Implementation and Review of *Fast Path-Based Neural Branch Prediction*[1] NUS CS5222, Project 1

Luca Krueger

March 18, 2020

1 Introduction

This project is related to the course CS5222 Advanced Computer Architecture (National University of Singapore) and reviews the idea of *Fast Path-Based Neural Branch Prediction*[1] introduced by Daniel A. Jimenez. This review focuses more on the accuracy of the introduced branch prediction technique, than on hardware usage and timing. The corresponding paper provides different reasonable calculations on feasible hardware implementations and discusses hardware usage and timing constraints. This project covers a full implementation of a Fast Path Based Neural Branch Predictor (FPBP), which is later simulated within the *Sniper Multi-core Simulator*[2] using different common benchmarks. Neural networks nowadays are widely used for complex classification problems, which appeared to be unsolvable in the past. The networks can develop giant dimensions and excessive computation requirements. Both do not seem to apply to the problem of branch prediction. Branch predictors have a incredible small time window for a single decision, strong hardware constraints and the problem of branch prediction is already solved to a for most cases sufficiently small misprediction rate. Nevertheless, branch prediction saturated over the past years. Only a few new successful branch prediction techniques were published. Neural branch prediction might have the ability to give this field of research another boost with new accuracy improvements. Some chip companies claim to have a neural branch predictor in their most recent processor, but do not reveal any details about their architecture and accuracy improvements. Knowledge from past research in the field of deep neural networks, which achieve remarkable results, cannot be applied here, because of hardware restrictions. This bring us back to simple perceptrons, the basic blocks of every neural network. The FPBP realizes a combination of perceptrons to process available information such as the current branch address and past branches to a precise branch decision output. By cascading perceptrons, required calculations can also be cascaded without any additional latency tradeoff, which is a smart improvement.

2 The prediction algorithm

The author provides a pseudo-code implementation of their prediction algorithm in the paper. The description along the code only reveals the most important details and gives a rough intuition of its functionality. Unfortunately, a clear explanation or derivation is missing in the paper. Therefore, the underlaying design choices made by the author is unknown. In the following section I would like to derive the algorithm, which hopefully helps to better understand its mechanisms. For simplicity, similar variable naming used here as proposed in the paper:

$$\begin{aligned} h &: \text{ history length} \\ hashlen &: \text{ length of the branch address register} \\ \hat{x} = \begin{cases} 1 & \text{taken} \\ -1 & \text{not taken} \end{cases} &: \text{ speculative prediction outcome} \\ W \in [0,h] \times [0, hashlen-1] \times \mathbb{Z} &: \text{ weight matrix} \\ SR \in [0,h] \times \mathbb{Z} &: \text{ speculative result history register} \\ i = pc \mod hashlen &: \text{ hashed branch address } pc \end{aligned}$$

Forward For a traditional perceptron predictor a vector of past branching decisions $\hat{x}(t-j) : j \in [0, h-1]$ forms the inputs and is multiplied by a vector of weights $W \in \mathbb{Z}^{h+1}$ and biased by $W[0]$ to calculate the dentritic potential $y(t)$. The perceptrons output is the binary decision $y \geq 0 \rightarrow \{\textit{taken}, \textit{not taken}\}$.

The FPBP derives from the traditional perceptron predictor, but introduces some significant structuaral changes. It now uses a two dimensional weight matrix W, which might give the first intuition that the single perceptron is now extended to a full single layer of *hashlen* perceptrons, so called Hidden Layer, but this is not the case. Another might intuitively think that the algorithm uses separate perceptrons for different branches within a range of hashed branches, but the outcome of the FPBP does not depend on separate identifiable perceptrons. It depends on a path of previous branches, which is identified by their hashed branch address.

This comes with the advantage, that the sum over the path can be calculated iteratively prior to when the final branch decision is required. The branch outcome for a new branch with hashed branch address i then only depends on the accumulated sum $SR[h]$ biased by $W[i,0]$ and therefore does not cause any additional delay. Using the algorithm from the provided code segment of the prediciton function can let us derive the following math:

$$\begin{aligned} y &= SR[h] + \hat{x}W[i,0] \\ \textit{prediction} &= y \geq 0 \end{aligned}$$

The timing variable t is used to point out the chronological behavior of the algorithm. The function updates $SR[j]$ $\forall j \in [1, h]$ each call.

$$SR[h-j+1] = SR[h-j] + \hat{x}(t)W[i(t), j]$$

This can be used to derive a general term for $SR[h]$ in order to trace how the prediction outcome depends on past iterations:

$$\begin{aligned}
\Rightarrow & \; SR[h] = SR[h-e] + \hat{x}(t)W[i(t), 1] \\
\Longleftrightarrow & \; SR[h] = SR[h-2] + \hat{x}(t-1)W[i(t-1), 2] + \hat{x}(t)W[i(t), 1] \\
\Longleftrightarrow & \; SR[h] = SR[h-3] + \hat{x}(t-2)W[i(t-2), 3] + \hat{x}(t-1)W[i(t-1), 2] + \hat{x}(t)W[i(t), 1] \\
\Longleftrightarrow & \; SR[h] = SR[h-4] + \hat{x}(t-3)W[i(t-3), 4] + \hat{x}(t-2)W[i(t-2), 3] + \dots \\
& \quad \vdots \;\; = \;\; \vdots \\
\Longleftrightarrow & \; SR[h] = \underbrace{SR[h-h]}_{SR[0]=0} + \hat{x}(t-h)W[i(t-h), h+1] + \sum_{j=0}^{h-1} \hat{x}(t-j)W[i(t-j), j+1] \\
\Longleftrightarrow & \; SR[h] = \sum_{j=0}^{h} \hat{x}(t-j)W[i(t-j), j+1]
\end{aligned}$$

