of all the blocks in the set at cache misses. In several applications, it causes the priorities of the existing blocks are minimized or unchanged. This paper proposes a cache replacement policy called Adaptive Demotion Policy (ADP). The demotion policy decides the subtraction value, which is decreased from the priority values of the blocks at a cache miss. ADP generates the subtraction value based on the average of the priority values of all the blocks in the set. ADP can be implemented with small hardware overheads for high-associativity caches, and achieve the performance improvement compared with the LRU policy at all the levels of caches. We in our current literature surevey of this paper implement the various versions of ADP and do a comparative analysis reproducing the paper's results and further exhibiting the performance across Levels of Caches.**

I. Introduction

LEAST RECENTLY USED (LRU) : replacement policy, the LRU chain represents the recency of cache blocks referenced with the MRU position representing a cache block that was most recently used while the LRU position representing a cache block that was least recently used.

RE-REFERENCE INTERVAL PREDICTION (RRIP), uses M-bits per cache block to store one of 2M possible Rereference Prediction Values (RRPV). RRIP dynamically learns rereference information for each block in the cache access pattern. Like NRU, an RRPV of zero implies that a cache block is predicted to be re-referenced in the near-immediate future while RRPV of saturation (i.e., 2M–1) implies that a cache block is predicted to be re-referenced in the distant future. Since the re-reference predictions made by RRIP are statically determined on cache hits and misses, we refer to this replacement policy as STATIC RE-REFERENCE INTERVAL PREDICTION (SRRIP) :.

With only one bit of information, LFU/LRU can predict either a nearimmediate re-reference interval or a distant re-reference interval for all blocks filled into the cache. Always predicting a near-immediate re-reference interval on all cache insertions limits cache performance for mixed access patterns because scan blocks unnecessarily occupy the cache space without receiving any cache hits. On the other hand, always predicting a distant re-reference interval significantly degrades cache performance for access patterns that predominantly have a near-immediate re-reference interval. Consequently, without any external information on the re-reference interval for every missing cache block, LRU or LFU cannot identify and preserve non-scan blocks in a mixed access pattern. Whilst, scanresistance using RRIP requires that the width of the RRPV register to be appropriately sized to avoid sources of performance degradation.

Computer Science and Engineering, Texas A&M University

II. Problem Statement

RRIP LIMITATIONS :Although the RRIP policy can achieve the performance improvement compared with the LRU policy at the last-level cache, the higher-level caches cannot benefit from the RRIP [3] because the RRIP policy is not suitable for several applications. As the RRIP policy does not consider the priorities of all the blocks in the set, the priorities of the existing blocks are minimized or unchanged at several applications. Therefore, an adaptive policy that controls the priorities of the blocks is needed to improve the performance.

ADP POLICY : This paper proposes a cache replacement policy called Adaptive Demotion Policy (ADP). The demotion policy decides the subtrac- tion value, which is decreased from the priority values of the blocks at a cache miss. ADP generates the subtraction value based on the average of the priority values of all the blocks in the set. ADP can be implemented with small hardware overheads for high-associativity caches, and achieve the performance improvement compared with the LRU policy at all the levels of caches.

We in our current literature surevey of this paper implement the various versions of ADP and do a comparative analysis reproducing the paper's results and further exhibiting the performance across Levels of Caches.

III. Proposed Improvements

The Paper claims an improvement of 1.5% improvement over SRRIP during implementation in L2 cache using ADP_ I2_ FP_ LS_ HOA policy. Further paper claims improvement of around 50% on some of the benchmarks with different ADP Policies on IPC and MPKI. There is no detailed description of the previous works in this direction besides this being an incremental step over RRIP, gaining advantage in Higher Level Caches as well as in Hardware Implementation.We try to reproduce the results across benchmarks for simulation part, conclusively talking about this in detail in the Results and

the Conclusion section underneath. Please note, we in our survey are not evaluating the hardware complexity aspect, as delineated in the paper amongst different replacement policies.

IV. ADP Technique

A. Short description of ADP technique

The RRIP policy subtracts the smallest priority of all the blocks from the priorities of existing blocks when cache misses occur. This causes the priorities of the existing blocks are minimized or unchanged at several applications, and degrades the performance. To avoid this problem, ADP uses an adaptive demotion policy, which is based on the average of the priority values of all the blocks in the accessed set. ADP prepares priority values for each cache block. Regardless of the associativity, the 2-bit value is enough for storing the priority [3]. Figure 1 shows the behavior of ADP. Since caches should quickly send the data to computational resources when cache hits occur, the control logic at cache hits for updating priority value should be simple. The same as the RRIP policy, the priority value of a block is promoted by the promotion policy when cache hits occur in that block. The behavior at a cache miss is as follows. First, the block that has the smallest priority in the accessed set is selected as the victim. If there are two or more blocks which have the smallest value in the accessed set, one block is selected by the selection policy. Next, the priority values of all the blocks in the accessed set are decreased by the subtract value, which is generated by the demotion policy. Finally, a new block coming from the lower-level memory hierarchy is stored, and the priority value of the block is set to the initial priority value. The demotion policy affects the performance of ADP. The large subtraction value decreases the priority of the existing blocks in the cache, and increases the priority of the incoming block. The small subtraction value causes a reverse trend. Therefore, an adaptive demotion policy with considering the priority values of all the blocks in the set is needed to improve the cache performance.

