Computer Architecture

Project report:

Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer

Diane Uwacu and Fatmaelzahraa Elsheimy

Abstract

Many of the new-era processors support fetching instructions with instruction cache and branch target buffer. I-Cache and branch target buffer have limited capacities and therefore, different types of replacement policies are being explored to reduce the misses in I-cache and BTB. In this project, we implement a new policy Global History Reuse Prediction (GHRP), a replacement policy that uses the history of previous instructions and behaviour predict and evict dead blocks. GHRP is implemented in the main paper using Championship Branch Prediction simulator, but we try to implement it using Zsim to test the easiness of implementing the new policy in different simulators. GHRP's performance is compared against the famous policy LRU (Least recently used) and SRRIP(static re-reference interval prediction). Using Championship simulator, GHRP reduces the I-cache misses (MPKI) by 18over LRU policy on a 662 industrial workloads. In Zsim, LRU data is collected over PARSEC benchmark and the results are speculated from the Championship results.

1 Introduction

Current processors need efficient instruction fetch for high performance. There are two main structures that the processor rely on for efficient fetching: I-cache and branch target buffer. The instruction cache (I-cache) keeps record of recently used instructions, and the branch target buffer (BTB) caches targets of previously taken branches. Therefore, the I-cache helps in improving instruction throughput and latency. Branch target buffer improves latency because of branch target-recomputation. This paper [2] explores efficient replacement policies at the I-cache and BTB levels. Samira et-al introduces a new idea for policy replacement called Graph History Reuse Prediction. This algorithm is based on the history of past instruction addresses and behaviour in order to predict dead blocks in an efficient way. Predicting dead blocks is very important as they free up space for other blocks that might be used in the future. A block is said to be dead if it is not going

to be used before eviction. A block is said to be live if it is going to be used before eviction. Samira et-al have done a survey on the current implemented policies such as LRU, SSRIP..etc and tried to find a better policy. Most of the work on cache optimization focused on instruction prefetching and block reordering. Many of these solutions didn't lead to good outcome as it resulted in large overhead or high MPKI. Similarly, most of the work on BTB optimization focused on the design structure or using different replacement policy. Again, the results weren't that satisfactory. Samira et-al first, tried to apply PC-based predictor that uses set-sampling for Cache behavior generalization, however, when being applied on I-cache and BTB, they haven't seen prominent results and hence, this idea was discarded. On the other side, based on the survey done over all current replacement policies, they noted that that sequences of recently accessed instructions correlate with the likelihood of block reuse. Their proposes policy results in a prominent improvement in cache efficiency and overall performance. They evaluated seven recent replacement policies and compared them based on average MPKI values. While, Samira et al applied their policy on Championship simulator. We have tried applying it on Zsim to test the ease of implementation of the policies in different simulators and compare the results using different benchmark; PARSEC.

2 Background

When looking at previous publications, they have only focused on enhancing prefetching methods or software-based approaches. In this project, We are focusing our effort on improving replacement policies in I-cache and BTB based on dead-block prediction. Recently, many studies are based on the effect of replacement policy, not many were found or mostly gave poor results [10, 7] . They implemented different approaches such as recent static prediction [4] and using a hashed value for indexing branch predictors table [9]. Other works that studied dead-block prediction mainly focused on using it in power reduction [1], cache optimization [3] or bypassing [5]. Therefore, our efforts will be focused on studying the use of dead block prediction in the I-cache and BTB. Different algorithms will be tested to be able to adapt the dead block information to BTB and I-cache replacement using Zsim as we are most familiar with.

In this paper [2], Samira et al have surveyed several replacemenet policies as not much work has been done to evaluate thems on the I-cache and BTB level. Except for the work in [8] that was done in the 80s with simpler policies like FIFO, and the work by Perleberg et al [6] from the 90s, this is the first recent work to look at this problem from this angle.

Authors discuss how different prediction replacement policies affect and are affected by the I-cache and BTB. In particular, they showed how a sampling-based prediction policy that works well for PC-based dead block prediction, does not work as well in this area, due to the fact that a given PC only accesses one set at a time. In addition to the sampling-based method, authors compare their work againt the least recently-used (LRU) policy, the random policy, and the static re-reference interval policy (SRRIP).

3 Work Description

The GHRP method uses past history to predict dead-blocks in the cache. It indexes a table of counters with a signature generated from features correlated with reuse behavior using the global path history of instruction addresses. The global history is updated by shifting three lowest-order bits of the PC, and the signature is found by exclusive-ORing the history with the PC. GHRP uses three hash tables that provide a prediction via majority vote. With a 64kB I-cache with a 64B block size, we created three prediction tables with 4096 entires and a two-bit counter per entry.

