High-Performance Neural Branch Predictor with Perceptrons

Junnutula Meghanath Reddy
Department of Computer Science and Engineering
Texas A&M Universtiy
College Station, Texas 77801
Email: meghz17@neo.tamu.edu

***Abstract*—This paper implements a method for branch prediction using the well-known successful concept of the neural networks. It combines the branch prediction with areas neural networks as they have very good capabilities to exploit for huge history length and make the accuracy higher than the current trending branch predictors. The perceptron predictor described in this paper performs better than the famous Gshare predictor on the given traces. It has been implemented carefully considering the given hardware constraints and budget for the CBP contest. Also, it undergoes training to update its status for future prediction which is the most important functionality of the perceptron.**

I. Introduction

Pipelining has been one of the most effective ways to improve performance on a single processor. On the other hand, branches impede machine performance due to pipeline stalls for unresolved branches. As pipelines get deeper or issuing bandwidth becomes greater, the negative effect of branches on performance increases. Hence, the need of a branch predictor is very crucial in the modern day processor to increase accuracy.

In computer architecture, a branch predictor is a digital circuit that tries to guess which way a branch will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high effective performance in many modern pipelined microprocessor architectures. Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline.

The branch predictor attempts to avoid this waste of time by trying to guess whether the conditional jump is most likely to be taken or not taken. The branch that is guessed to be the most likely is then fetched and speculatively executed. If it is later detected that the guess was wrong then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay. Branch instructions test a condition specified by the instruction. If the condition is true, the branch is taken: instruction execution begins at the target address specified by the instruction. If the condition is false, the branch is not taken, and instruction execution continues with the instruction sequentially following the branch instruction. As the pipeline deepens and the number of instructions increase, the penalty for a misprediction increases. The Fig.1 below shows how the basic branch predictors have been built. Also, the accuracy of static and dynamic branch predictors have been depicted in Fig.2 using gcc Spec95 benchmark instruction.

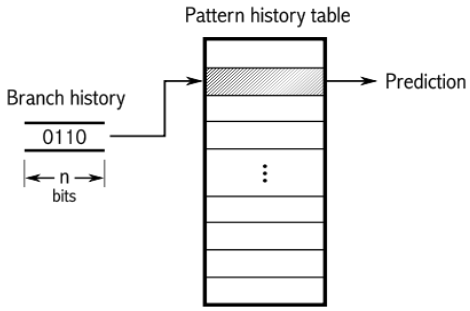

Fig. 1. Basic prediction algorithm

In computer science, neural networks are computational models that are capable of machine learning and pattern recognition. They are usually presented as systems of interconnected ”neurons” that can compute values from inputs by feeding information through the network. Like other machine learning methods, neural networks have been used to solve a wide variety of tasks that are hard to solve using ordinary rule-based programming, including computer vision and branch prediction. Given a specific task to solve, and a class of functions , the neural network observations and tasks to solve the task in some optimal sense. The perceptron predictor presented in this paper is also based on this basic idea but it uses the neural networks concept to exploit large history lengths for accuracy in prediction.

II. Related work and Background

Perceptron is a Neural Networks property that allows learning and identifying of stronger and weaker connections and correlations between branches. After a certain amount of learning steps, at uncertainties, or after mistakes it re-teaches itself. Basic perceptron usage assigns a vector of weights to each object in system where each weight represents the

strength of connection between another object and this objects' perceptron. The stronger the connection, the bigger the weight that represents it. Thus, this basic idea is implemented in this paper giving the predictor the capability to exploit the huge branch history for high accuracy.

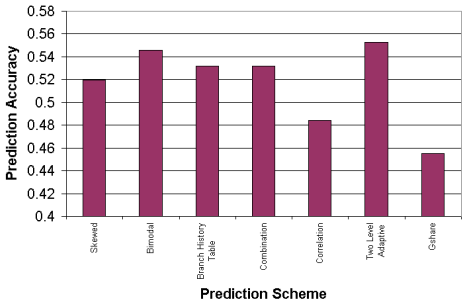

Fig. 2. Various prediction schemes

A. The Concept of Perceptrons

The neural predictor using perceptrons improves accuracy by combining path and pattern history to overcome limitation inherent to previous predictors. It uses a different prediction algorithm that would allow parallel execution of instructions during every prediction, thereby keeping the latency low.

The most important advantage of prediction of perceptrons in branch prediction is their memory consume, which is linear to considered history size. Perceptrons allow the incorporation of long history lengths when making prediction regarding whether a branch is going to happen or not.