B. Implementation details

One idea to adaptively generate the subtraction value is the averaging of the priority values of all the blocks in the set. This paper examines two demotion policies; the HOA policy and the AVE policy. The HOA policy takes the half of the average of the priority values in the accessed set, and sets it as the subtraction value. The HOA policy prevents the rapidly decreasing of the priority values. Besides, the HOA policy decreases the priority values even if there is the block that has the minimum value, and does not decrease the priority value when the total priority value is low. The AVE policy uses the average of the priority values in the accessed set. Compared with the HOA policy, the AVE policy tends to rapidly decrease the priority values at a cache miss. Although the AVE policy causes the problem that is also caused by the RRIP policy when the existing blocks have the maximum priority values, other features are the same as the HOA policy. In Figure 1, the AVE policy is adopted. For example, after the access 2, the average of the priority values is 01, so the priority values

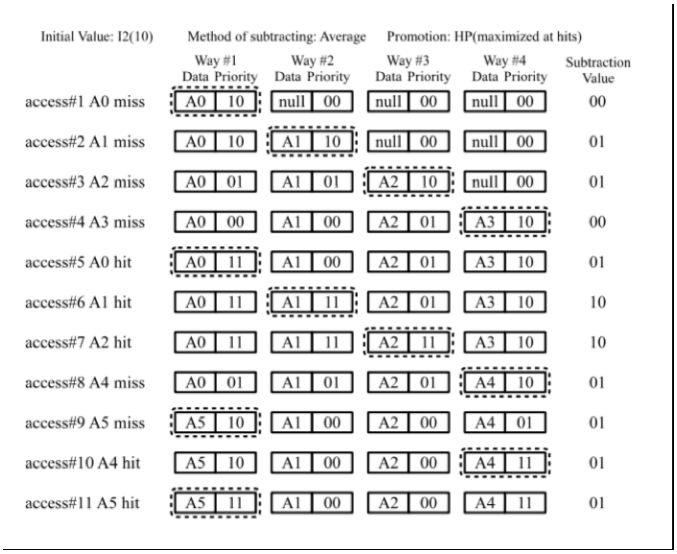


Fig. 1: An example of the algorithm of ADP (2 bit)

are subtracted by 01 at the access 3. In the other case, after the access 7, the average of the priority values is 10, so the priority values are subtracted by 10 at the access 8.

1) Insertion Policy: The insertion policy decides the priority of the incoming block. As ADP uses a 2-bit value for each block, the candidates of the initial priority value are of 00, 01, 10, and 11. In Figure 1, the initial priority value is of 10, and the priority value of the incoming block is set to that value. (E.g., at the access 1.) The small initial priority provides scan-resistant. However, if the initial priority is smaller than the smallest priority in the set, and the new block is not re-referenced before a cache miss occurs, the block will disappear. On the other hand, if the initial is too high, the new block remains in the cache for a while even though the block is never re-referenced. Based on the above, 01 and 10 are selected as the initial priority value in the experiments, and these values are illustrated as I1 and I2, respectively.

2) Promotion policy: At a cache hit, the priority of the hit block is promoted. There are two promotion policies; Hit Priority (HP) and Frequency Priority (FP). The HP policy gives the highest priority to the hit block, and the FP policy increases the priority by one. Figure 1 shows the behavior when the HP policy is adopted, and the priority of the hit block is maximized. (E.g., at the access 5.) Because the HP policy gives the highest priority to the hit block, the HP policy enlarges the importance of the hit compared with the FP policy. As the FP policy increases the priority by one, that provides scan-resistant when the initial priority is small. Although the HP policy achieves higher performance than the FP policy in the RRIP policy [3], the FP policy is suitable for several benchmarks. In the experiments, the effects of both promotion policies on ADP are examined.

