Piecewise Linear Branch Prediction Implementation [*]

Mohit Upadhyay
mohitu@comp.nus.edu.sg

February 19, 2020

Abstract

Branch prediction is a crucial part of the traditional processor pipeline. The performance of branch predictors are very critical in current processors which have very deep pipelines. This project report implements the Piecewise Linear Branch Prediction Implementation on the Sniper Multi-Core Simulator.

The report also provides a limit study of the Piecewise Linear Branch Predictor that has been implemented by changing different tunable parameters in the branch prediction logic.

1 Introduction

In current generation processors, the number of pipeline stages are significantly high. Intel Pentium architecture had initially 5 pipeline stages in the original Pentium architecture. Currently, the Intel Skylake architecture has 14-15 pipeline stages in each of its cores. Due to the increasing number of pipelines in the architecture, branch prediction latency is becoming extremely critical for the performance of the processors. Hence, branch predictors performance need to be continually improved in order to achieve higher performance.

In this work, the idealised version of the Piecewise Linear Branch Predictor[Jimenez (2005)] has been implemented. There is another version practical version of the same which has been implemented. The practical implementation of the branch predictor removes the two assumptions which the ideal version took into account. The implementation done in this work removes one of the assumptions which was fairly obvious. However owing to the constraints in time, the second assumption was not removed. Hence, the branch predictor implemented in this work is same as the idealized case.

2 Branch Predictor Description and Implementation

This section describes the branch prediction algorithm of the Piecewise Linear Branch Predictor in detail.

2.1 Description of the Algorithm

The algorithm has two parts: a predict function and an update/train procedure. The following variables are used in the algorithm:

[*]This Report is a part of CS5222 : Advanced Computer Architecture module at the School of Computing, National University of Singapore

```
function $predict$ ($address$: integer): boolean
begin
    $output := W[address, 0, 0]$                        (* Output is initialized to bias weight *)
    for $i$ in $1..h$ do                                (* Find the sum of weights (or their negations) chosen *)
        if $GHR[i] =$ true then                         (* using the addresses of the last h branches *)
            $output := output + W[address, GA[i], i]$   (* If the $i^{th}$ branch in the history was taken, *)
        else                                            (* add the chosen weight *)
            $output := output - W[address, GA[i], i]$   (* otherwise subtract it *)
        end if
    end for
    $predict := output \geq 0$                          (* Predict the branch taken if the output is at least 0 *)
end
```

Figure 1: Prediction Algorithm

- W : A three dimensional array of integers. the indices of this address are the branch address, address of the branch in the path history and the position in the history itself. *W[B,0,0]* is the bias weight for the the Branch with the address B. The addition and subtraction on elements are saturated at 127 and -128. Hence, each element of W is an 8-bit value. In the idealised case, it is assumed that W is large enough to span the entire address space. But in the implementation done in this work, we hash the values of the first two dimension of W to specific values.
- h : Global History Length
- GHR : Global History Register. It keeps track of the outcomes of the previous branches as they are executed in form of an array of outcome values. Branch outcomes are shifted to the first position of the array as they are executed
- GA : Array of Addresses which keeps the addresses of the branches as they are executed. Branch addresses are shifted into the first position of this array as they are executed. GA and GHR together give the path history for the current branch to be predicted.
- output : An integer value which is the value of the linear function used to predict the branch to be predicted.

Figure 1 shows the predict function which computes the outcome for the current branch. The function accepts the address of the branch to be predicted as a parameter. The branch is taken if predict function returns true, else the branch is not taken. Figure 2 shows the train/update procedure which is used to update the predictor as the branches are executed. The training algorithm uses a parameter θ as a threshold to stop updating the parameter. This algorithm is similar to other neural predictors such as the perceptron predictor[Jimenez and Lin (2001)] and the path-based neural predictor[Jimenez (2003)]. The value of θ is computed using the same formula from the path-based neural predictor paper, $\theta = 2.14*(h + 1) + 20.58$.