This term shows us, that the cummulative outcome of a sequence does not depend on every single weight (first intuition) nor it depends on a single row of weights corresponging to a single branch address (second intuition). It depends on a sequence of past branch addresses which forms the so called path.

In theory there is a set of $(hashlen)^h$ detectable distinct paths. In practice, a perfect classification of this set is hard and highly depends on the self adjusting weights. Classification also suffers from aliasing of different paths, because of the hash function. This might result in a changing data set.

One might also argue, that a sum and a simple threshold classification can also decrease the number of detectable distinct paths, because of the sum's commutative property. Imagine two or more weigths without significant difference (significant in terms of a difference, which can change to overall outcome) in their values, being permuted without any effect on the branch decision. This reduces the set by all possible permutations regarding these weigths.

Learning The FPBP algorithm uses a very common learning rule, which can be derived by following the gradient descent approach. The gradient of the error function $E = t - y$ can be 1 or -1, depending on the corresponding input $\hat{x}(t-j)$.

$$\Rightarrow \frac{\partial E}{\partial W_{k,l}} = \frac{\partial}{\partial W_{k,l}} \sum_{j=0}^{h} \hat{x}(t-j)W[i(t-j), j+1] = \begin{cases} \hat{x}(t-i) & k = t-j \\ 0 \end{cases}$$

In a traditional perceptron learning algorithms, every single weight is updated. Here, the most recent output only depends on a single path of weights through the weight matrix. Apparently, the learning algorithm only updates weights beeing traversed earlier along this path.

The author proposed to use a threshold $\theta = 1.93h + 14$, such that W is updated in any case, when the magnitude of y is below this threshold. The value of θ is empirically determined to give the best result. This results in a higher average magnitudes of W and causes more updates of the weights. Therefore, the chance of pairwise almost equal weights should be reduced. Introducing the threshold indeed slightly improves the prediction accuracy. On the other hand, forcing the magnitude of W to high values will reduce the flexibility of the learning algoritm as it updates the weights only in small steps of $+/-1$.

3 Comparison

In the following the FPBP is compared to two other branch predictors, which come with Sniper's[2] default implementation. The One Bit Branch Predictor is a vectorof length 1024 of so called saturating counters associated with a specific hash value of branches. The Pentium M Branch Predictor is more complex and combines multiple branch prediction techniques. The advantage of a combination of different branch predictors became clear during the benchmarks. MPKI and misprediction rate do not strongly depend on the tested application.

