CS6600: Computer Architecture
Assignment 3: Perceptron based Branch Predictors

Bachotti Sai Krishna Shanmukh EE19B009
Akshat Joshi EE19B136

November 21, 2022

1 Introduction

Branch predictors are hardware modules used in a super-scalar processor to avoid stalls by executing instructions in a speculative fashion. It aims to reduce the number of cycle stalls due to branching by predicting the , thus increasing the average IPC of the processor. However it is essential to build a highly accurate branch prediction module as the penalty of misprediction scales with the depth of the pipeline and the processor has to roll back to the state before branching to ensure functional correctness.

2 Literature Review

2.1 Bimodal & GShare Predictor

One of the naivest implementations is to maintain a 2-bit saturating counter (bimodal predictor). The instruction address is hashed to a counter and the prediction is made based on the state of the counter. If the value of the counter is greater than or equal to two then it predicts that branch as taken for that address, and not taken otherwise. The state is later updated when the branch instruction executes and outputs the actual result. The major drawback of bimodal predictors is when two different branches index to the same counter and can cause a deadlock of mispredictions due to aliasing. Gshare is a branch predictor which again uses two bit counters but also resolves the earlier mentioned issue. The instruction address is first hashed into a local history table, which maintain a two bit information of the recent branch taken/not taken for instructions with the same hashed index. This local history value is indexed into a predictor table of 2-bit counters and the counter value gives the prediction. The implementation of this algorithm can be made simpler by using one single global history table and hashing the xor-ed ip, shifted ip and local branch history. This can be further extended to an (m,n) branch predictor with an increased capacity of learning any boolean function. However the hardware budget scales exponentially with the branch history length.

2.2 Perceptron based predictor

The perceptron based predictor give an advantage of efficient implementation in hardware and scales linearly with the history length. Perceptrons, unlike other neural network based predictors, have no derivative compute and back-propagation. The input history vectors are composed of bipolar elements i.e., -1 and +1 for branch not taken and taken respectively. The weights of the perceptron convey information about the rule being learnt/imposed on prediction. One of the simplest approach is to have a single layer perceptron indexed from a table using a hash function. If the output of perceptron is positive, then the branch is predicted to be taken. The weights of the perceptron are later updated based on the actual result of the branch. Weights which contribute to correct decision are rewarded and the other ones are penalized.

$$y_{out} = w_{bias} + \Sigma_i x_i \times w_i \tag{1}$$

This predictor outperformed all history tables based predictor approaches like TAGE, Tournament predictor and Gshare. However, perceptron based predictors can learn only linearly separable boolean functions. Also, the problem of destructive aliasing exists in this approach. When same history lengths are maintained in G-share and perceptron, the former shows better accuracy due to it's potential to learn any boolean function.

3 Double Perceptron: A Proposed Perceptron based Predictor

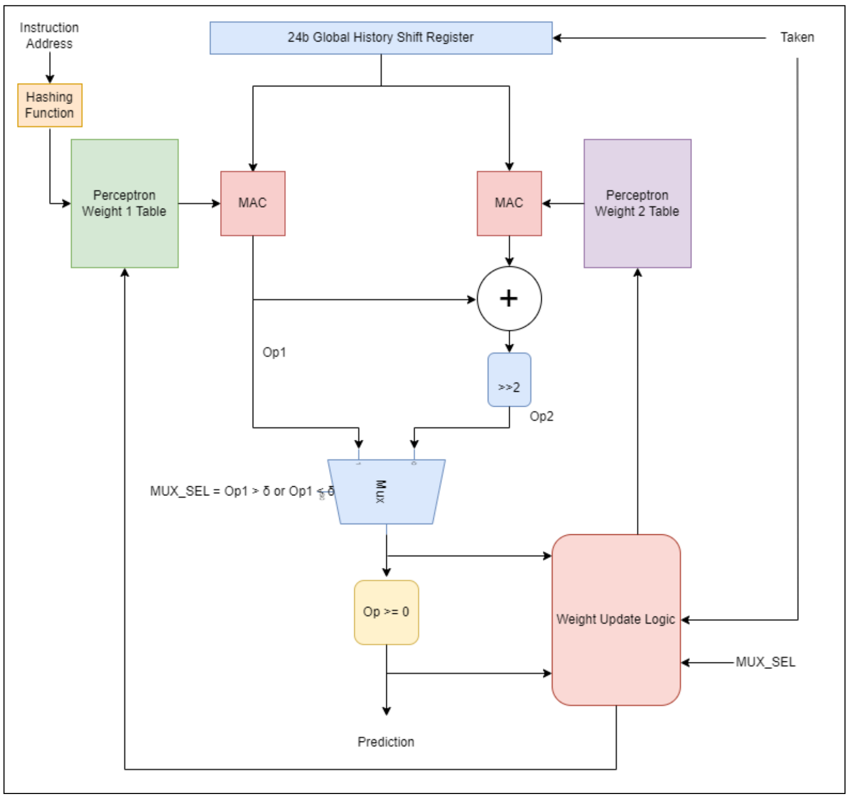
Instruction Address
24b Global History Shift Register
Taken
Hashing Function
Perceptron Weight 1 Table
MAC
MAC
Perceptron Weight 2 Table
+
Op1
>>2
Op2
MUX_SEL = Op1 > δ or Op1 < δ
Mux
Op >= 0
Weight Update Logic
MUX_SEL
Prediction

