Perceptron Learning for Reuse Prediction

Swathi Changalarayappa
Department of Electrical and Computer Engineering, Texas A&M University
swathi_c@tamu.edu

***Abstract*—A miss on the last level cache causes a huge penalty on the performance. This is because of large size of Last Level Cache (LLC) and the extra burden of missing on the previous caches. This gives rise to the need for a better replacement policy in last level caches. This paper is an implementation of [1], which is one of the best replacement policy till date. It incorporates the idea of learning from a few blocks and generalizing it to the entire cache, thereby, reducing area and power. This paper makes a few improvements on [1] to achieve a speedup of *6.4%* as compared to *6.1%*, the speedup obtained using [1]. Reuse predictor identifies dead blocks in a cache, that would be replaced instead of an LRU block using the PC and memory address of the instruction accessing LLC. The predictor is updated using perceptron learning to yield a normalized geometric mean speedup of A against *3.8%* for SHiP and *3.5%* for SDBP over LRU for single threaded workloads.**

Keywords—cache, replacement, perceptron, dead block, reuse

I. INTRODUCTION

The large disparity between memory latencies and last level cache motivates the search for a good replacement policy. Cache, in general, exhibits good temporal and spatial locality. Hence, it is not very hard to learn and predict from the program behavior. However, this is not true for LLC, because all the locality is filtered out from the first 2 levels of caches. This results in a very poor footprint of the program on LLC, making it hard to predict which block to replace. Adding to this complexity is a heavy area and power requirement of an LRU policy. The area and power grow at a faster rate with increasing size and associativity of cache. The hit time is also affected as getting a victim block falls on the critical path. This paper, which is the implementation of [1], can predict better than a LRU policy, without being on the hit critical path, with much lesser area and power consumption.

A large amount of data in cache is sitting idle, and is finally evicted after it moves from MRU to LRU position. Such blocks are called as dead blocks and are known to have no reuse. Blocks that will be referred again after a particular hit are called as live blocks. This paper tries to identify dead blocks in a LLC and presents them as victim as compared to least recently used blocks. There are a number of features associated with a dead block, as described in section II. We use these features to train our predictor using perceptron learning.

II. FEATURES ASSOCIATED WITH DEAD BLOCKS

A. Program counter

A PC is an indicator of how a program behaves. PC can be used as a feature for dead block prediction by learning which PC can lead to a block getting replaced. However, a PC might behave differently based on the context. As an example, a PC might have a reuse or might be dead based on the caller. Hence, we keep track of a few previous PCs to understand the behavior better.

B. Memory address

The actual memory the program is trying to access is a good indicator of what will be accessed next. If a particular block is evicted, it is safe to say that the blocks belonging to the same page would also be evicted soon. A last access to one block correlates to predicting it's the last access to the neighboring blocks as well. This fact can be used to predict dead blocks.

III. PREVIOUS WORK

There have been a number of cache predictors in the past:

A. Trace based predictor

Also known as reftrace predictor, this predictor keeps a hash of all the references to a particular block [3]. The corresponding signature is used to index a prediction table. A disadvantage of this mechanism on LLC is that the signature obtained is not very accurate in prediction because of the fact that the lower level caches filter out most of the locality.

B. Time based predictor

Also known as reuse distance predictor, this predictor classifies blocks as near-immediate re-reference interval, immediate re-reference interval and distant re-reference interval. Blocks with distant re-referral are expected to have little use and are replaced [4].

C. Dead block prediction

A dead block predictor consists of a sampler kept outside the cache, that keeps track of a few tags for some special sets [2]. The tags are maintained with a true LRU policy, but the cache itself can be maintained with any base policy. The sampler learns from evictions on these sampled sets to generalize it to the whole of cache to predict if a block will

have any reuse in the future. If not, it is predicted as dead and will be the victim for next miss on that set.

IV. PERCEPTRON LEARNING

This paper utilizes the concepts of perceptron learning to make a prediction. We do not actually use perceptron, but use the concept of perceptron learning to update the weight tables that are used for reuse prediction. Perceptron learning is based on the below two rules:

1) *Making a prediction:* A set of weights is choosen according to a criteria. If the sum of dot products of the weights and input features, *yout*, exceeds a threshold, the result is predicted to be true, else false.

2) *Updating the weights:* Since weights are the primary contributors in descision making, updating the weights accurately is the most important step. When the sum of weights is the less than a threshold and the actual outcome is true, the weights are incremented. When the sum of weights is more than a threshold and the actual outcome is false, the weights are decremented. The weights are saturated so that the predictor can easily adjust according to program behaviour.

We consider here the input vector to be a unit vector. Hence, the *yout* is just the sum of all weights.

V. THE ALGORITHM

Perceptron learning for reuse prediction algorithm is discussed in detail in this section.

A) Main idea

We consider 6 inputs from the program flow, which are discussed in detail later in this paper. These 6 inputs are XORed with the current PCs to obtain 6 hash values that index 6 weight tables. If the sum, *yout*, is more than a threshold, which is 128 for this implementation, the block is predicted dead. Else, the block is predicted alive. Each cache block has an extra bit to store this information. Figure 1 depicts this idea.

