Survey of Branch Prediciton With Machine Learning

Arshan Hashemi
EE Dept.
California Polytechnic University
Fremont, United States
arhashem@calpoly.edu

Neal Nguyen
EE Dept.
California Polytechnic University
Fremont, United States
nnguye57@calpoly.edu

***Abstract*—**

I. Introduction

This paper surveys different branch prediction methods, analyzing different heuristics with different instruction sets. This paper covers traditional branch prediction methods such as static predictors, 1-bit saturating predictors, 2-bit saturating predictors, 2-level adaptive predictors (history length = 1, 2, 3, 4), globally shared predictors, and tournament predictors. It then ends two new branch prediction methods with a static feedforward neural net utilizing different history bits, and a dynamic machine learning model.

This analysis shows the trade-offs to different methods and reveals if machine learning is a viable option to branch prediction. This research is valuable because branch prediction is a critical bottleneck in many computer architectures. Conditional branches still comprise of about 20% of instructions [Eggers(2006)]. Before branch predictions, valuable time was lost while waiting for the branch to evaluate. However, now, it is crucial to reduce mispredictions as much as possible as well. So this paper places a significant focus on examining the miss rate (amount of times a branch predictor is incorrect), as well as the overhead, is added in the Python simulations.

Our methods confirm that Neural Networks out-perform conventional methods of branch prediction without significant performance or memory impact.

This paper is structured as follows. In section III we discuss prior work regarding the use of Neural Networks for branch prediction. Section IV outlines our methodology, and, finally, section V presents the results of our work as well as a discussion of the significance of our findings.

II. Background

This section covers the current state of branch prediction within computer architecture. The section begins with the first stages of branch predictions and will move through different methods for each subsection.

A. Static Branch Prediction

Static branch prediction is the simplest of the branch prediction methods. This style does change as the program runs. It always guesses taken or not taken (one or the other). 66% of branches, branch forward & are evenly split between taken & not taken. The other 34% of branches, branch backward & are always taken [Eggers(2006)]. It is statistically expected for this predictor to guess 67% of branches correctly if it always guesses not taken. These results are used less so as valid results, but more as a baseline for the absolute worst a branch predictor should perform.

B. 1-Bit Saturating Counter

1-bit saturating counters provide the bare minimum in history remembrance. It records the last outcome of the branch at that address and predicts the same outcome the next time the same address comes around. This simple predictor is not very accurate.

C. 2-Bit Saturating Counter

2-bit saturating counters utilizes a state machine for its predictions as shown in figure 1. There are four states: strongly not taken, weakly not taken, weakly taken, strongly taken. When a branch is evaluated, it predicts taken if the state machine is in weakly/strongly taken states and not taken if the state machine is in weakly/strongly not taken states. Then, when the branch is evaluated, the state machine updates as according to figure 1.

Requiring 2 of the same branch decisions before moving from not taken to taken or vice-versa. This is useful when a loop closing conditional jump is mispredicted once rather than twice. This method performs well for its simplicity, resulting in 93.5% correct predictions on SPEC'89 benchmarks [McFarling(1993)]

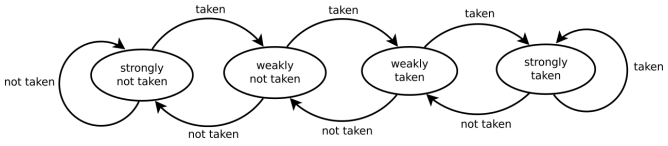

Fig. 1. State machine for 2 bit saturating counter [wiki(2018)]

D. 2 Level Adaptive Predictor

This predictor keeps track of history, adding an extra layer to the prediction algorithm. For each branch, n bits are used

to store the past n branch directions (taken or not taken, 1 or 0 respectively), these branch history bits index into another table which will then have a 2-bit saturation counter. How this differs from a 2-bit saturating counter is that instead of a counter for each branch address, now, for each branch address and the possible combinations of history, there is a 2 bit saturating counter as shown in figure 2.