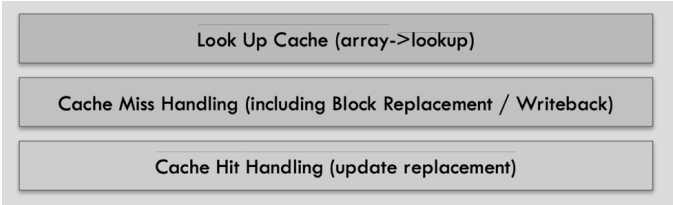


Fig. 2: Zsim hierarchy flow for ADP Implementation (2 bit)

V. Methodology

We use Zsim, a full featured memory based system simulator for Caches, to conduct our performance studies. Our baseline processor is single core westmere system(or a 8 core processor for multi threaded instructions)with 64 bit wordlength and three level cache. The L1 instruction cache is a 4way associative 32K, L1 data cache is 8 way Set associative 32K. L2 cache is 256K, 8 way associative and L3 is 2MB 16 way associative. Only demand references to the cache update the LRU state while non-demand references (e.g., write back references) leave the LRU state unchanged. The load-to-use latencies for the L1, L2, and L3 caches are 1, 10, and 24 cycles respectively. Cache access herein happens as described in Fig. 2. [h!]

```
// Cache Access implementation
Look Up Cache (array->lookup) {
If find the line {
Update repl_policy and return line ID;
} else {
return -1;
}
}
```

And this is how a miss is handled inside the zsim implementation.

```
Cache Miss Handling (including Block
    Replacement / Writeback)
{
array->preinsert(); // consult repl_policy to
    find a victim
cc->processEviction(); // write back if needed
array->postinsert(); // finish and update the
    replacement
}
```

Thus we implement the update(), replaced() and rank() functions to our SRRIP implementation.

```
      void update(uint32_t id, const MemReq*
          req) {
         if(!miss)
     {
     array[id] += 1; // Frequency Promotion,
         will vary this part for Promotion
         Policies.
       }
         miss=0;
  }
```

Fig. 3: Internal breakup of array access functions

Underneath is the code snippet of the replaced() function.

```
     void replaced(uint32_t id) {
      // printf("\nThis is miss case, updating
          value of %d to 1", id);
          array[id] = 1; // Set the replaced
              value to 1.
          miss = 1; // Setting a variable for
              miss, such that update()
              function sees this.
       }

void replaced(uint32_t id) {
          array[id] = rpvMax - 1; //Reduce
              the score by 1 upon replacement
          miss = 1;           // Set miss
              variable for update().
       }
```

Underneath is the code snippet of the Calculate_ ave() function, that is used to devise the Average and the half of Average.

```
  template <typename C> inline uint32_t
      calculate_ave(C cands){
            uint32_t sum=0;
            uint32_t count=0;
            uint32_t ave=0;
            for (auto ci = cands.begin(); ci
                != cands.end(); ci.inc()) {
                  sum += array[*ci];
                      //Cumulative sum to
                      compute average
                  count +=1;
            }
     count = count*2; //Halfin the count to
         get HOA, in case of AVE this count
         kept same.
            ave = (sum/count);
     if(ave <= 0) ave = 1; //Check to ensure
         that if the average of the priorities
         is zero then, we need to give it a
         non zero positive value, else the
         State Machine will hang.
     count =0;
          return ave;
       }
```

```
template <typename C> inline uint32_t
    rank(const MemReq* req, C cands) {
      uint64_t bestCand = -1;
  uint32_t avg = 0;
  avg = calculate_ave(cands); // This
      variable gets the average value for
      subtraction.
  bestCand = -1;
      while(bestCand == -1){
         bestCand =
             search_candidate(cands); //
             Call to search the
             candidates, independent
             function.
         if(bestCand == -1) //No value of
             3/0 found ,(depends on the FP
             or HP promotion policy)
           {
    auto ci = cands.begin();
                for (ci = cands.begin(); ci
                    != cands.end();
                    ci.inc()){
          if(avg >array[*ci]) array[*ci]
              =0;
               else array[*ci] =
                   array[*ci] - avg ;
                   //Difference from
                   SRRIP, this subtracts
                   the avg value than the
                   simple 1 in case of
                   SRRIP. This avg has
                   been computed above.
     }
           }
       }
    return bestCand;
}
```

Our setup has been made modular with defines that enable all the runs to be fired from a change of define in the run command.

```
#ifndef ADP_I1_FP_LS_HOA_REPL_H_
#define ADP_I1_FP_LS_HOA__REPL_H_
```

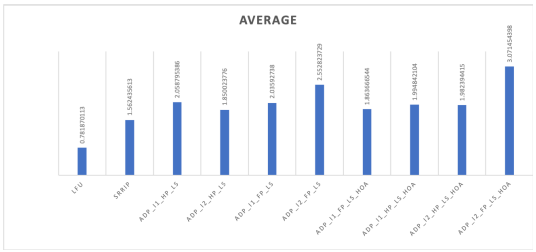
VI. Evaluation

We conduct evaluation based on the following criterion amongst SRRIP and various ADP Policy versions. Here all the percentage improvements are considered relative to LRU.