When an incoming block is to be placed, first, we check if the block is dead on arrival. If it is, it bypasses the cache. That is, the block is not placed in the cache. To make such a prediction, we compute *yout* for the current input features. If the value of *yout* is more than a threshold, which is 3 for this implementation, we bypass the block. Else, we search for a dead block in the set. If found, we evict it. Else we fall back on the default LRU policy to return the least recently used block. Figure 2 depicts the complete picture.

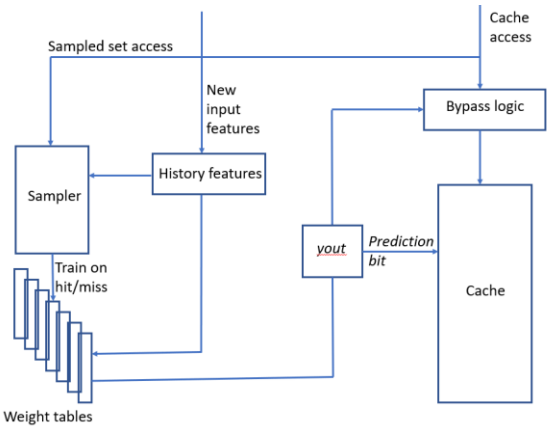

Figure 2: Accesses to sampler and cache

B) Input features

The input features are the features from the program that leads us to determine whether or not a block is dead. These, as discussed in section II, carry a correlation between current and future misses. We exploit these correlations in making a prediction. We use 6 features to index 6 tables while making a prediction. Suppose PC_i is the PC of *ith* recent access to LLC. Then, the feature set is:

1) PC_0 shifted by 2 bits.
2) PC_i shifted by i bits, i=1,2,3
3) Tag of current block shifted by 4
4) Tag of current block shifted by 7

All these features (except that last one) are XORed with the current PC to get the final hash values. The last 8 bits of the 6 hashes are used to index the 6 weight tables. 6 tables ensure that not all features are correlated with the program. Some weights might be negative saturated indicating a strong reuse behavior while others might be positively saturated indicating the least reuse behavior. The overall weights help in determining the best reuse behavior of the program. The weight tables have 1024 entries, hence we use 10 bits from each of the features to index the tables.

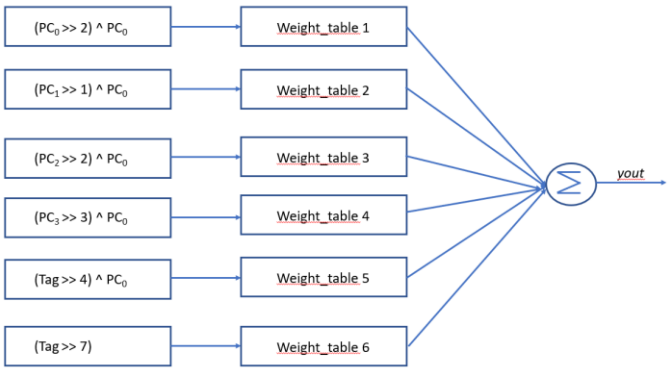

Figure 1: Computation of yout

C) The sampler

The sampler is the main idea behind this algorithm. A sampler is a separate structure that is maintained outside the

cache block. It consists of tags of a few selected sets from the cache. For this competition, we have considered every 16th block as a sampled set. For 4096 sets, we have 256 sets in the sampler. The sampler is accessed whenever there is an access to these special sets in the cache. Sampler has following fields:

1)*Partial tag:* Each block in the sampler has a partial tag (can be 15 bits). This tag is used to match the tag with an incoming block. A tag match is considered a hit and the predictor is trained that the features corresponding to this block results in a hit. If there is no tag match, the predictor is updated to learn that the corresponding features lead to a block being evicted, and hence, any such future references lead to a dead block.

2) *Input features:* Each block in the sampler is associated with 6 input features. These input features associated to a particular tag are placed in the sampler. These are then used during the update of predictor during a tag hit/miss.

3) *LRU bits:* Tags in sampler are maintained using true LRU policy. Upon a tag miss, the LRU victim is replaced with the new incoming block.

Note that we do not store *yout* as presented in [1]. This is based on the observation that the yout stored is outdated and does not help much in descision making. Instead we compute a new *yout* based on current sum to decide whether or not we want to make an update to predictor.

D) Updating a sampler

According to the original idea as presented in [1], the sampler is updated as follows:

1)*Tag hit:* When there is a tag hit in the sampler, compare the *yout* corresponding to the block to see if the value is less than a threshold, which is -74 in the implementation. If it is, do nothing. Else, decrement the counters corresponding to the hash values recorded in sampler.

2)*Tag miss:* A tag miss implies that the victim to be reaplced leads into believing that similar blocks correspond to dead blocks. Hence, the *yout* of the victim block is compared against a threshold, 74 in this implementation. If *yout* is greater than threshold, then do nothing. Else the weights cooresponding to hashes in recorded in sampler are incremented.