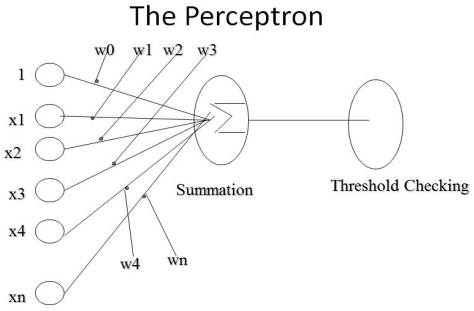

Fig. 3. The perceptron concept

III. Methodology

The Perceptron predictor uses perceptron learning to predict directions of conditional Branches. It is a correlating predictor that makes a prediction for the current branch based on the history pattern observed for the previous branches. The most important steps followed in implementing our algorithm are:

- Indexing
- Prediction
- Training
- Threshold Maintenance

A. Indexing

The index functions have a good effect on the branch prediction and considering their low cost of implementation they must be carefully designed for better accuracy in branch prediction. To predict a branch with an instruction memory address PC , there are 3 mentioned hash functions that can be used which have been mentioned in this paper. One of which is a direct and simple implementation which takes the modulus of the number of rows in the table for indexing.

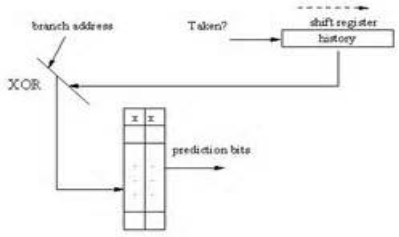

Fig. 4. Indexing using XOR

Secondly, we use a try to combine the instruction memory address and the history path for a better analysis when long histories are exploited by using the XOR operator. Finally, a prime number based hashing has also been implemented for better accuracy as they are known to be produce distributed results and less aliasing which is a common feature when trying to use perceptrons.

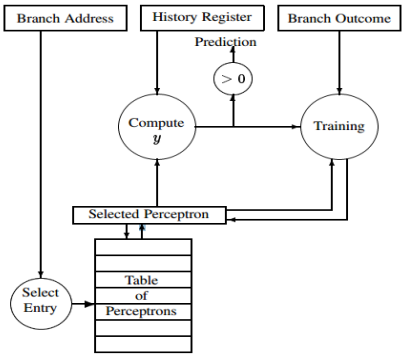

Fig. 5. Total concept of prediction using perceptron

B. Prediction

From the indexing stage we get a perceptron vector to carry out our prediction algorithm. This row is an artificial perceptron that is responsible for predicting that particular branch. A single perceptron could be responsible for the prediction of multiple branch instructions. The prediction is made based on the weight values in that row and on the outcome of the most recent "h" branches, which is stored in the global history register historyArray. A dot product is computed between the history register and the perceptron

weights vector and prediction is carried out according to the result.

$$y = W_o + \sum_{i=1}^{n} X_i * W_i$$

If the value of the dot product is atleast zero then the branch is predicted as not taken and otherwise if it is lesser than zero.

C. Training

Once the outcome is known, the training of the perceptron that was responsible for the prediction in a specific way depending on whether the prediction was correct and also based on the pattern history , that is stored in the global history register historyArray. This is accomplished by updating the weights in the corresponding row of the tab matrix. The bias bit in our table is changed according to the correct outcome of the branch. Now if there is a misprediction we have to update the values in two ways:

```
if sign(y_out) ≠ t or |y_out| ≤ θ then
        for i := 0 to n do
                w_i := w_i + t x_i
        end for
end if
```

Fig. 6. Various prediction schemes

1) If the branch is Taken then the historyArray having 1's in their indices have supported the prediction, hence their corresponding perceptron value should be incremented and the perceptron weight values with 0's should be decremented.
2) If the branch is Not Taken then the historyArray having 0's in their indices have supported the prediction, hence their corresponding perceptron value should be incremented and the values with 1's should be decremented.

D. Threshold Maintainence

The training algorithm is fed with an integer parameter theta that controls the tradeoff between long-term accuracy and the ability to adapt to phase behavior. It has been determined from various studies that choosing theta = [2.14*H + 20.58] gives the best accuracy, where H is the history length. Thus, for a given history length, theta would be a constant. The predictor would only be trained if there is a misprediction or if the result of dot product is be less then theta. Finally, the history register needs to be updated for each branch prediction for the further use of this prediction information on to the next and so on.