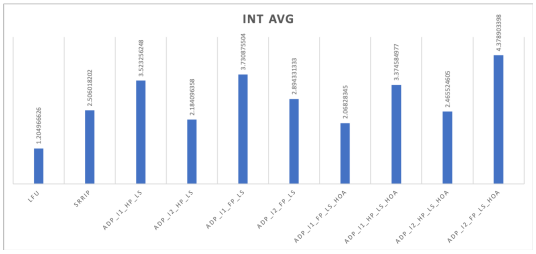
- MPKI

$$total\_misses = mGETS + mGETXIM + mGETXSM \quad (1)$$

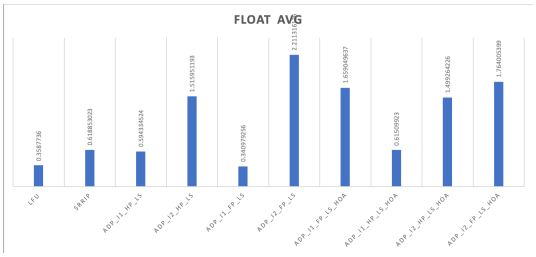
$$MPKI = (total\_misses / total\_instruction) * 1000 \quad (2)$$



(a) Cumulative average on MPKI



(b) Integral Benchmarks average on MPKI

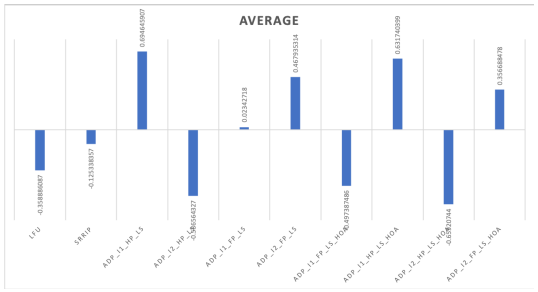


(c) Float Benchmarks average on MPKI

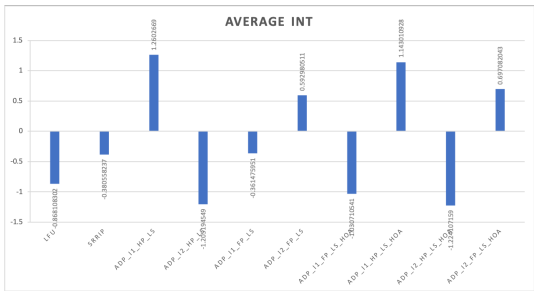
- Here we observe that on cumulative benchmarks of SPEC, a few of the ADP policies namely, ADP_ I2 _ FP_ LS_ HOA (ADP with Instertion Policy =2, Left Shift, Frequency Promotion, Half of average) policy shows the percentage improvement to 3.06% vs. 1.56% of SRRIP demonstrated in Fig. a above. For the further points we consider this (ADP_ I2 _ FP_ LS_ HOA) policy as our representation for this parameter.
- Overall, on integer Benchmarks this policy shows a performance improvement of 4.37% vs. 2.5/% of SRRIP.
- Overall, on Floating Benchmarks this policy shows a performance improvement of 4.37% vs. 2.5/% of SRRIP.
- In general, majority of ADP Policies flavors perform equivalent to SRRIP, with a few exceptions like our zeroed policy of ADP_ I2 _ FP_ LS_ HOA.
- ADP_ I1_ FP_ LS performs quite erratically with drastic improvement in couple of benchmarks but quite inferior performance compared to SRRIP.
- IIndividually, we observe that performance improvement with ADP Policy is most dominant with integral benchmarks than Floating Benchmarks. With ADP, it either performs just at par with SRRIP.
- For Integral benchmarks: bzip2, gcc and hmmer

are the ones exhibiting significant improvement over others, shooting the average performance to 4.37% vs. 2.59% achieved with SRRIP.
  - For Floating benchmarks: soplex is the only one exhibiting significant improvement over others, shooting the average performance to 1.7% increment vs 0.61% achieved with SRRIP.
- IPC or Speedup/Performance

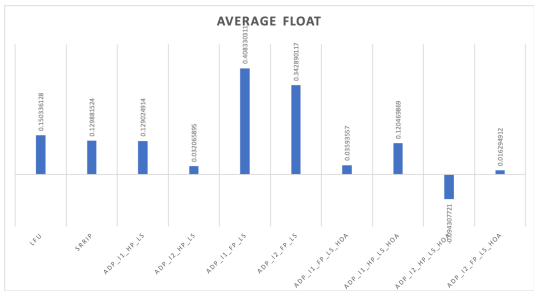
$$IPC = total_instruction/total_cycles \quad (3)$$



