Perceptron Learning with Reuse Prediction for Improved Cache Replacement

Gulshan Mandhyan, Department of Electrical and Computer Engineering, Texas A&M University, College Station, Texas

***Abstract*— The last level caches(LLC) with just the LRU replacement policy are quite ineffective due to dead blocks existing in the cache for a long time before they finally get evicted. A good replacement policy plays an important role in providing high performance for memory intensive programs.**

This paper discusses the implementation of perceptron learning with reuse prediction[4]. The main idea is to do reuse prediction using perceptron learning to detect whether a block is likely to be reused, providing the ability to perform improved placement, replacement and bypass in the LLC.

I. Introduction

Reuse predictors have been used in the past where features such as instruction, block addresses have been used to provide correlating data to predict whether a block would be reused in the future or not based on past accesses. The perceptron learning helps in extracting the accuracy out of these features for a better reuse prediction.

II. Related Work

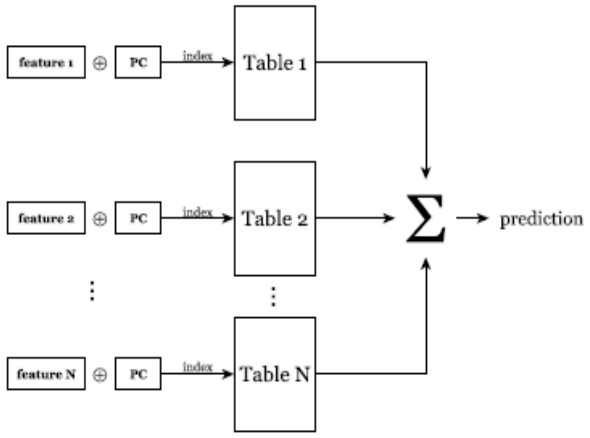
Reuse prediction mainly combines the two approaches of 'dead block prediction' and 'hit prediction' that have been used as the basis of many predictors for cache replacement and bypass in the past. For instance previous work has been done that predicts the distance to the next reuse, Re-reference interval prediction(RRIP)[1] uses the strategy of classifying blocks as near-immediate re-reference interval, intermediate re-reference intervals, and distant re-reference interval.

Then there are also Dead block predictors that predict whether a block will be used again before it is evicted. The most recent version of this is the Sampling dead block predictor of Khan[2]. Here basically we sample only a few sets of the cache into a sampler structure that is separate from the cache. We dead block prediction we use three predictor tables of 2-bit saturating counters that hashed using 3 different hashes of the PC of the relevant block being accessed. To make a prediction the weights are then added and sum is used to perform the dead block prediction. For every hit in the sampler set, the PC of the block is used to again index to the predictor table and the corresponding weights are decremented. Similarly for each eviction in the sampler set, the stored PC for the victim block is used to index the predictor table and the corresponding weights are decremented.

The Perceptron learning with reuse predictor also uses a sampler structure similar to the sampling dead block predictor and all the training is similarly done in the sampler itself. Perceptron learning has been proposed previously for predicting conditional branches[3]. This implementation also takes a similar approach where it uses the input features to index weights tables and updating the weights with perceptron learning.

III. The Main Idea

This predictor combines multiple features to make a reuse prediction of a block. Each feature is used to hash into distinct table of saturating weights that are then summed to make a prediction. If the sum exceeds a threshold the block is predicted not to be reused. The figure below shows the organization of a perceptron based reuse predictor. Each feature is hashed and then XORed with the PC to index into the corresponding predictor table.



The training is done only for the accesses in the sampler. When there is hit in the sampler we use the perceptron rule to update the weights, that is if the absolute value of sum is less than the training threshold or if there is a misprediction, we decrement the weights. Similarly on an eviction in the sampler, if the absolute value of sum is less than the training threshold or if there is a misprediction then we increment the weights.

Training with a Sampler

A sampler is an array of some sets of the LLC that are chosen to be represented by a set in the sampler. Each sampler block consists of the following fields:

1. A partial tag used to identify a block.
2. A signed integer sum(y_{out}) that represents the most recent prediction output for this block in the sampler. This is used for the perceptron update rule which checks for the magnitude of prediction output against a training threshold.
3. The hashes of the input features which will be used to index the predictor table and update the corresponding weights for the block.
4. LRU bits for replacement within the sampler.

IV. METHODOLOGY

The simulation infrastructure for the Cache replacement along with the traces were provided in class by professor D.A. Jiménez. The basic idea is that a 16-way set-associative cache simulator will call our code to decide which among 16 blocks in a set should be evicted when a replacement is needed. Our code will return a number from 0 to 15 to indicate the block number (way) of the victim block, or -1 to indicate that no block should be replaced, i.e. that the incoming block should bypass the cache. Our code will also be consulted when a block is referenced so that it may update its state.