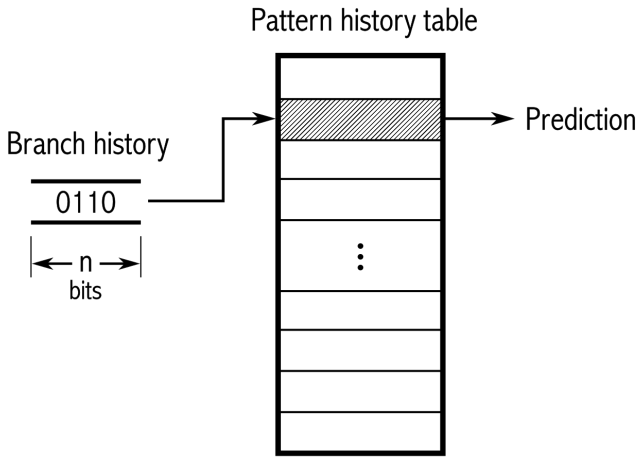

Fig. 2. 2 Level adaptive predictor model.

E. Local Branch Prediction

Local branch predictors have a separate history buffer for each conditional jump instruction. This prediction method applies to any predictor that uses history. It does not necessarily have to separate the pattern history table as well.

This method is 97.1% correct on SPEC89 benchmarks [McFarling(1993)]

F. Local Branch Prediction

Global branch predictors share a history buffer for all conditional jumps. This method is good when if statements are correlated. However, the history buffer can become diluted if the conditional statements are not correlated. This method is only better than the saturating counter in large programs and requires a large history buffer for it to work well.

G. Tournament Predictor

Tournament predictors are a combination of local and global branch predictors. It chooses the best one to use by performing both and keeping track of which one is right more times.

H. Multilayer Perceptron Neural Networks

A common class of artificial neural networks is the *Multilayer Perceptron (MLP)*. An MLP is composed of interconnected nodes, or *neurons*, contained in an *input layer*, one or more *hidden layers*, and an *output layer*. In a feedforward MLP, each neuron in a layer receives a set of inputs from the previous layer, calculates the sum, and outputs a value referred to as the *activation* to each of the neurons in the next layer. If enough inputs to a particular neuron activate, a threshold is surpassed and the neuron, itself, activates.

This activation threshold is modeled by the *bias*, a number that the sum of the inputs has to surpass in order to activate the neuron. However, to model complex decision making, the inputs to a neuron need to be able to take on different weights. Whether or not a neuron activates can now be defined as a function of z:

$$z = w \cdot x + b$$

A sum of the weighted input, w_x, to a neuron added to a negative bias, b, for that neuron. If z is greater than zero, the threshold is exceeded and the neuron activates outputting a number close to one, but if z is less than zero, the neuron does not activate and outputs a number close to zero. This is mathematically modeled by an activation function such as the sigmoid function

I. Perceptron

A Perceptron is the simplest possible neural network consisting of only a single neuron which similarly connects several input units by weighted edges to one output unit. Again, the output of the perceptron can be modeled as a Boolean activation which is a function of the weighted sum of the inputs added to a bias. In the simplest case, a positive sum will result in a positive prediction while negative sum will result a negative prediction.

III. Related Works

Due to the nature of branch prediction it is a problem that is well suited for neural networks, and a selection of prior work attempts to accomplish this task.

We encountered various strategies to accomplish this task. Static Methods are proposed to perform branch prediction through on evidence-based methods. In this methodology known programs are profiled to predicted branches in unseen programs. This relies on standard supervised machine learning methods (models such as Neural Networks and Decision Trees) and does not use any type of heuristics. These methods can be extremely effective with repetitive and predictable workloads, but the models require retraining every time the workload changes. For this reason static methods are considered irrelevant on modern CPUs.

Dynamic Methods do not need to be trained ahead of time and can improve their performance as they make predictions. The most common dynamic predictor is the Perceptron which represents a simple neural network. Because they are not profiled to a particular workload, dynamic methods are well suited for general purpose processors where the workload is unknown ahead of time. The perceptron acts as a linear predictor and makes dynamic predictions through an approach that is more similar to traditional methods such as saturating counters and a branch history table.