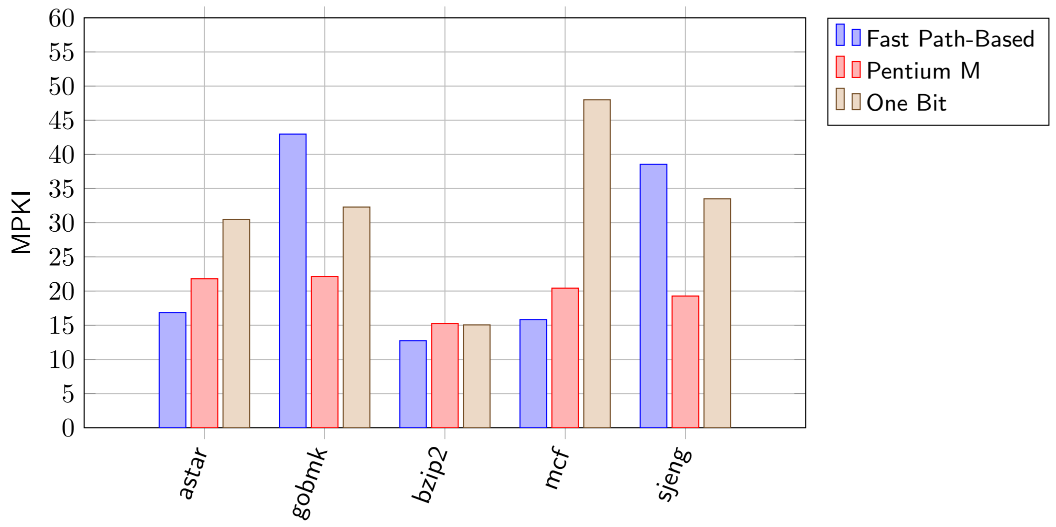

Figure 1: MKPI Plot: FPBP compared to Sniper's[2] build-in branch predictors

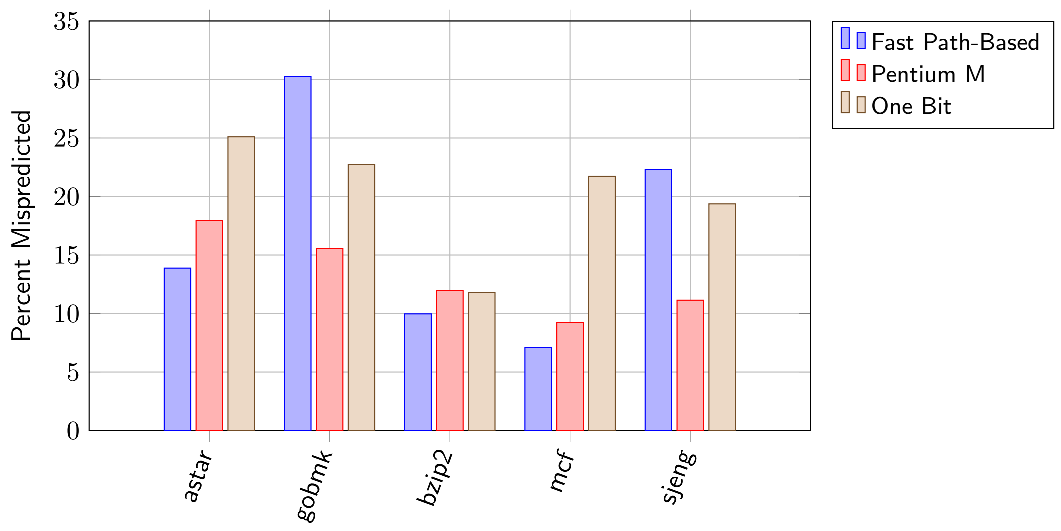

Figure 2: Mispredicted Percent: FPBP compared to Sniper's[2] build-in branch predictors

Whereas the One Bit branch predictor shows sufficient accuracy for the bzip2 benchmark, but its accuracy is much worse for the other benchmarks. This drop in accuracy is due to the simple structure of this predictor. It does not have the ability to detect more complex patterns. The FPBP shows a remarkable different behavior than the two other branch predictors. It performs with superior accuracy for astar,bzip2 and mcf, but appears to have the worst accuracy for gobmk and sjeng. The amount of simulated instructions varies by the fact, that the benchmarks have differant complexity and length. So, first idea was, that the FPBP just needs a minimum amount of instructions to warmup, such that weights are initialized properly. But the result of the astar benchmark refutes this. The benchmark only uses a small amount of approximately 3.9M instructions compared to the other benchmarks. (Table 1)

Benchmark	**Simulated Instructions**
astar	3.9M
gobmk	140.7M
bzip2	1B
mcf	1B
sjeng	234.4M

Table 1: simulated instructions for each benchmark

Branch Predictors in general ideally should have negligible warmup behavior. A neural predictor which requires a significant warmup to work properly cannot be flexible to future changes in the program flow. Observation of the predictor's weights have showed that they already have reasonable values soon after starting the simulation.

It appears to be true, that the FPBP also suffers from arbitrarily accuracy drops. The fact that the underlaying structure of this FPBP is similar to a tiny neural network, makes it difficult to understand its final decision-making process. During testing and adjusting the parameters, *hashlen*, the amount of entries related to branch addresses seemed to have a larger impact on accuracy than the history length h. This might lead to think, that branch addresses reveal slightly more information about an upcoming branch decision, than the branching history over the last h branches. Given a scenario, with one or more nested loop with large counters and several branches with pseudo random branching behavior inside of the loops, a branch address-oriented predictor definitely performs with better accuracy. Since today's programs indeed can consist of large loops, this scenario might be realistic and give a good explanation of the observed behavior. Fully optimizing the parameters remains a NP-hard multivariable problem. Therefore, the author uses this FPBP on top of a simple bimodal branch predictor. How the two branch predictors are combined is not revealed in the paper, but the combination of the two produces remarkable results. E.g. the mcf benchmark scores for a misprediction rate lower 2%. Branches of the bzip2 can be predicted with an accuracy of larger than 90%. The tradeoff of a combined branch predictor is still the hardware overhead of two units with almost the same in functionality and an additional control circuit to combine them.

4 Verification

The implementation of the FPBP does not perform as well as promised in the paper. The accuracy is significantly lower than the authors implementation showed in his benchmarks. This might lead to think that the implementation is simply wrong or consists of some tiny mistakes. In earlier stages, before debugging, my implementation indeed consisted of a few mistakes. For example, first the branch prediction history and weights were misaligned by one position. Errors like these result in a branch prediction accuracy not significantly better than a random guess. This, and the fact that the neural predictor indeed performs with very high accuracy for some parts during a running application, whereas it does not for other parts, should reduce the chance that the accuracy still suffers from bad mistakes to a negligible amount. Furthermore, the FPBP was able to beat the two default branch predictors One Bit and Pentium M in some benchmarks

References

[1] D.A. Jimenez. Fast path-based neural branch prediction. pages 243– 252, 01 2004.

[2] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, and Ibrahim Hurand Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.