(a) Cumulative average on IPC

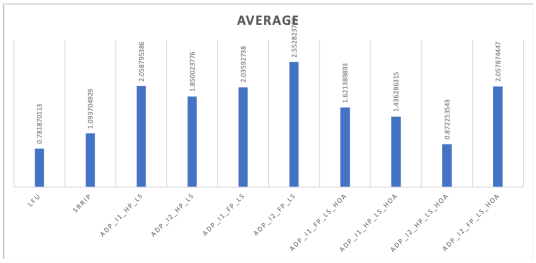


(b) Integral Benchmarks average on IPC

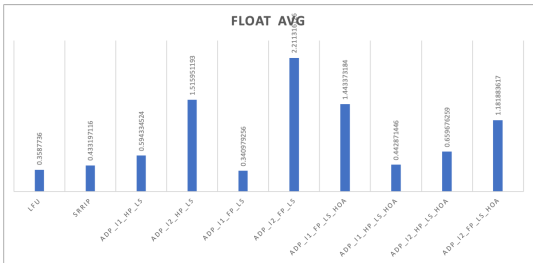


(c) Floating Benchmarks average on IPC

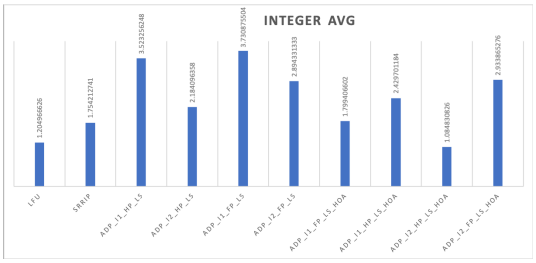
  - Here we observe that on cumulative benchmarks of SPEC, the ADP policy (ADP_ I1 _ HP_ LS_)(ADP with Instertion Policy =1, Left Shift, Hit Promotion, Full average) policy shows the best performance with improvement of 0.69% vs. a decrement of 0.12% of SRRIP demonstrated in Fig. a above. Our chosen policy of ADP_I2_FP_LS_HOA that exhibited the leading performance in MPKI is not the leading improvment provider here, but nevertheless still demonstrates a significant improvment of 0.35% vs. -0.12%.
  - Overall, on integer Benchmarks our chosen policy(ADP_I2 _FP_LS_HOA) shows a performance improvement of 0.69% vs. a decrement of 0.38/% with SRRIP.
  - Overall, on Floating Benchmarks this policy shows a relatively performance degradation of 0.012% vs. 0.12/% of SRRIP.
  - In general, half of ADP Policies flavors do not demonstrate a performace improvement with respect to SRRIP.
  - There is no clear discernible criteria, but it seems that HP_ LS and FP_ LS_ HOA are not very good combination to extract improvement of IPC.
  - IIndividually, we observe that performance improvement with ADP Policy is most dominant with integral benchmarks than Floating Benchmarks as was observed in MPKI enhancement.
  - For Integral benchmarks: hmmer is the one bringing significant degradation of performance over others, this individual benchmark deteriotes the performance to -10%.
- Replacement Policies on L2 Cache.



(a) Cumulative average on L2 Cache



(b) Integer Benchmarks average on L2 Cache



(c) Float Benchmarks average on L2 Cache

  - Here, we observe that on a higher level cache(L2) ADP unanimously shows better performance over SRRIP. It exhibits a percentage improvement of