In the paper, "Dynamic Branch Prediction with Perceptrons", Jimenez et. al. present a perceptron based approach to branch prediction that can be implemented in hardware

[Jimenez and Lin(2001)]. Their work builds on the observation that counters within tables can be replaced with neural networks to improve accuracy.

In their proposed methodology each static branch is allocated its own perceptron. Similar to counter-based methods, a table stores the perceptrons and is indexed global history shift registers and branch address.

In the paper they demonstrate that their predictor performs well for the class of *linearly separable branches*. The perceptrons are learn by adjusting their weights and bias according to whether the branch was taken or not taken. Each weight corresponds to a particular history bit and adjusts as the linear correlation associated with that particular bit of history. So for a negative correlation the weights tend towards the a negative value, and for a positive correlation they tend towards a positive value.

This procedure is outlined in the steps below:

1) The branch address is hashed to produce an index into the table of perceptrons.
2) The i^{th} perceptron is fetched from the table and placed in to a vector register.
3) The value of y is computed as the dot product of P and the global history register.
4) The branch is predicted not taken when y is negative, or taken otherwise.
5) Once the actual outcome of the branch becomes known, the training algorithm uses this outcome and the value of y to update the weights in P.
6) P is written back to the i^{th} entry in the table

The proposed predictor performed with a error rate of 6.89% which was an improvement over traditional methods like *gshare* and bimode which had error rates of 10.1% and 8.2% respectively.

IV. Methods

Here we highlight what we did in this project.

We compared our machine learning algorithm to a branch prediction simulator created by BLin409 [BLin409(2013)].

Our methodology consisted of 4 parts:

1) Selecting a dataset
2) Selecting a simulator
3) MLPs for Static Branch Prediction
4) Perceptrons for Dynamic Branch Prediction

A. Selecting a Dataset

To select the appropriate dataset we searched for a trace files that were taken from realistic workloads or benchmarks and were large enough to train our models.

We decided to use the CAR Branch Simulation Dataset [nihakue(2014)] which provided traces for two common benchmarks, *gcc* and *mcf*. The makeup of the trace files is shown in table I:

	Size	Taken	Not Taken	Unique Addresses
gcc	592,262	23.61%	76.39%	286
mcf	339,493	32.24%	67.76%	33

TABLE I
BENCHMARK CHARACTERISTICS

B. Selecting a simulator

In order to generate a baseline for our model performance it was necessary to run a branch simulator that evaluated more conventional methods of branch prediction. For this we relied on two simulators that we found online:

- CAR Branch Sim [nihakue(2014)]
- Branch Instruction Simulator [BLin409(2013)]

The CAR Branch Sim was written in Python and provided implementations of Static Prediction (take/no take), branch profile, 2-bit saturating history, 2 lvl adaptive(depth 1-4), and tournament predictors.

The Branch Instruction Simulator was written in C++ and provided implementations of 1 bit Bi-modal, 2 bit Bi-modal, *gshare*, and tournament predictors.

C. MLP for Static Branch Prediction

First we implemented MLP-based Static Branch Prediction. We trained a single feed-forward Neural Network with 12 neurons in the input layer, 6 neurons in a single hidden layer, and a single output neuron corresponding to taken/not taken. The input vector corresponded to a standardized branch address (the memory address scaled down to have unit mean and variance) and four bits of history.

The advantage of this approach is that a single network can make prediction for every address while the dynamic predictor must keep a table of perceptrons corresponding to the each individual branch address.

D. Perceptrons for Dynamic Branch Prediction

Next, building off of GitHub code that we found, we implemented the algorithm for Perceptron-based Dynamic Branch Prediction proposed by Jimenez et. al [Jimenez and Lin(2001)]. As described in Section III the algorithm iteratively updates weights and biases based on the whether the branch prediction was correct.

We created a Perceptron class with methods to make a prediction and a update its weights accordingly. The table of Perceptrons is represented by a dictionary. However, to simulate a table of fixed size we manually hashed the branch addresses and modulo-ed by the table size. In this manner, we could simulate collisions in the Perceptron table.