IV. Hardware Design Space

The design space has been carefully managed so as to not exceed the hardware budget of 16KB (i.e. 16,384 8-bit bytes) of state plus 256 extra bits. The C++ template library and STL have not been used to reduce the complexity and improve the accuracy. In order to achieve good reliable results three parameters have to be tuned in correctly.

- History length: Although the predictor can implement long histories we only consider values from 11-31 according to the hardware budget.
- Size of the weights table: The weights of the perceptrons can be both negative and positive hence they are signed intergers or short. This simplifies our design as analysis the integer weight components would be easier in debugging as well.
- Threshold: This parameter decides if our predictor needs training initially. It's value depends on the history length bits in the implementation and thus stays constant throughout the prediction.

V. Results

The perceptron branch predictor has been successfully implemented taking the concepts of the neural weights forming the vector of and the history lengths. The figure below shows relation between different configurations of the indexing methods. It also depicts the average MPKI for these methods carefully considering the hardware budget and constraints.

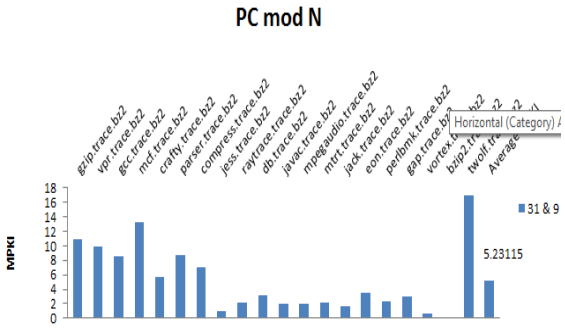

Fig. 7. Results of PC mod N with given Benchmarks

The predictor outperforms the traditional Gshare predictor and achieves good accuracy due to the exploitation of the long history lengths. The first case shows a average MPKI of 5.23 for maximum history lengths in the given hardware space. The precise result is due to the long history and the indexing the whole table although there might be conflict misses. For the given traces the method of indexing works well with the predictor.

Secondly, the XOR operator considers the history bits into account and combines the effect of the instruction address giving a good hashing function for indexing. This value is seen to be accurate for very long histories and less effective for the lesser hardware implementation.

Thirdly, we have the prime modulo function which gives an MPKI of 4.83 the best results when used as the hashing function. But the main disadvantage of the this method is that it does not take all the tables into account as the number of rows in the table need not be equal to the prime number used for

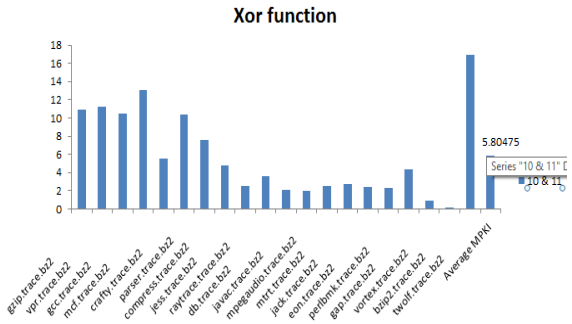

Fig. 8. Results of XOR function with given Benchmarks

hashing. Thus, it causes some misbehavior in the predictor. In our case, the supreme ability of prime numbers as a hashing function considering their limited divisibility dominates the negative case.

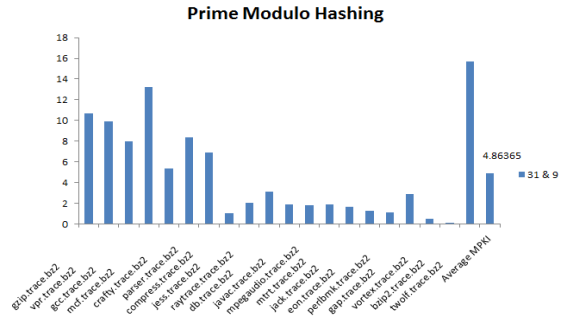

Fig. 9. Results of Prime Modulo with given Benchmarks

Finally, the Fig. 9. shows how different indexing methods stand against each other when evaluated with the same SPEC benchmarks and the given hardware constraints. As the given budget is small (i.e 16KB + 256 bits), the PC mod N functions exceptionally well and then the Prime Modulo. The XOR functions good with large history length which can be realized if the hardware budget is increased.

VI. Project Overview and Difficulties

The basic process of the project through different cycles has been explained thoroughly in this section.