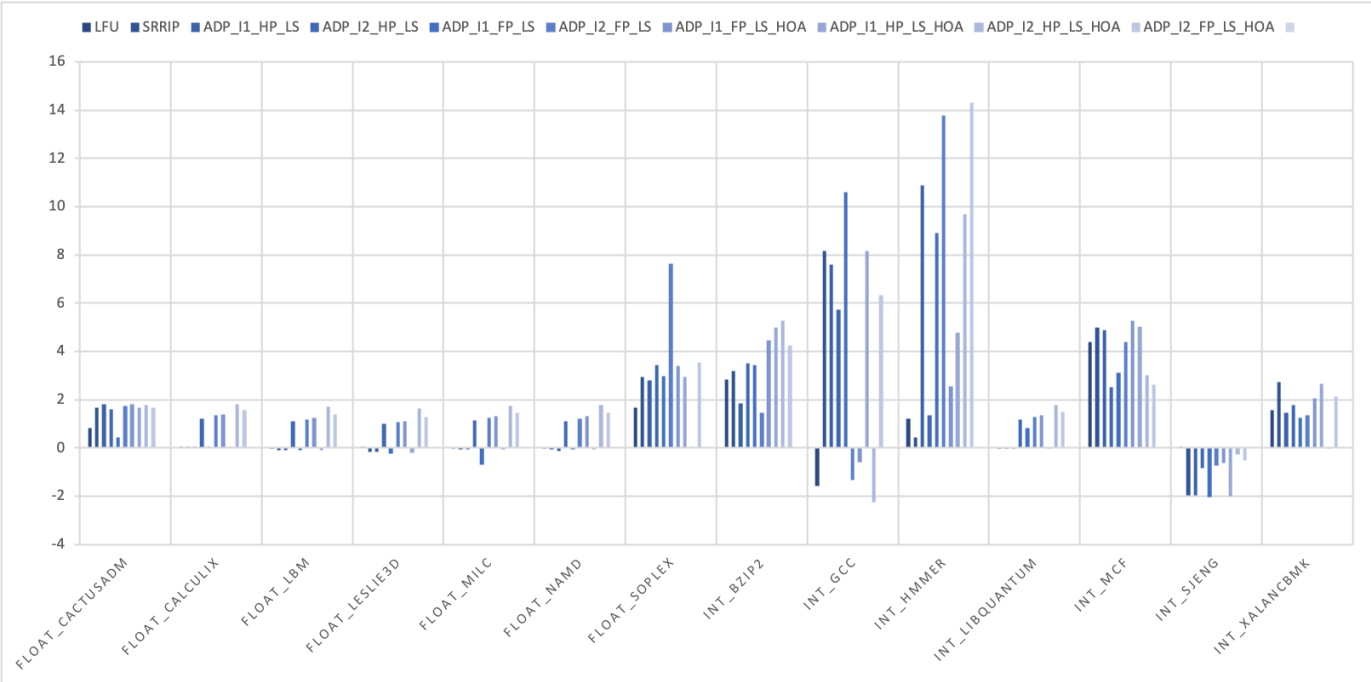


Fig. 4: MPKI Comparisons across Benchmarks for various ADP Policies vs. SRRIP

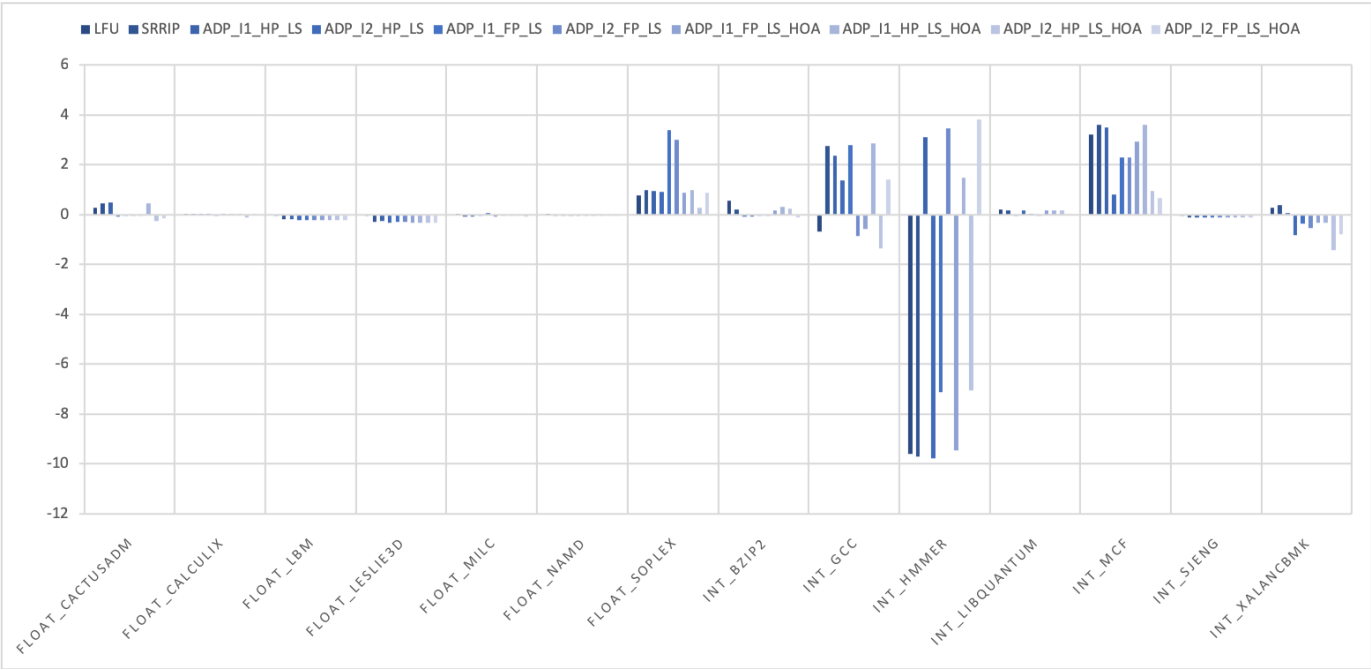


Fig. 5: IPC Comparisons across Benchmarks for various ADP Policies vs. SRRIP

2.05% vs. 1.09% for SRRIP. Shown in Fig. (a) on the next page. This pretty much matches as per the paper's proposed improvements.

- Overall, ADP_I2_FP_LS and ADP_I2_FP_LS_HOA are the two best policies amongst ADP, exhibiting percentage improvements of 2.55% and 2.05% over LRU.
- Individually over Integral Benchmarks, we get improvement of 2.93% vs. 1.75% with SRRIP.
- Individually over Floating Benchmarks, we get improvement of 1.18% vs. 0.43% with SRRIP.
- Percentage improvement gain we get in Integer Benchmarks is much comparative to Floting Benchmarks.