Next, we varied the number of history bits and table size to see the effect on both the prediction time and accuracy. Since each branch address would hash to a particular perceptron, the optimal scenario would allocate one perceptron for each branch address. However, in reality memory is much more constrained and table size is limited. Therefore collisions, where two branch addresses hash to the same perceptron, must be accounted for. We varied the table size from 1 (a

single perceptron for all addresses) to N where N was the total number of unique branch addresses.

We expected to se

V. Results and Discussion

This section covers the performance results of various predictors tested in this research. Subsection V-A begins with the saturation predictors and their performance over time, subsection V-B then covers the static machine learning model trained in this research, subsection V-C covers the adaptive machine learning model, and finally, subsection V-D compares the best performance between each predictor.

Not every predictor presented in the overall comparison has an in-depth subsection due to time constraints or the simplicity of the program. For example, with static (taken or not taken) branch predictors, there is no table size to vary.

A. Saturation Predictor

This subsection covers the performance of the saturation predicts for 1 and 2 bits for GCC and MCF. Figures 3 and 4 shows that the 1 bit predictor is always worse than the 2 bit predictor. This is expected as the 2-bit saturation predictor has the extra bit to protect against loops that are only not taken a few times. The 1-bit predictor is significantly worse than the 2-bit predictor in GCC as opposed to MCF where it is very close. This is possible because GCC has 200,000 more branches than MCF.

Regardless of whether the saturation predictor is 1 bit or 2 bits, they follow the same graph within GCC or MCF. Figures 3 and 4 show that the miss rate stops decreasing as quickly at around a table size of 256 for GCC and around 128 for MCF. GCC contains 286 unique addresses for the branch commands while MCF contains 33 unique addresses. So an increase of table size would benefit GCC much more to prevent conflicts between 286 addresses as compared to 33 unique addresses.

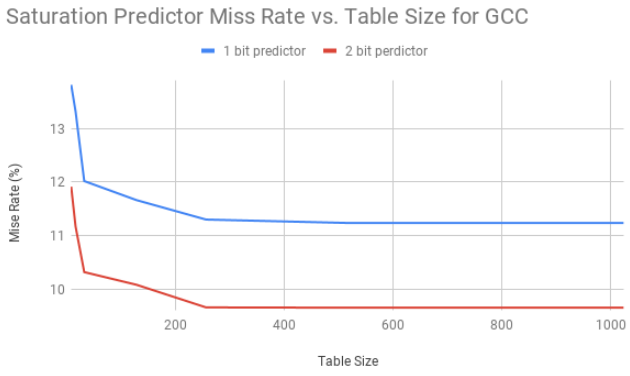

Fig. 3. Miss rate for saturation predictor as table size increases.

B. Static Machine Learning Model

This subsection covers the static machine learning model we created and the results for its accuracy and training. Figures 5 and 6 show that increasing the history actually reduced the

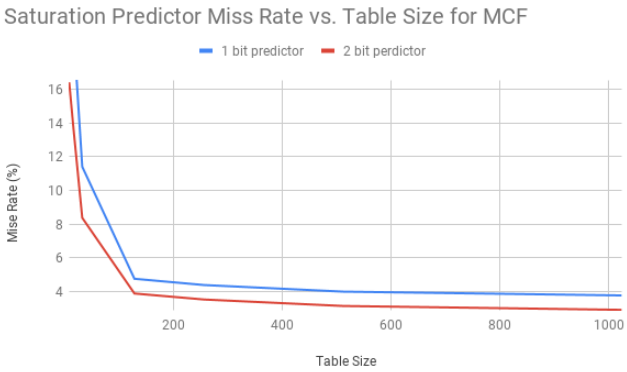

Fig. 4. Miss rate for saturation predictor as table size increases.

training time and miss rate. It took 5 hours and 52 minutes to train this model three times on GCC and 3 hours and 43 minutes to train this model three times on MCF. This limited the number of data points acquired for these figures.