Figure 1: Proposed Double Perceptron Branch Predictor

3.1 Why Double Perceptron?

Single layer perceptron predictors try to model the problem as partition of n-dimensional space using a hyperplane. If a particular history sequence is predicted to be taken, then all prediction sequences which lie on the same side of hyperplane, as that of the mentioned sequence, will output the prediction as taken, whereas all prediction sequences that lie on opposite side are predicted to be not taken. This smooth continuous hyperplane keeps moving in space as we update the weight values, yet it cannot partition the input space in a non-linear way. The ground truth function of all branch instructions and their branch status need not be linear or approximately linear.
In double perceptron we maintain an extra layer of weights which also perform a dot product on the global branch history. The output is computed using the following way:

$$y_{out} = \begin{cases} y_1 = w_{bias1} + \Sigma_i x_i \times w_{1i} & -\delta < y_1 < \delta \\ y_1 + w_{bias2} + \Sigma_j x_j \times w_{2j} & otherwise \end{cases} \tag{2}$$

If the output of the first perceptron yields a value within the delta confidence threshold, which is a hyper-parameter, then the branch prediction is more likely to be a misprediction than the output beyond the confidence. In a scenario where $0 < y_1 < \delta$, the second layer is in active mode and the second perceptron tries to learn if the sequence truly wants to lean on the positive side of prediction or get pushed to the negative side. When visualized in N-dimensional space, our prediction function now creates a zig-zagged piece-wise continuous planes in space in all dimensions, instead of single hyperplane, with an aim to attain prediction outputs greater than confidence value δ. This construct induces non-linearity in the function, and the weights of the second layer are learnt in a way such that it boosts the output confidence to correct side.

3.2 Weight Update Logic

The weights and bias of the layer 1 are updated in the same manner as in that of single layer perceptron. The step size for layer 1 weights is 2 and for bias term is 1. The range of weight values is limited by defining a maximum value and a minimum value. The weights of layer 2 are updated with a step size 2 only if the second layer is activated during the prediction. If the prediction output is less confident than the delta, then the weights of the layer 2 are updated with a step size of 4. This ensures the boosting property of layer 2 and faster convergence to a more confident prediction.

```
if( sgn(y_out) != taken || abs(y_out) < THETA)
    wi1 = wi1 + 2*taken*xi // taken = -1 implies not taken
    bias = bias + taken
if (layer_2 is activated && abs (y_out) < DELTA)
    wi2 = wi2 + 4*taken*xi
if (layer_2 is activated && sgn(y_out) != taken) // highest penalty for misprediction when
    wi2 = wi2 + 6*taken*xi
```

A FIFO buffer is also maintained to queue in the global history register state, the instruction address and their corresponding prediction, and will be queued out when the actual branch status is known and the weights are updated correspondingly.