2.2 Implementation

The above branch prediction logic was used along with the Sniper Multi-Core Simulator[Carlson, Heirman, and Eeckhout (2011)]. The limit study focuses on the branch mis-

```
procedure train (address: integer; taken: boolean)
begin                                                  (* If magnitude of output is less than $\theta$ or prediction was *)
    if $|output| < \theta$ or $predict \neq taken$ then                (* incorrect then update the weights *)
        if $taken$ = true then
            $W[address, 0, 0] := W[address, 0, 0] + 1$         (* If branch was taken, then increment the bias weight, *)
        else
            $W[address, 0, 0] := W[address, 0, 0] - 1$         (* otherwise decrement it (with saturating arithmetic) *)
        end if
        for $i$ in $1..h$                                      (* For each address and branch outcome in recent history... *)
            if $GHR[i] = taken$ then                           (* If the $i^{th}$ most recent outcome is equal to current outcome *)
                $W[address, GA[i], i] := W[address, GA[i], i] + 1$   (* then increment the weight that contributed to this prediction *)
            else
                $W[address, GA[i], i] := W[address, GA[i], i] - 1$   (* otherwise decrement it (with saturating arithmetic) *)
            end if
        end for
    end if
    $GA[2..h] := GA[1..h-1]$                                   (* Shift the current address into the global address array *)
    $GA[1] := address$
    $GHR[2..h] := GHR[1..h-1]$                                 (* Shift the current outcome into the global history register *)
    $GHR[1] := taken$
end
```

Figure 2: Training Algorithm

prediction percentage and also on the MPKI(Misses per 1K instructions) statistics which are shown in the next section. For this limit study, the benchmarks that are focused are the astar, gobmk, sjeng, mcf and bzip2 benchmarks from the SPEC2006 CPU benchmark suite[Henning (2006)]. These are generally known to have high MPKI values.

In this implementation as described above, the first two indices of the W array are restricted to certain values(n for the first index and m for the second index). So in this implementation, W is a *n * m * (h+1)* array. GHR and GA are one dimensional arrays with size *h*. The bias weight is kept at *W[B mod n,0,0]*.

As discussed in the paper if the value of m = 1, then the Piecewise Linear Predictor is basically a Perceptron Predictor. If the value of n = 1, then the Piecewise Linear Predictor is a Path-based Neural Branch Predictor. This is the most interesting case of the Piecewise Linear Branch Predictor that it is basically a Generalized Neural Predictor. Only by changing the hash values(n and m) of the indices of W matrix, it can be made to act like the Perceptron predictor which predicts based on the global history or like the Path-based Neural Branch Predictor which predicts based on each branch's local history. The more interesting aspect of this predictor is that by choosing a optimised value for n and m, it can outperform both the Perceptron predictor and the Path-based Neural Branch Predictor.

3 Results

The Piecewise Linear Predictor is used with astar benchmark along with changing values of n and m values (first and second indices of W array) as seen in Figure 3 to observe the behaviour of the branch predictor in term of the misprediction percentage. The values of n and m are taken such that their product is always 64K. As discussed, the case where m = 1 is a Perceptron Predictor and the case where n = 1 is a Path-based Neural Branch Predictor.

The effect of history length(Figure 4) on the Piecewise Linear Predictor was also considered.In the paper, the performance keeps improving on increasing the history length.