The reduction in training time significantly decreased as the history size increased when training under GCC as compared to MCF. We are unsure of why this is. Increasing the history size would require more parsing for the history and result in more nodes, requiring more weights to be backpropagated in the neural net.

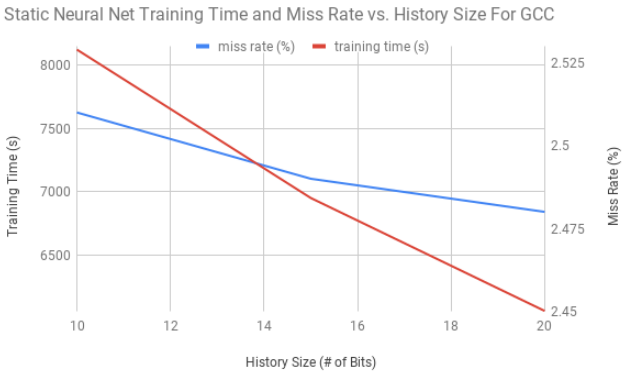

Fig. 5. Effect on miss rate and training time as global history size increases (lower is better) for Static Branch Predictor in GCC program

However, regardless of performance, the training time is far too long to be practical. But, with a known program that consistently branches the same way the static model could be pre-trained and hardcoded in. At which point, the timing, and accuracy during the validation step would be important. But, due to the significant training time, we were unable to obtain those numbers

C. Adaptive Machine Learning

This section covers the results for an adaptive machine learning model. This model was found on GitHub and updated to run at various table sizes and global history sizes for GCC and MCF.

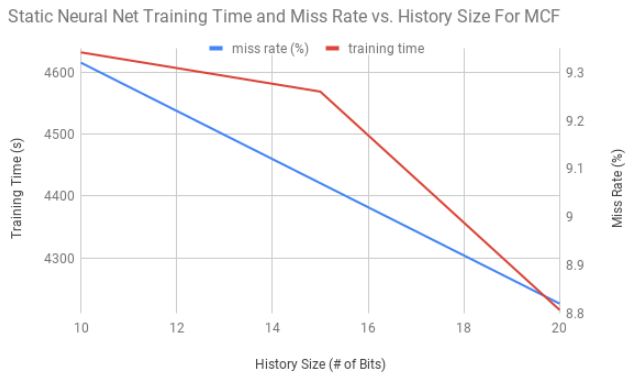

Fig. 6. Effect on miss rate and time as global history size increases (lower is better) for Static Branch Predictor in GCC program

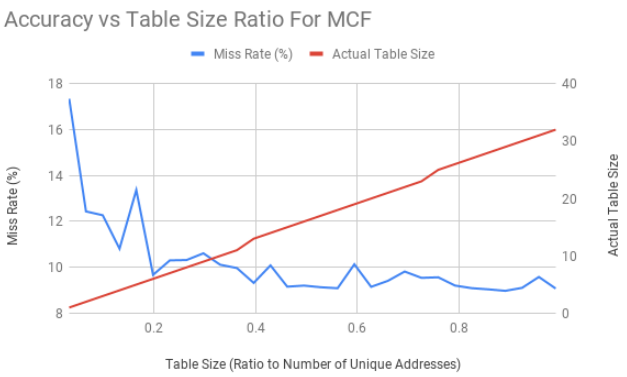

Fig. 8. Effect on miss rate as table size increases (Lower is better) for Adaptive Branch Predictor in MCF program

Figures 7 and 8 show the miss rate and table size to run as the table size ratio increases. The table size is the lookup table for storing each predictor. The smaller the table, the more each branch will share predictors. The table size ratio is the ratio between the table size and the number of unique addresses the program uses. This is used as the x-axis to keep a consistent increase in table size as GCC contains 286 unique addresses while MCF only contains 33.

From figures 7 and 8, a table size that is 20% of the unique addresses in a program is the best number. If GCC contains 286 unique addresses and MCF only contains 33 unique addresses, then the optimal table size would be 57 and 7 respectively before the miss rate drop plateaus.