3.3 Performance and Hardware Budget

For the ChampSim simulation, the global history register is chosen to 24 bits, number of perceptrons as 163 and the bit-width of each weight is 8 bits. The FIFO size is limited to 100 entries. The total hardware budget is approximately 8.85KB (1.86x single layer perceptron). The predictor latency remains approximately same as in case of a single layer perceptron as the second layer computation happens in parallel with first layer. Hyper-parameters like θ, δ are chosen by iteratively checking for the best branch prediction accuracy and least MPKI (misses per kilo instructions).

4 GL-Double Predictor: Aim to tackle the aliasing pitfall

Although the initially proposed double perceptron aims to get more confident predictions, it doesn't solve the problem of aliasing. The instruction address is hashed into a global perceptron table and instructions with same hash value can potentially cause destructive aliasing in the perceptron weight learning which can lead to poor accuracy. To account for this issue, we further extended our design to include a local history table which maintains an 8 bit history for each hashed index. Together, the 8bit local history and 24b global history is sent as input to perceptron for branch prediction. The GL-Double predictor has higher hardware overhead due to the local history table and increased number of perceptron weights. It also increases the latency of prediction due to increased input size for perceptron.

5 Results

The following results were recorded in ChampSim with 50 million warmup instructions and 50 million simulation instruction. The traces were bench-marked on bimodal, Gshare predictor with 14b history length and perceptron predictor with 24b history length. Our predictors outperformed g-share and bimodal on all traces. When compared with the perceptron based predictor, the MPKI and accuracy are same for few traces and slightly dropped for few more. The g-share has slightly better accuracy than all other predictors in the milc trace. On the whole, the perceptron based predictor has the best accuracy and least MPKI. Between the GL-Double and Double perceptron predictor, the latter one gave lower MPKI on average. However on traces like Bwaves, gamess, mcf and namd the GL Double predictor showed better accuracy than the Double perceptron. One of the possible pitfalls of the proposed predictors is highly fluctuating initial weights causing more mispredictions at the beginning of the program. This can be countered by analysing a good initialization of weights for the perceptrons and tuning the hyper-parameters to get the best results.

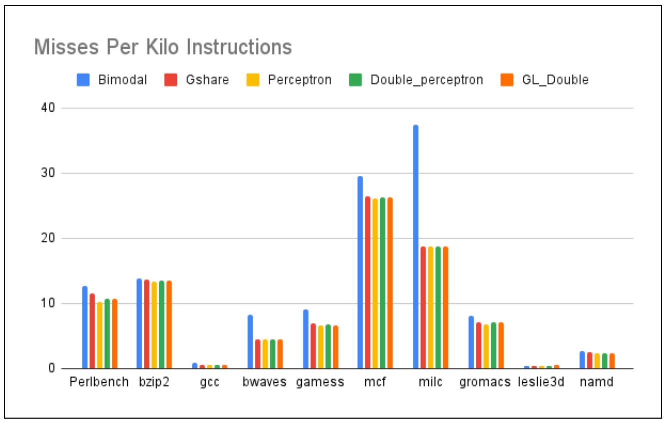

Figure 2: Misses per Kilo Instructions

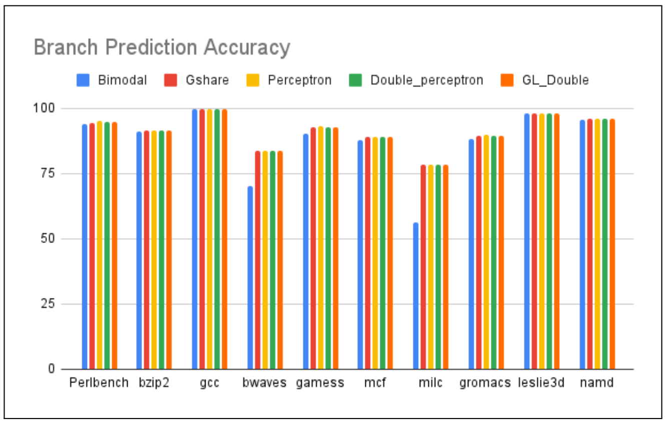

Figure 3: Branch Prediction Accuracy

More statistics about the predictor's performance on traces can be found here

6 References

- A Survey of Deep Learning Techniques for Dynamic Branch Prediction
- Dynamic Branch Prediction with Perceptrons