Perceptron-learning for Reuse Prediction

Ankur Roy Chowdhury
Department of Computer Science
Texas A&M University
ankurrc@tamu.edu

***Abstract*— Last-level caches act as the last level of defense against the large disparity of memory access latency between the cache and the memory. Therefore, it is quite relevant to optimize the utilization of the LLC to enable maximum efficiency. Predicting block reuse and bypass is a way to do so. In this paper we talk about *perceptron-learning for reuse prediction*. The proposed predictor relies on the perceptron-learning algorithm and greatly improves prediction accuracy over baseline LRU. On a 4MB LLC with pre-fetching enabled, we obtain a geometric mean speedup of 3.1% over baseline LRU, on single-thread workloads.**

***Index Terms*— cache replacement, LLC, sampler, perceptron**

I. INTRODUCTION

The disparity between the last-level cache and the memory access latencies motivates the search for more efficient cache utilization policies.The aim is to have better utilization of live blocks - blocks that would be reused and are thus, useful. A reuse predictor predicts if a block is likely going to be reused later or if it is dead. The objective of prediction is usually achieved by correlating between various features, namely data and instruction addresses - thereby, helping us predict future events based on past activities. In this paper, we are going to discuss the perceptron learning algorithm for reuse prediction.

The rest of the paper is organized as follows:

- **Section 2** discusses the background of sampler-based cache replacement policies
- **Section 3** talks about the various algorithms
- **Section 4** details the infrastructure and talks about the results
- **Section 5** concludes the paper

II. BACKGROUND

A. Perceptron

A perceptron is the smallest building block of an artificial neuron[1]. It relies on a vector of signed weights and bias to predict an outcome learned through training. The output is the dot product of the weights and an input vector added to the bias. In classification problems, the output is passed through an activation function, which classifies the output based on a threshold.

The perceptron learning algorithm conceptually comprises of:

- A weighted sum comprising of a vector of chosen signed weights and inputs generate an outcome, y_{out}. If y_{out} exceeds the threshold, the prediction is true, otherwise false.
- The learning aspect comes into play when the weights are incremented or decremented based on the relation between the predicted and actual outcomes. If the outcomes disagree and there is a positive correlation between the input and the output, the weights are incremented. Otherwise they are decremented. If the predicted and actual outcomes match, we do not need to change the weights.

Over time, we reinforce this feedback loop of training and prediction, which ultimately leads to the weights being aligned to actual outcomes with a high probability.

The idea of employing perceptrons[2] in branch prediction lead to the idea of using them in cache reuse prediction. However, in the original implementation the set of input vectors was the global history of the branch outcomes, wherein the weights were the weight indices were the same across all the weight vectors, However, in subsequent implementations, the weight vector indices were skewed and the instead of a binary vector of inputs, the the input features were used to hash the indices to the weights. The weights were then simply added to compare against a threshold. In this paper, the input features are derived from the memory address, and the program counter addresses.

B. Sampler-based Policies

This paper details the prediction of block reuse and bypass using an auxilary data structure called the *sampler*. A sampler helps us maintain metadata for a few sampled sets in the cache. This approach lends to the idea that discovering correlating patterns between the access of cache blocks and their reuse in the set can be generalized for the entire cache based on a few sampled sets. The following characteristics are presented by a sampling-based reuse predictor:

- Memory access patterns are consistent across sets, so it is sufficient to sample a small fraction of references to sets to do accurate prediction.
- The sampler can be configured differently than the cache. The sampler could be 12-way associative, whereas, the cache is 16-way. Such a configuration would enable the sampler to evict unused blocks faster.
- Metadata associated with cache blocks for the predictor to work with is significantly reduced. Thus, reducing space and power requirements.

- Dead block prediction can benefit from a skewed organisation, as discussed above in Section II A.

Thus, combining the idea of a sampling-based reuse predictor with perceptron learning is the basis of this paper.

III. Perceptron-learning reuse predictor

A. Organization

Before we discuss the algorithms involved, the current section details the various bits of information that needs to be maintained in our predictor.

- **1 bit** per cache block for reuse prediction
- **15 bits** per cache set for tree-based PLRU (16-way associativity)
- **15 bits** for maintaining a partial tag match in each block of the sampler
- **9 bits** for the most recent computation of the prediction output for each block of the sampler
- **8 bits** per feature to maintain a sequence of hashes used to index the predictor's tables
- **4 bits** per sampler block to maintain LRU stack position within the set
- **1 bit** per sampler block to maintain valid status

Apart from the above mentioned requirements, we need to maintain a separate weight table per feature tracked. In this paper we are tracking six features. We need **8 bits** per weight and 256 weights per table.