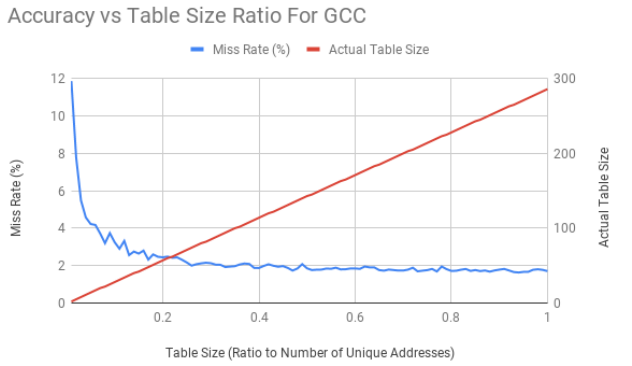

Fig. 7. Effect on miss rate as table size increases (Lower is better) for Adaptive Branch Predictor in GCC program

Figures 9 and 4 shows the change in time taken and miss rate for each graph as the global history size changes. This is the number of bits to store the past branch outcomes. Increasing the history provide more context and reduces the miss rate, but the time to run the predictor will increase.

The miss rate will logarithmically decay and eventually plateau, but the time linearly increases with the global history size. To optimize the global history size, we maximize the logarithmic decay before it plateaus to minimize the linear time increase. Figures 9 and 4 show that the optimal global history size is 17 for GCC and 8 for MCF.

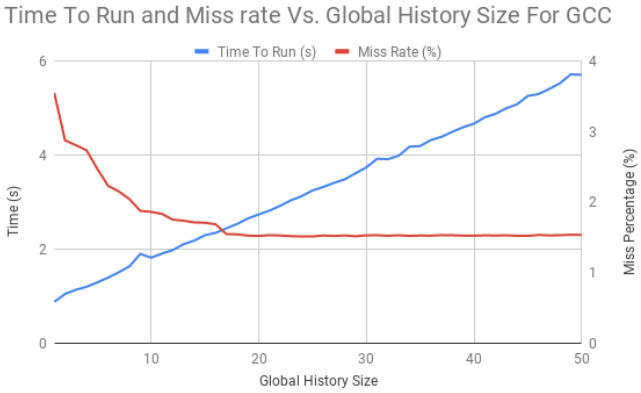

Fig. 9. Effect on miss rate and time as global history size increases (lower is better) for branch predictions in GCC program

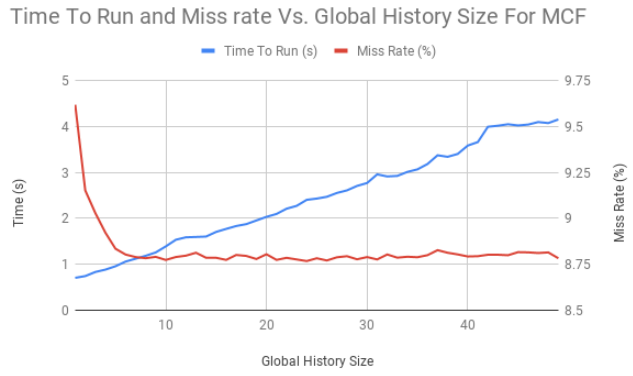

Fig. 10. Effect on miss rate and time as global history size increases (lower is better) for branch predictions in MCF program

This analysis shows the optimal operation of the adaptive machine learning model, assisting in the understanding of this branch prediction model. This optimal adaptive machine

learning model is used to compare to other branch predictors in section V-D.

D. Overall Comparison

This subsection compares all the branch prediction models discussed in this paper. Figures 11 and 12 focus on the miss rate. Figure 11 shows all methods including the static taken and not taken models. These models were only placed as a reference point. As expected, these two methods perform significantly worse than the other branch prediction methods. Figure 12 shows the more significant prediction models, showing that the tournament predictor performs best in both cases with the lowest miss rate. We speculate that this is because it switches between two different models and adapts to the branch case by case.

This comparison shows that the adaptive machine learning mode performs best. The tournament model performed second best in GCC and third best in MCF. The static Machine Learning model performed second best in MCF. Training the static model took hours to complete and running the adaptive model took around 4 seconds. The tournament model was the best traditional performer. The Adaptive, machine learning model is the best new predictor we tested. This implies that machine learning may be the solution to branch prediction. If implemented in specific hardware, it could even compare in terms of time.