Our plan was to implement the Global History Replacement Policy (GHRP) method as described in the paper, and assess its performance by comparing it to the LRU method. The method was implemented using a simulator that we were not familiar with, so we decided to implement the logic in the zsim simulator instead. Algorithm 1 describes our implementation. We split Algorithm 1 of [2] into three procedures that are required to use the Cache classes in zsim.

The update() function is used in zsim to update the prediction tables after a hit. In our implementation, we called updatePredTables() without flagging that the block is dead, as detailed in Algorithm 1 of [2]. The replaced() function is used in zsim to replace In our implementation, we called updatePredTables(), and this time we that the block is dead, as detailed in Algorithm 1 of [2]. The rank() function is used in zsim to rank replacement cadidates. We didn't need it in our implementation, since we called the MajorityVote() function detailed in Algorithm 1 of [2] that is used on the hashed indices from the three prediction tables to find a block to update or replace depending on how the flag is set.

To fall back to a LRU replacement when there is no block picked by the GHRP method, we added an implementation of LRU replacement in our implementation. We did not get a chance to adapt the method for BTB replacement due to time restrictions.

Algorithm 1 GHRP

```
1: procedure UPDATE(uint32_t id, MemReq* req)
2:     sign ← Signature(id, req)
3:     indices ← ComputeIndices(sign)
4:     boolisDead = false
5:     UpdatePredTable(indices, isDead)
6: end procedure
7: procedure REPLACED(uint32_t id)
8:     sign ← Signature(id, req)
9:     indices ← ComputeIndices(sign)
10:    boolisDead = false
11:    UpdatePredTable(indices, isDead)
12:    pred ← MajorityVote(indices)
13:    array[id] = pred
14: end procedure
```

4 Evaluation

4.1 Experimental Setup

We opted to implement the method in zsim since we were more familiar with the simulator, and we thought that the paper provided enough information to implement the method. Apart from the difference in data structures, certain components that are explicitely updated in the method were assumed to be computed by the Cache class in zsim, and thus outside of the scope of the replacement policy code.

We were able to implement the code, but we were unable to link it to the Cache class properly to be able to run experiments as it was difficult to adapt it to the current zsim architecture. In this section, we will discuss the performance of GHRP compared to SRRIP and LRU polices over PARSEC benchmark. These are speculations done by comparing our LRU results to the paper's results.

4.2 Experimental Results and Discussion

In order to compare the efficiency of the GHRP policy, it was implemented on two different simulators; Champ and Zsim. In the paper, Samira et al implement the policy using Champ and comparing it with other policies such as LRU, and SRRIP on industrial benchmarks. In our project, we tried implementing on Zsim and comparing the results against SRRIP and LRU on PARSEC benchmark. Unfortunately, due to Zsim architecture and the way the policies are implemented. It was difficult to implement the GHRP policy in the current Zsim architecture. We collected data from Samira et

al as well as collected data from the LRU AND SRRIP on PARSEC and we speculated the results by GHRP. In figure 1, It shows the results of the implementation of different policies on an 8-way 64KB I-cache with 64B blocks using Champ simulator. Figure 1 shows the results of these policies in relative to the MPKI of the LRU. On average of the 662 workloads, GHRP has a lower MPKI average of 0.86 compared with 1.02 for SRRIP and 1.10 for SDBP. By percentage, GHRP improves the average MPKI by 18combinations of 8KB, 16KB, 32KB and 64KB caches with 64B blocks with 4-way or 8-way associativity. It can be clearly seen how GHRP outperformed all other policies. LRU has performed the worst results by average of 10Using Zsim, GHRP is compared against LRU and SRRIP on PARSEC benchmark. The MPKI value is calculated using the following equations: total misses = mGETS + mGETXIM + mGETXSM

MPKI = (total misses/total instruction)*1000 Where mGETS are misses of the GETS , mGETXIM are the misses of GETXI =¿ M, and mGETXSM are the misses of the GETXS =¿ M .