A. Victim Selection

The first part of our implementation is Victim Selection in a set of the last level cache. To do this we perform the following steps:

1) Compute the hashes of the input features by XORing features with the PC and then taking a modulo by the predictor table size.
2) Use the hashes to index into the predictor table and compute the sum of the corresponding weights.
3) If the sum is greater than bypass threshold(T_{bypass}), then we bypass the block.
4) Otherwise the set is searched for a block predicted not to have reuse. If one is found, it is evicted.
5) Else the block chosen by the LRU policy is evicted.

B. Update Replacement Policy

The second part of our implementation is Update Replacement Policy. This includes the following design steps:

1) Compute the hashes of the input features by XORing features with the PC and then taking a modulo by the predictor table size.
2) Use the hashes to index into the predictor table and compute the sum of the corresponding weights. If the sum is greater than the replacement threshold($T_{replace}$) then the reuse bit of the block in LLC is updated as false, otherwise it is updated to be true.
3) If the access is to a set that needs to be sampled then the sampler access routine is called.
4) Lastly we update the PC History buffer of the LLC.

C. Sampler Access

The third part of the implementation is the Sampler Access routine where the training of the predictor is done:

1) We check if there is a tag match in the sampler for the accessed block.
2) If there is a tag match it means that the current set of features is correlated with reuse. Hence if the value of the y_{out} for the entry exceeds the threshold – θ, then the predictor table entries indexed by the hashes in the accessed sampler entry are decremented with saturating arithmetic.
3) If it's a miss then the set is first searched for an invalid block and if there is one the incoming block is placed at that position in the sampler.
4) If the complete set is valid, then the LRU victim in the sampler is found. Since this block is evicted it is unlikely to be reused. Hence we check if the y_{out} of the victim entry is less than a threshold θ, or if there is a misprediction, then the predictor table entries indexed by the corresponding hashes in the victim entry are incremented with saturating arithmetic.
5) At last the tag, valid bit, hashes and the most recently computed y_{out} is updated for the sampler entry that hit or missed.

D. Prefetches

The first issue found during the implementation was that the predictor provides a very poor performance with the prefetch accesses being treated as normal cache accesses. This is because the prefetcher issues many prefetches that hit in the cache which increases the locality of some blocks in the sampler sets and hence promote the blocks which should have remained near LRU. Hence since the prefetch address are not related to the instruction addresses, we use a fake PC address for all prefetches. Also we ignore the hitting prefetches for the update of our cache replacement policy.

Initially ignoring the prefetch accesses completely during Victim Selection and also during the update of our replacement policy provides better performance. During my experiment the geometric mean speedup over LRU increased from 1.008 to 1.0425.

Then after using fake instruction address for all prefetches and ignoring the hitting prefetches for the update of our cache replacement policy gives us the geometric mean speedup over LRU increase from 1.0425 to 1.0460

E. Features for the predictor

Different feature combinations were tried based on the suggestions of the paper and some based on the improvement we got in the Speedup over the traces provided. The six features selected for best Speedup were:

1. Current PC right shifted by 2
2. PC_i right shifted by i, where i=1to 3. PC_i indicates the ith recent PC accessing the LLC.
3. The tag of the current block
4. The tag of the current block shifted by 4

F. Sampler Set Structure

The sampler is just like a small cache with sets and blocks. Each block contains the tag for the block, 4-bits for the LRU position,

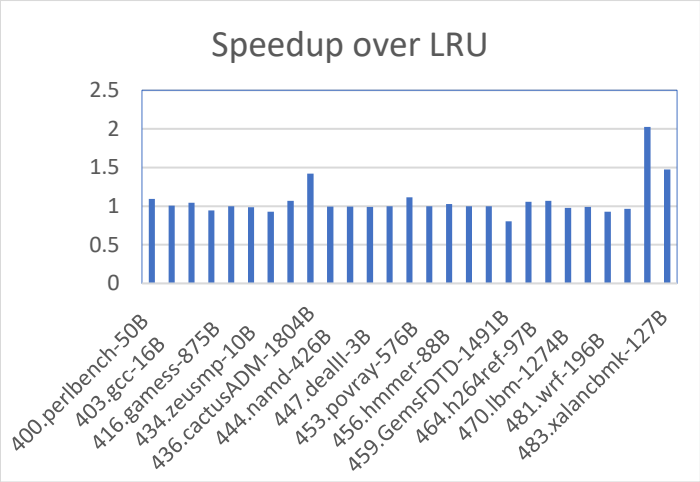
9-bit signed value of the last prediction for the block, a valid bit and six 8-bit hashes of the features.

G. *Hash function*

We need to choose the hash function for indexing into the predictor table. As suggested in the paper, each feature is XORed with the current PC and then a simple modulo by the predictor table proves to be a very good hash function that is sufficient for a very good performance.