- With the exception of Sjeng in Integer Benchmarks, all the benchmarks exhbit a performance improvement or are at par with SRRIP.
- Bzip2, gcc and hmmer in Integer Benchmarks and soplex in Floating Benchmark brings in the maximum performance improvement across all benchmarks.

Analysis We observe the claim of the paper under survey that exhibits the performance improvement over SRRIP in Higher Level Caches is reproducible to almost similar extent. In the above sections, as demonstrated we observe that amongst the different combinations of ADP Policies that were proposed, all exhibit wide varying results.

Through our results we observe that ADP_I2_FP_LS_HOA performs reasonably well cumulatively across all paramters on all our run benchmarks. The paper claims the same, alongside somewhat equivalent improvement with ADP_I2_HP_LS, but we on our benchmarks observe this policy slightly inferior than the earlier. Because of the temporal locality being too high at the L1/2 instruction cache, the HOA policy cannot reduce enough the priority values of the existing blocks to prevent the new block from being evicted. The AVE policy can reduce enough the priority values of the existing blocks, and help evicting unnecessary data. At the other-level caches, the AVE policy reduces too much priority values, so it increases the probability of evicting useful data. ADP policy in general is a broad hybrid mixture of LRU and MRU, just like the SRRIP and proves to be of significance when the data structure on operation's replacement size is greater than cache size. In assumption, if there is a data structure of k entries and is then updated a different data structure of m entries. For access patterns, when m + k is less than the available cache, the total working set fits into the cache and LRU works well. However, when m + k is greater than the available cache, LRU discards the frequently referenced working set from the cache. Consequently, accesses to the frequently referenced working set always misses after the scan. In the absence of scans, mixed access patterns prefer LRU. However, in the presence of scans, the optimal policy preserves the active working set in the cache after the scan completes. This is where SRRIP and ADP exhibits its dominance over LRU/LFU. Since we are getting more enhanced amplification in Integral benchmarks, broadly we could say that Integer Benchmarks tend to explot the access patterns much more than Floating Benchmarks.

When the hit rate of the cache is high, a cache miss minimizes the priorities of all the existing blocks in the set. This causes a harmful effect to the performance by evicting useful data. Besides, if there are blocks that have the minimum priority, the priorities of the blocks in the set are not decreased even though a cache miss occurs. This causes remaining unnecessary data in the cache and decreases the performance. However, the performance of ADP is lower than that of SRRIP at several benchmarks even though some of the ADP policies exhibit an improvement over the same. This is because one configuration is not suitable for the benchmark, and it increases the overheads of the set-dueling. To improve the performance of ADP, the configuration should be changed by the behavior of the application. If ADP could dynamically change the demotion, insertion, promotion and selection policies. In future work, a dynamic method for optimization of these policies could prove very promising results, even extending from where this work was left.

Highlighting the outliers, hmmer, gcc and bzip2 are the benchmarks exhibiting the maximum gain for the ADP Policies. From the extrinsic observations it seems that these patterns are relient highly on temporal locality with high number of successive hits thus demonstrating a signficant gain for the ADP Policies. Also, on similar lines we could infer the soplex benchmark from SPEC suite of Floating Benchmarks, shows degradation with the proposed approach hinting to the fact of highly hit, miss pattern repetition in presence for this benchmark.

Acknowledgments and Contribtions

The above effort was an effort of two of us in a group. Rohan and Saloni were equally responsible for running the simulations after setting up the setup that was done at a common place. The effort required a humongous 320 simulations to be run to extract the information across benchmarks. We even had to resort to not running our simulations on L1 cache with RS policy as done in paper due to lack of time and compute respurces. We are thankful to Akhilesh and the complete Piazza discussion chain which helped us run the simulations on a local Ubuntu environment on my system, which has significantly helped in reducing the running time by helping in conducting jobs in parallel. Post the results reproduction, the report was handled by Rohan, while the Graphical Plotting was catered by Saloni. The Conclusion, Analysis and brainstorming behind it was an equal effort of the group.

Conclusions

This paper claimed a high-performance cache replacement policy called ADP, which is suitable for implementing high-associativity caches. The proposed policy claimed to surpass the SRRIP, and does not affect the processor cycle time even though the associativity is high. Our results are on the similar page, with a few exceptions as highlighted in the above sections. Alongside, we propose a suggestion over the implemented paper that we strongly believe could result in even higher performance achievement.