The sampler entries are updated with latest *yout*, tag and input features.

In this paper, we completely eliminate *yout* from sampler. Following is the update mechanism implemented:

1)*Tag hit:* A tag hit implies that the features present in the sampler lead to a hit. We do not use any threshold to decrement weights. Weights corresponding to features in sampler are always decremented on a hit. This is based on the idea that the counters must be kept at their low as possible to avoid them being predicted as dead. If one particular feature has a strong correlation and others don't, then a threshold can hinder the correlating feature from getting decremented. We avoid such a situation and let the counters decrement to their minimum upon hit.

2)*Tag miss:* We use a threshold here to ensure we don't predict dead blocks unnecessarily. However, we do not use *yout* from sampler as above. Instead we compute the new *yout* indexing weight tables from the features captured by sampler. This works better that the previous approach as the *yout* captured in the sampler is very old and no longer is correlates with the current weights. We also decrement the counter by 2 to make up for the lack of threshold on hit. This helps in adjusting to the program behaviour quicker. We maintain the threshold of 74 as above.

E) Prefetcher

A prefetcher helps reduce the number of misses by prefetching the data consecutive to a miss. However, there a few special requirements of the sampler to work well with a prefetcher:

1) *Sampler update:* Since the prefetch is an extra effort to bring in data that might or might not be used in future, data from prefetcher should not be used to train the sampler. The prefetch data just bypasses the update policy.

2) LRU update: The LRU of both sampler and the cache are not updated for a prefetch instruction. This is to make sure that we do not consider an extra fetch as a true data entry in block. When a victim is required, we would replace the prefetch data rather than data available in block until the prefetch data proves itself as a hit.

3) *Dead block:* We do not use dead block to place prefetched data. Instead we used default LRU policy.

VI. RESULTS

In this section we present the simulation framework and the results. We run 27 benchmarks from SPEC CPU 2006 on a single thread. The cache simulator is efectiu, that is being distributed for the sake of project.

The results presented below is speedup against LRU policy for various benchmarks. The geometric mean average of *6.4%* is obtained as compared to *3.5%, 3.8% and 6.1%* for SDBP, SHiP and perceptron learning.

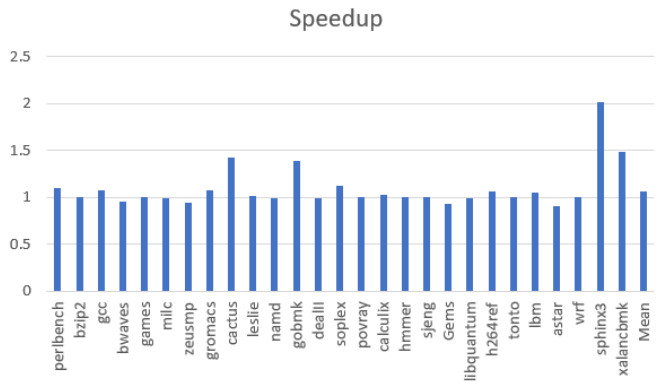

Figure 3: Speedup for various benchmarks and geometrical mean speedup

It can be observed that 14 out of 27 benchmarks have a speedup of more than 1. Bwaves, zeusmp, Gems and astar have a negative speedup of *5%*, *4%*, *6%* and *10%* respectively, which pulldown the overall speedup. The best performing benchmark is sphinx3 with a speedup of *100%*. Following it are xalancbmk, cactus and soplex with significant improvements in performance. Any improvements in algorithm should be such that it does not harm the negative speedup benchmarks much, and improves the speedup on benchmarks that are represented with high speedup values. Such improvements can be considered as generalized improvements and can be accepted.

VII. CONCLUSION

This paper describes the implementation of perceptron learning for reuse prediction algorithm for last level cache. It uses 6 features to index 6 distinct tables to obtain weights, which are trained using perceptron learning and can predict the reuse of a block. A few optimizations made in the paper result in a better speedup as compared to the original implementation reported in [1]. Future work can be to explore more such reuse behaviors to be able to predict dead blocks better.

ACKNOWLEDGMENT

Humble thanks to Professor Daniel A. Jimenez for presenting this opportunity to learn and implement state of the art cache replacement policy and for his valuable advice on understanding the concepts presented in [1].

REFERENCES

[1] Elvira Teran, Zhe Wang, Daniel A Jim´enez, "Perceptron Learning for Reuse Prediction," in Proceedings of the International Symposium on Microarchitecture, October 2016, pp. 1–12

[2] S. M. Khan, Y. Tian, and D. A. Jim´enez, "Sampling dead block prediction for last-level caches," in MICRO, pp. 175–186, December 2010.

[3] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, J. Simon C. Steely, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, (New York, NY, USA), pp. 430–441, ACM, 2011

[4] A. Jaleel, K. Theobald, S. S. Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA-37), June 2010.