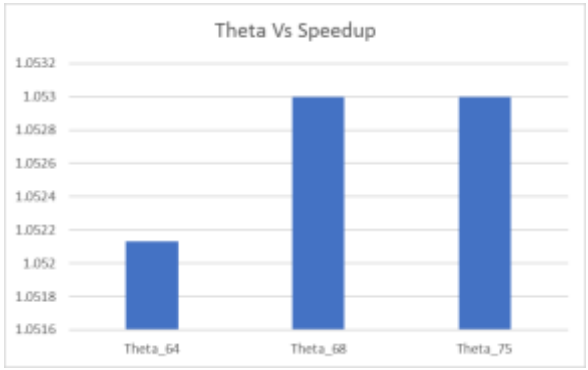
V. RESULTS

The section discusses the performance of my cache replacement policy over the LRU for the 27 SPEC CPU benchmarks provided by professor D.A. Jiménez. The figure below shows the Speedup over LRU for each of the traces.

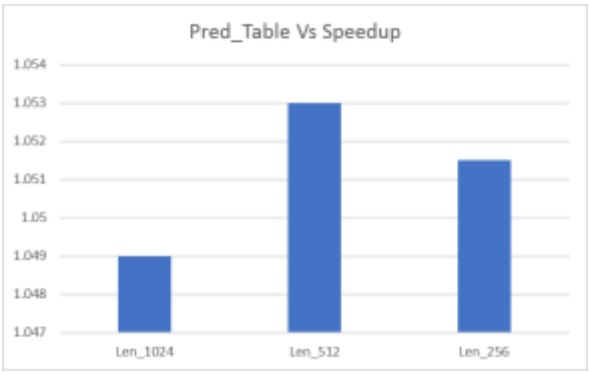


The geometric mean speedup over LRU achieved by the policy is 1.05302. Out of the 27 SPEC CPU benchmarks we notice that the perceptron exceeds the performance over LRU for 10 of the CPU benchmarks and matches the performance of LRU for the rest of the benchmarks.

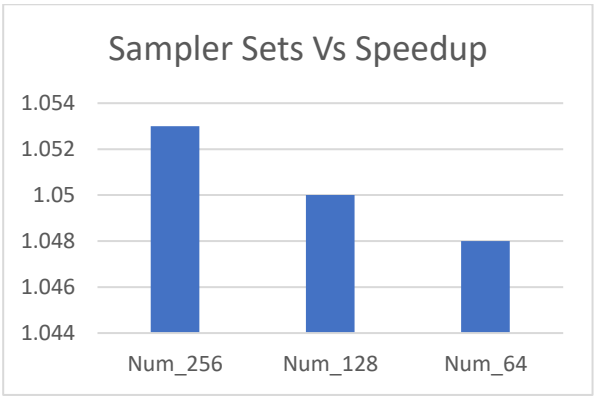
To achieve this speedup various parameters such as the number of sampler set, predictor table size, the perceptron training threshold and the replacement threshold were tuned for best performance.



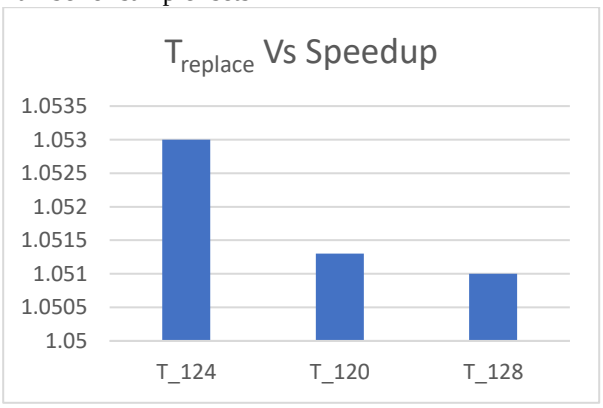
a. Geometric mean Speedup over LRU with varying θ



b. Geometric mean Speedup over LRU with varying predictor table size



c. Geometric mean Speedup over LRU with varying number of sampler sets



d. Geometric mean Speedup over LRU with varying $T_{replace}$ threshold value

VI. CONCLUSION

The paper proposes the usage of multiple features that provide correlating data to perform the reuse prediction of a block in cache. The perceptron learning helps us find these correlations between the multiple features in order to perform a better reuse prediction. For future enhancement, more features could be used to provide much more correlating data to improve the reuse prediction.

REFERENCES

[1] 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.
[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] D. A. Jim´enez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 197–206, January 2001.
[4] E. Teran, Z. Wang, and D. A. Jim´enez, "Perceptron learning or reuse prediction," in Proceedings of the 49th ACM/IEEE International Symposium on Microarchitecture (MICRO-49), October 2016.