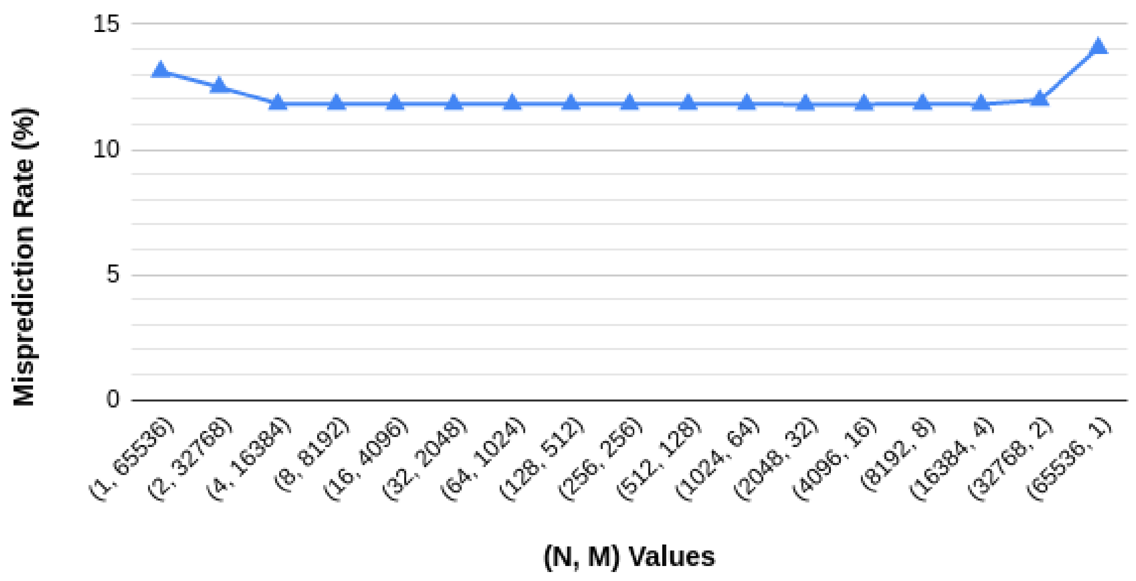

Figure 3: Misprediction rates with different values of n and m

But for the case considered here, there is an optimum value obtained.

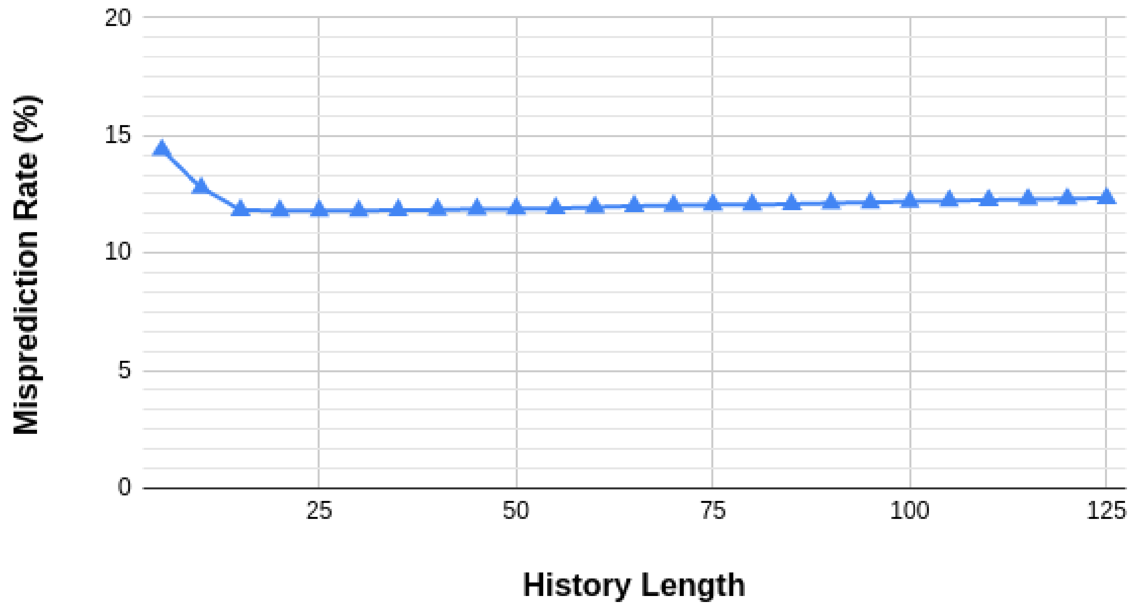

Figure 4: Effect of History length on misprediction rate

Other than this, the five SPEC 2006 benchmarks namely astar, gobmk, sjeng, mcf and bzip2 benchmarks are used in this implementation as these are generally known to have high MPKI values as shown in Figure 5. The Piecewise Linear predictor is compared with the Pentium Branch predictor, the one-bit branch predictor which are already a part of the Sniper Simulator Infrastructure and also with the Perceptron Predictor (m = 1) and Path-based Neural Branch Predictor (n = 1).