Proposed Future Extensions

The performance of ADP is lower than that of SRRIP at several benchmarks even though some of the ADP policies exhibit an improvement over the same. This is because one configuration is not suitable for the benchmark, and it increases the overheads of the set-dueling. To improve the performance of ADP, the configuration should be changed by the behavior of the application. If ADP could dynamically change the demotion, insertion, promotion and selection policies. In

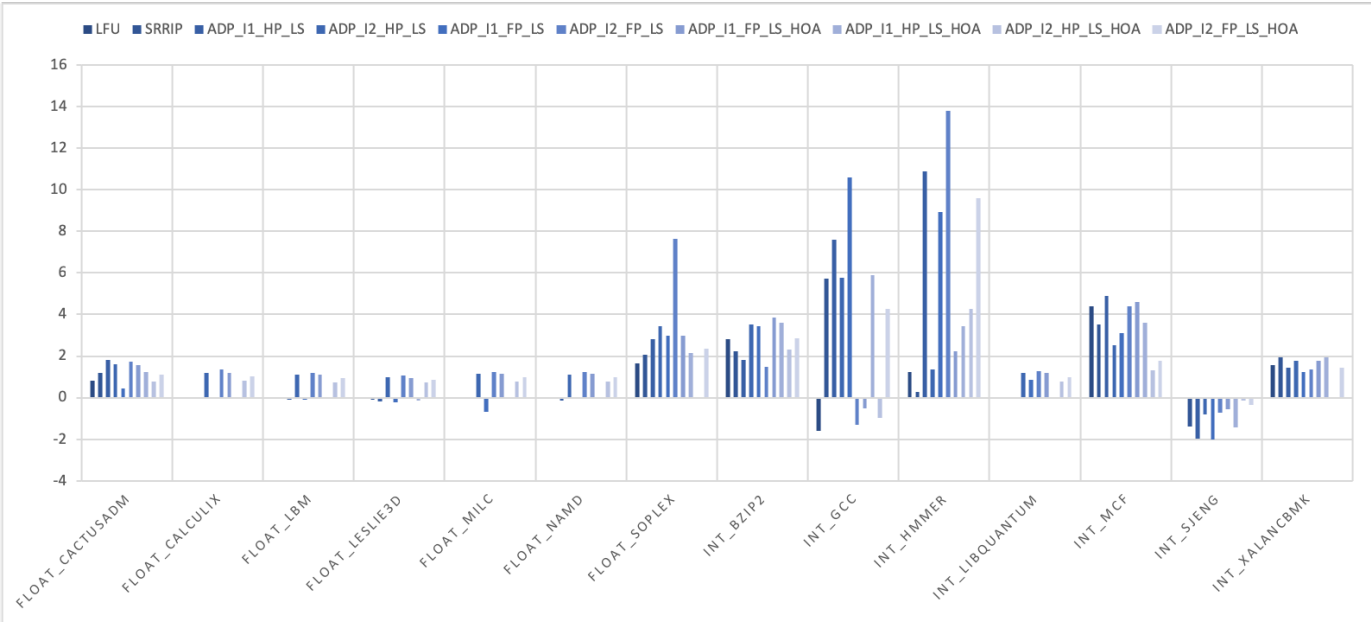


Fig. 6: Performance across benchmarks on L2 Cache with various ADP Policies vs. SRRIP

future work, a dynamic method for optimization of these policies could prove very promising results, even extending from where this work was left.

We plan to extend this paper's analysis in the future if we decide to work on this. We plan to amalgamate the usage of Prediction Models to make the ADP, a dynamic proposition in future. As per our current Neural Networks understanding, this seems a pretty straightforward proposition without a need of Deep Neural Networks and decent size of Hidden Layer. But, till we implement this, we can always be surprised by the results.

APPENDIX

The key code snippets have been pasted above in the relevant sections. Also, while turning-in the zip has been included to include all the sources, run and analysis files. We have also published our analysis on git for further usage and references.

REFERENCES

[1] Jubee Tada, Masayuki Sato, and Ryusuke Egawa. 2017. An Adaptive De- motion Policy for High-Associativity Caches. In Proceedings of HEART2017, Bochum, Germany, June 07-09, 2017, 6 pages.

[2] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," ISCA, pp. 529–551, April 2013.

[3] A. Jaleel, K. Theobald, S. Steely and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP), ISCA, Jun 2010, pp.68–73.

[4] T.S.B.Sudarshanetal.,"HighlyefficientLRUimplementationsforhighassociativity cache memory," in Proceedings of 12th IEEE International Conference on Advanced Computing and Communications, 2004.

[5] M.K.Qureshietal.,"AdaptiveInsertionPoliciesforHighPerformanceCaching," in Proceedings of International Symposium on Computer Architecture (ISCA 2007), 2007

[6] A. Jaleel et al., "High Performance Cache Replacement Using Re-Reference Inter- val Prediction (RRIP)," in Proceedings of International Symposium on Computer Architecture (ISCA 2010), pp. 60–71, 2010

[7] H.Ghasemzadeh,etal.,"ModifiedpseudoLRUreplacementalgorithm,"inProceedings of 13th Annual IEEE International Symposium and Workshop on Engi- neering of Computer Based Systems, 2006.