B. Features

The following six features were used for prediction:

- **Feature 1** PC_0 shifted right by 2
- **Feature 2** PC_1 shifted right by 1
- **Feature 3** PC_2 shifted right by 2
- **Feature 4** PC_3 shifted right by 3
- **Feature 5** Tag of the current block shifted right by 3
- **Feature 6** Tag of the current block shifted right by 7

Here PC_i denotes the i^{th} most recent PC, where PC_0 is the most recent PC address.

Apart from the features, the following threshold values were used:

- $\theta = 68$
- $\tau_{bypass} = 3$
- $\tau_{replace} = 124$

C. Algorithms

The following sections would go over the various algorithms, namely ones for victim selection, and cache access.

1) Victim Selection: The prediction algorithm is used to get a victim block on a cache miss. The algorithm returns a way id indicating a dead block with the corresponding way, or if the incoming block is predicted to be dead on arrival, -1 is output to indicate a bypass. The steps involved are:

- Compute the current features
- Use the prediction algorithm on the current set of features
- If the prediction output is above τ_{bypass}, predict a bypass and return -1
- If the prediction output is below τ_{bypass}, search the corresponding set for a dead block: loop through all the blocks to find a block with a reuse bit set to false. If such a block is found, return it's way id. Otherwise, proceed to select a victim candidate through the PLRU policy.

Algorithm 1: Victim Selection Algorithm

Input: set_index, pc, address
Output: way: $[0, 15] \cup \{-1\}$

```
features:= compute_features(pc, address);
prediction := predict(features);
if prediction > τ_bypass then
    way := -1;
else
    for j ← 1 to 16 do
        if reuse[j] == false then
            way := j;
        else
            /* if no deadblock found     */
            way := PLRU_get(set_index);
        end
    end
end
```

2) Cache Access: Whenever there is a cache access the following steps take place:

- Check to see of the incoming set is a sampler set
- If it is a sampler set, loop through the set to find a tag match with the incoming tag
- If a tag match is found, check to see the sampler block's previous prediction output. If the output is less than $-\theta$ and the prediction outcome does not match with the actual outcome, train the predictor on the current features by decrementing their weights
- If a tag match is not found, we need to evict a block from the sampler. Loop through the set to find an invalid block. If not found, loop through through the set to find a dead block, i.e a block with $y_{out} > \tau_{replace}$. If still not found, get eviction victim based on LRU stack position.
- Once victim has been determined in the step above, train the predictor if the selected block's $y_{out} < \theta$ and prediction outcomes mismatch. While training, the weights need to be decremented.
- Update the sampler block with the new features and prediction output, and set it's valid bit to $true$.
- Update the cache's PLRU status.
- Calculate the cache block's reuse_bit by checking it's prediction output against $\tau_{replace}$. If $y_{out} > \tau_{replace}$, set reuse_bit to $false$, else $true$.

IV. Results

The cache simulator was implemented on the *efectiu* infrastructure. The infrastructure includes the reading of trace