These formal timing results show that the machine learning models we created are worse in time and but better at correcly predicting branches. It was difficult to fully compare all methods as some models were implemented in C and some were implemented in Python, which is much slower. But, these numbers can provide a high level, informal understanding of the timing for the machine learning model methods specifically.

VI. Conclusion

In this work we have shown that Neural Network based branch prediction is a viable alternative to more conventional predictors. By conducting a survey of the several of the best performing predictors we established a baseline to which our models can be compared. We constructed a static branch predictor based on an MLP Neural Network and a dynamic branch predictor (adaptive predictor) based on a table of perceptrons. Our results show that the adaptive predictor out performed any other individual method of branch prediction. A tournament predictor which utilizes the adaptive predictor as well as the other predictors could therefore provide the best possible performance.

The viability of perceptron-based predictors opens the door to more complex networks which can make predictions with increased accuracy. The benefit of perceptron-based predictors is that it can be implemented in hardware and trained continuously. Further, it is not constrained in the number of history and address bits it can use like more conventional methods.

Our static predictor also provides better accuracy than conventional methods. Although it must be trained beforehand, because the network size is independent of the number of unique branch addresses it might be a good alternative to consider for known workloads that will be run repetitively. This is especially true when the code size and number of branch addresses is very large.

VII. Future Work

References

[Eggers(2006)] S. Eggers, "Branch Prediction," 2006.

[McFarling(1993)] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[wiki(2018)] wiki, "Branch Predictor," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Branch_predictorOne-level_branch_prediction

[Jimenez and Lin(2001)] D. A. Jimenez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206.

[BLin409(2013)] BLin409, "Branch prediction simulator," https://github.com/BLin409/Branch-Prediction-Simulator, 2013.

[nihakue(2014)] nihakue, "branch$_sim$," , 2014.