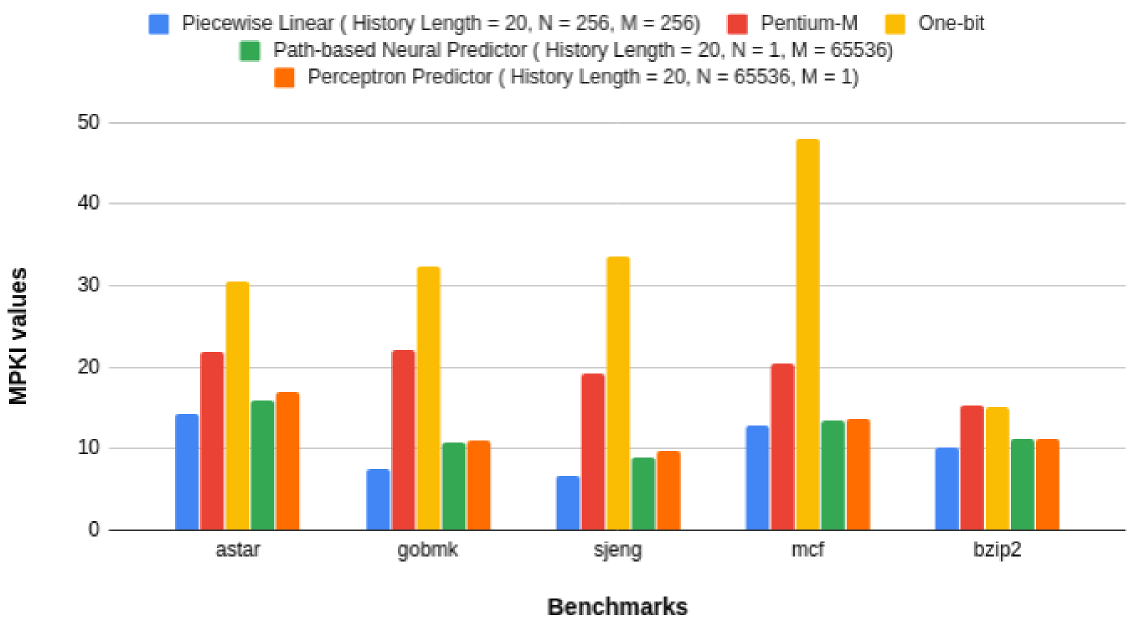

Figure 5: MPKI Miss chart for the five SPEC2006 benchmarks

4 Discussion and Conclusions

From the results , it can be seen that the Piecewise Linear Predictor outperforms all the other predictors for all the benchmarks that are considered. We can also see that how the history length affects the performance of the predictor. In the paper it was found that for the Piecewise Linear Predictor, the performance keeps improving on increasing the history length. But for the case considered here, there is an optimum value obtained. In this work, the optimum value of history length obtained is around 20. This was the only difference in the results that was observed.

As the paper proves, the Piecewise Linear Branch Predictor is basically a Generalized Neural Predictor and hence combines best of both worlds to get improved performance.

References

Carlson, T. E., Heirman, W., & Eeckhout, L. (2011). Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis.* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2063384.2063454 doi: 10.1145/2063384.2063454

Henning, J. L. (2006, September). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, *34*(4), 1–17. Retrieved from https://doi.org/10.1145/1186736.1186737 doi: 10.1145/1186736.1186737

Jimenez, D. A. (2003, Dec). Fast path-based neural branch prediction. In *Proceedings. 36th annual ieee/acm international symposium on microarchitecture, 2003. micro-36.* (p. 243-252). doi: 10.1109/MICRO.2003.1253199

Jimenez, D. A. (2005). Piecewise linear branch prediction. In *Proceedings of the 32nd annual international symposium on computer architecture* (p. 382–393). USA: IEEE Computer Society. Retrieved from `https://doi.org/10.1109/ISCA.2005.40` doi: 10.1109/ISCA.2005.40

Jimenez, D. A., & Lin, C. (2001, Jan). Dynamic branch prediction with perceptrons. In *Proceedings hpca seventh international symposium on high-performance computer architecture* (p. 197-206). doi: 10.1109/HPCA.2001.903263