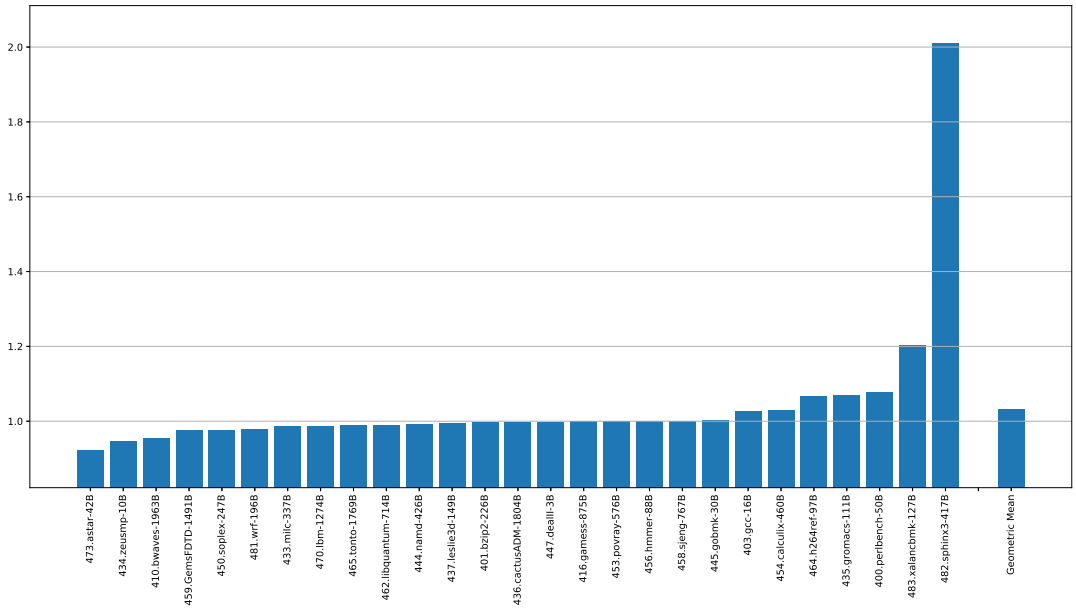

Fig. 1. Geometric Mean Speedup over LRU

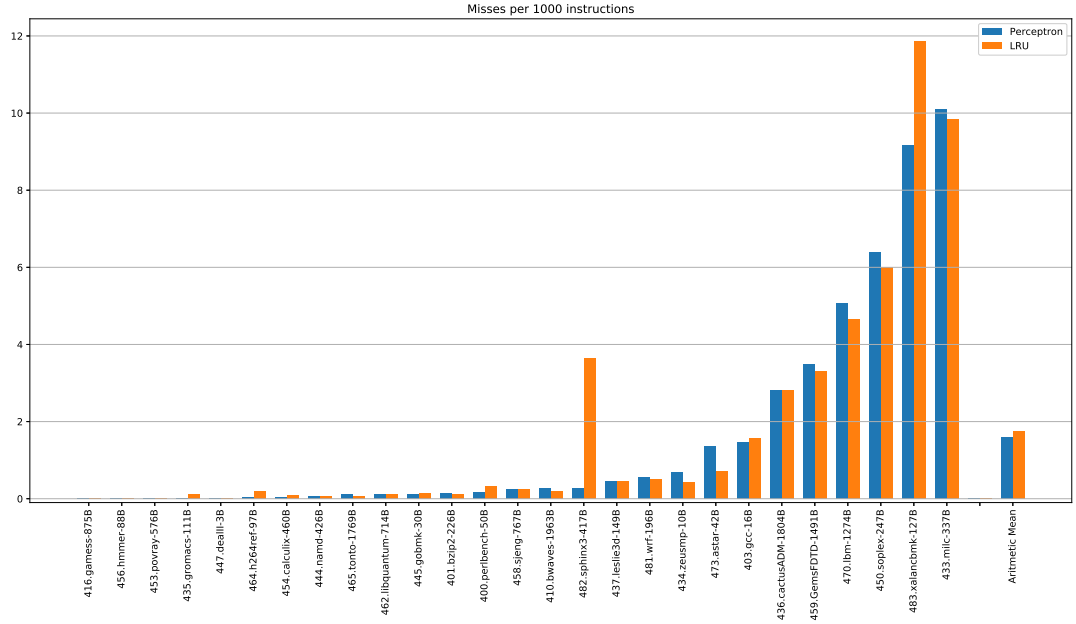

Fig. 2. Misses per 1000 instructions

files, line by line. The traces have pre-fetching instructions as well.

The infrastructure has two interfaces, one to *select victim* and one to *update cache predictor*. The *select victim* interface is called when there is a cache miss and a victim has to be selected. The *update cache predictor* interface is called on every cache access.

Running the simulator on the given set of 27 traces, the predictor yielded a geometric mean speedup of **3.06%** over baseline LRU. Fig. 1 and 2 details the geometric mean speedup and the misses per 1000 instructions of the predictor, respectively.

Algorithm 2: Cache Access Algorithm

```
Input: set_index, pc, tag, outcome, way_id
PLRU_update(set_index, way);
set[way]_reuse_bit := predict(features);
if is_sampler_set(set_index) then
    block_exists := false;
    for j ← 0 to sampler_assoc do
        if block_tag == tag and block == valid then
            block_exists := true;
            if block_y_out > −θ or outcome != set[way]_reuse then
                train(block, increment);
            else
        else
    end
    if !block_exists then
        way := -1;
        for j ← 0 to sampler_assoc do
            if block != valid then
                way := j;
            else
        end
        /* no invalid block; search dead block */
        for j ← 0 to sampler_assoc do
            if block_y_out > τ_replace then
                way := j;
            else
        end
        /* no dead block; use lru */
        way := LRU_get(set_index);
        train(block_way, decrement);
    else
else
end
```

V. Conclusion

In this paper we implemented the Perceptron-learning technique for reuse prediction. We started by extending the idea of Perceptron-learning from branch prediction to a cache utilization policy. To augment this technique, we used a sampling based method for generalizing cache behaviour.

The implemented policy exceeded the baseline LRU policy by a large margin in terms of performance.

Acknowledgment

I would like to thank Prof. Daniel A. Jiménez for teaching me about cache replacement policies; and encouraging me to explore modern cache utilization techniques by organizing a project-based contest on maximizing cache utilization.

References

[1] H. D. Block, "The perceptron: A model for brain functioning," *Reviews of Modern Physics*, vol. 34, pp. 123–135, 1962.

[2] D. A. Jiménez 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.

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

[4] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron Learning for Reuse Prediction", *Proceedings of The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-2016)*, Taipei, Taiwan, October 2016.