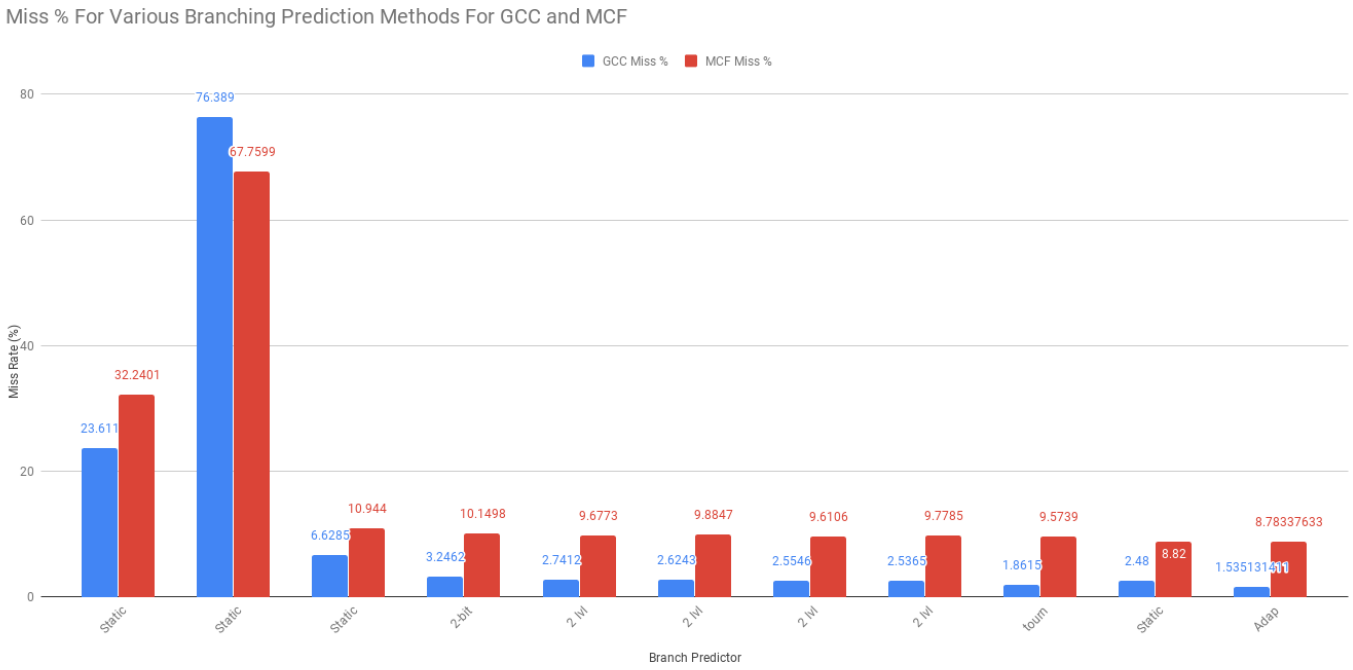

Fig. 11. Comparison of all branch predictors tested.

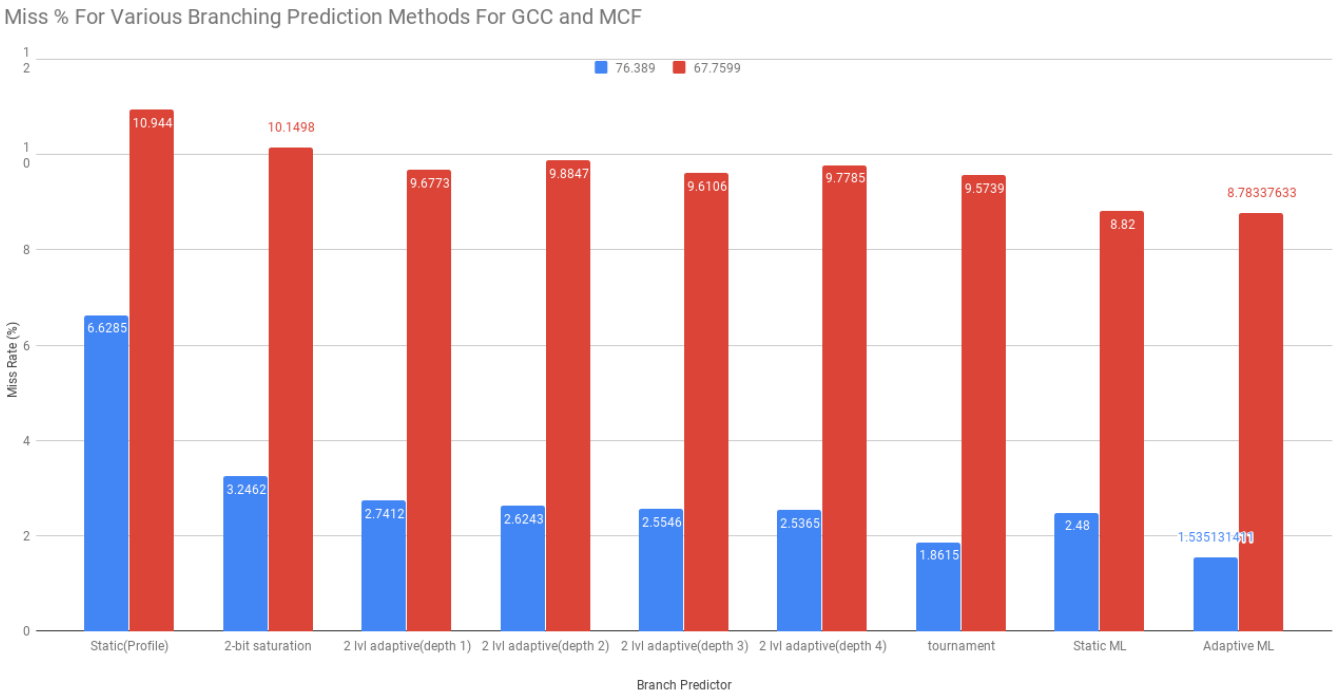

Fig. 12. Comparison of all branch predictors tested without static results.