A. Initial Research

The initial phase of the project took a lot of time gathering details about branch prediction and selecting the one which would be best implemented considering the programmers ability. The help and information provided during the lectures and class hours greatly helped in deciding which one could be implemented. After the selection of the perceptron predictor which performs better than the many of the available predictor till date study of the concepts was necessary.

Secondly, as the perceptron uses the concept of neural networks, its implementation, methods and other abilities of the neural networks were analyzed to see if any other neural network concepts could be incorporated into the predictor.

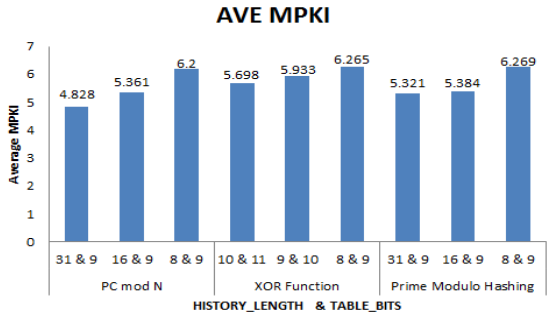

Fig. 10. Results for different Indexing Methods

B. Design Stage

In this stage all the hardware budget was carefully considered according to the projects requirements. The main advantage of the perceptron predictor was to utilize the long history length and predict the outcome. Since our predictor was constrained with the hardware complexity the history length, table size and the threshold were carefully designed to avoid anomalies at the implementation stage.

All the code has been written in C++ with the help of the header function math.h. The use of standard template library has been avoided as its realization on the hardware would be tough and sometimes impossible.

C. Coding

The implementation stage was carefully planned to avoid errors and stress at the debugging stage. Several C++ concepts were also revised through this stage for good code development and implementation. Also, the parameters and other segmentation faults were neatly dealt with according to the budget provided for the project.

D. Debugging

This was the toughest stage of the entire project cycle which involved dealing with segmentation faults, common logic errors and other compilation errors as well. After careful examination of the project code and eliminating all abnormalities the final working code produced considerable outputs and MPKI.

E. Graph Plots

After the debugging stage, the results of the perceptron predictor were plotted. The graphs presented above have been deployed using different methods of indexing and their response with the known SPEC benchmarks provided for testing.

VII. Conclusion

The paper implements the perceptron branch predictor using the concept from the neural networks. Perceptrons are used as because of their inherent capability to infuse huge history lengths into consideration. They perform prediction most accurately and utilize the basic weights function to deal with the anomalies and other defects of the earlier methods. The problem with perceptrons is that sometimes the computational complexity of the process increases thereby increasing the hardware complexity as well. Also, the perceptron is not able to accurately learn and distinguish the linearly inseparable functions, but despite this weakness the predictor performs well on the given traces and SPEC benchmarks.

Also, different indexing functions have been used in analyzing the benchmarks and their performance on using different hash functions. We see that the prime modulo function gives the best results but its ability to consider all the rows in the table is a potential flaw in the mechanism. The graphs have been plotted for the different result set that has been obtained from running the branch predictor on the given traces.

Acknowledgment

The author would like to thank Dr. Daniel. A. Jimenez for this wonderful opportunity and challenging task. It has been a great learning curve dwelling into the area of branch prediction. The information and support provided throughout the course of this project is most appreciated.

References

[1] D. A. Jimenez and C. Lin, “Dynamic branch prediction with perceptrons,” in Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7), January 2001, pp. 197–206.

[2] D. A. Jimenez, “Fast path-based neural branch prediction,” in Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36). IEEE Computer Society, December 2003, pp. 243–252.

[3] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive branch prediction,” in Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture, Nov. 1991, pp. 51–61.

[4] D. Jimenez and C. Lin. ”Neural methods for dynamic branch prediction,” in ACM Transactions on Computer Systems, November 2002

[5] D. T arjan and K. Skadron. Revisiting the perceptron predictor again. Technical Report CS-2004-28, University of Virginia, September 2004.

[6] D. A. Patterson and J. L. Hennessy, Computer Architecture: A Quantitative Approach, Morgan, Kaufmann Publishers Inc.,199.

[7] Hans Vandierendonck, ”XOR-Based Hash Functions”, IEEE Transactions on Computers, vol.54, no. 7, pp. 800-812, July 2005.

[8] Yi Ma, Huiyang Zhou, ”Using Indexing Functions to Reduce Conflict Aliasing in Branch Prediction Tables”, IEEE Transactions on Computers, vol.55, no. 8, pp. 1057-1061, August 2006.