Figures 3 and 4 display the results of the policies over the PARSEC benchmarks. The value of the MPKI for each policy was calculated using the above mentioned equations and recorded in figure 3. It can be clearly shown the LRU has done the worst compared to all of them. SRRIP has done better than LRU by average of 10results, the GHRP is expected to outperform both LRU and SRRIP by average of 8The analysis behind such results is due to the algorithm and the core idea of GHRP. GHRP targets the removal of dead blocks/BTB entries by using the history of past instructions addresses and behavior. LRU just uses information based on the last recently used block, which isn't really efficient to detect dead blocks especially when it does not think that the most recent entry will be re-referenced in the near-immediate future. In results, LRU is inefficient especially with applications that its referenced mainly in the distant future. LRU replaces the blocks very frequently. SSRIP uses lower hardware overhead and deals better with applications with larger working sets and applications with references in the distant future than LRU. SSRIP has two kinds of priorities: Hit priority and Frequency priority. For this project, the Hit priority one was implemented. Therefore, it can be deduced that many of the PARSEC benchmarks have blocks that are referenced in the distant future and that's why SRRIP outperformed LRU. For GHRP, it can predict dead blocks in application with larger worker sets and references in the distant future better than both SRRIP and LRU. SRRIP uses 2 bits per cache block to store $2\hat{2}$ prediction values (RRVP)for scan resistant, which means a block has 4 chances before being evicted. GHRP doesn't face this problem as it bases its prediction on indexing a table of counters with a signature generated

by correlated it to data from previous history. The final prediction is passed by majority vote which further improves the assurance of the prediction. Hence, a block has a better chance to stay in cache before being evicted. Overall, GHRP better predict dead blocks than SRRIP and LRU.

5 Conclusion and Future Work

We implemented the global history replacement policy in zsim and made enough progress to show that it is realizable in a different simulator than originally presented. However, due to zsim setup and the time constraint, it wasn't very easy to implement GHRP in a way that's adaptable to the current architecture. However, we still implemenetd the algorithms fully in zsim as shown in the pseudocode. Based on the results in the paper and the extrapolation done, it is clearly shown that GHRP outperforms all the other policies. In this project, we compared it against LRU and SRRIP, and GHRP shown a better performance by reduing the MPKI by average of 10For future work, it would be great to try implementting the policy on other tools other than zsim and Championship and tests its performance on other benchmarks as well.

References

[1] J. Abella, A. González, X. Vera, and M. F. P. O'Boyle. Iatac: A smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, Mar. 2005.

[2] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez. Exploring predictive replacement policies for instruction cache and branch target buffer. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 519–532. IEEE Press, 2018.

[3] An-Chow Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 139–148, 2000.

[4] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, 38(3):60–71, June 2010.

[5] T. L. Johnson, D. A. Connors, M. C. Merten, and W. . W. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, 1999.

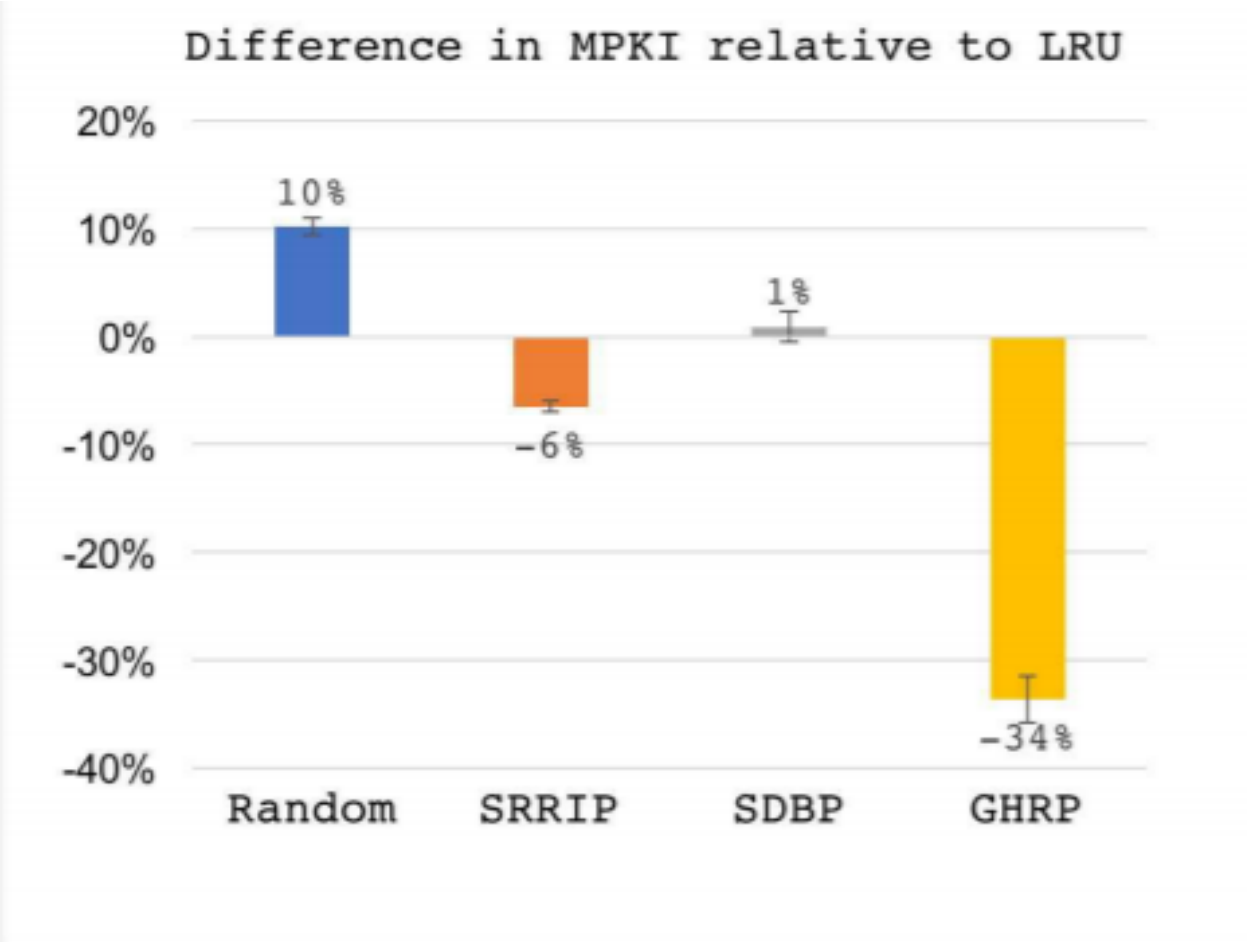
[6] C. H. Perleberg and A. J. Smith. Branch target buffer design and optimization. *IEEE Trans. Comput.*, 42(4):396–412, Apr. 1993.

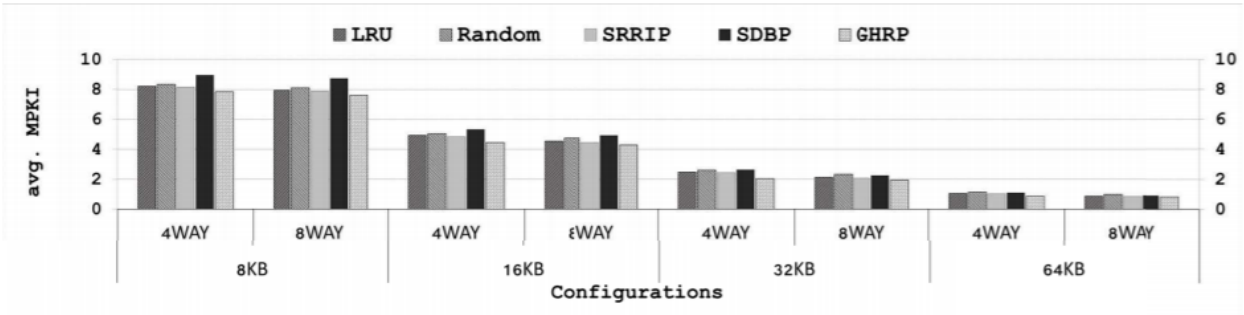
[7] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 167–178, 2006.

[8] J. Smith and J. Goodman. Instruction cache replacement policies and organizations. *IEEE Transactions on Computers*, 34(03):234–241, mar 1985.

[9] D. Tarjan and K. Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM Trans. Archit. Code Optim.*, 2(3):280–300, Sept. 2005.

[10] C. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer. Ship: Signature-based hit predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.


Difference in MPKI relative to LRU
20%
10%
0%
-10%
-20%
-30%
-40%
10%
-6%
1%
-34%
Random
SRRIP
SDBP
GHRP


LRU
Random
SRRIP
SDBP
GHRP
avg. MPKI
10
8
6
4
2
0
4WAY
8WAY
8KB
16KB
32KB
64KB
Configurations

	LRU	SRRIP	GHRP Speculated
Blackscholes	4.20	3.49	3.32
Bodytrack	9.10	7.55	7.55
Canneal	14.00	13.50	11.06
Dedup	10.20	8.47	8.06
Fluidanimate	3.00	2.49	2.37
Streamcluster	5.70	4.73	4.73
Swaptions	3.00	2.49	2.37
Freqmine	1.20	1.00	0.95
x264	6.80	4.52	4.50