CDA 5106 - Spring 2023

Machine Problem 2: Branch Prediction

# BPTP: A Machine Learning-based Branch Prediction Technique for Performance Optimization

by

Group 7

*I have neither given nor received unauthorized aid on this test or assignment.*

*Signed: Richard Morand, Ethan Bliss, Kate Fort, Zoe Batz, Lior Arin Barak, Tommy McCane, Wessley Dennis, and Cesar Hernandez*

University of Central Florida
Department of Computer Science

# BPTP: A Machine Learning-based Branch Prediction Technique for Performance Optimization

Ethan Bliss
*BS, Computer Science*
*University of Central Florida*
Orlando, Florida
ebliss@knights.ucf.edu

Lior Arin Barak
*BS, Aerospace Engineering*
*University of Central Florida*
Orlando, Florida
liobara3@gmail.com

Kate Fort
*BS, Computer Science*
*University of Central Florida*
Orlando, Florida
katefort@knights.ucf.edu

Wessley Dennis
*MS, Computer Science*
*University of Central Florida*
Orlando, Florida
wedennis0@knights.ucf.edu

Thomas McCane
*MS, Cybersecurity*
*University of Central Florida*
Orlando, Florida
tmccane@knights.ucf.edu

Cesar Hernandez
*MS, Computer Science*
*University of Central Florida*
Orlando, Florida
cesarhernandez@knights.ucf.edu

Zoe Batz
*MS, Computer Engineering*
*University of Central Florida*
Orlando, Florida
zoeb99@knights.ucf.edu

Richard Morand
*MS, Computer Science*
*University of Central Florida*
Orlando, Florida
rmorand@knights.ucf.edu

***Abstract*— Modern computer processors generally use static or dynamic branch prediction to accurately and efficiently predict branch outcomes. Here, we present a new method for branch prediction. This method allows machine learning to choose the optimal predictor for a specific program or thread for the next limited number of branches in order to yield the most accurate results. Our approach aims to determine the best standard branch predictor among the Gshare, Bimodal, and Smith Predictors for a segment of a program execution to dynamically maximize prediction accuracy across the entire program execution. Our results show that we can obtain a similar or greater level of performance than a traditional predictor when employing a feature-trained machine learning predictor.**

***Index terms*—Branch Prediction, Machine Learning, Optimal prediction technique, C++ Code**

## I. Introduction

Branch prediction is a form of speculative execution that is used to predict conditional operations; it wants to predict the direction a branch takes within the execution of a program. While speculative execution has been around in various forms since the 90s, very few breakthroughs in its overall operation and implementation have occurred that would lead to a significant increase in performance. Thus, we aim to explore a more cutting edge approach to speculative execution - machine learning to augment the performance of traditional branch predictor techniques.

At the core of the research is the Branch Predictor Technique Predictor (BPTP). This machine-learning software will analyze a set of instructions and determine, based on past results which branch predictor technique is likeliest to have the most success. Previous research on deep learning branch prediction has been highly successful, so it may be possible to abstract this machine-learning ability to the predictors themselves. As dynamic machine learning prediction is not performance-friendly, this static ability could be a large gain in performance without a significant loss of accuracy.

For this project, we seek to create a novel branch prediction technique by using machine learning to choose an optimal prediction technique for a certain series of branches. We will first analyze the program branching features dynamically, and then a simple predictor will be dynamically chosen to predict the branches to be taken. This step-back from directly choosing the direction of an individual branch makes the idea more applicable since the model does not have to make predictions for the individual branches. We believe hardware complexity may decrease since the machine-learning training portion will only be done for a chosen period of time, needing only the trained model and simpler prediction hardware moving forward. We hypothesize that performance will be increased for these programs because a targeted branch predictor should have higher success rates than generalized predictors. Security may be improved, as those looking to exploit the cache or timing may be unaware of which specific predictor is being used.

## II. Background

### A. Branch Prediction

Branch prediction is a concept in computer architecture created in the early 1990s to optimize the performance of processors. This technique predicts the outcome of conditional branch instructions, which may entirely alter the 'normal flow' of the instructions. In some cases, branching may cause a delay in instruction fetching and execution. Those possibilities of delays result in performance penalties, meaning the processor has not done meaningful work it could have been doing.

To combat these delays and increase performance, branch prediction techniques work by predicting the expected outcome of a conditional branch before it is called. If predicted correctly, the processor can continue to execute correct instructions. Because of this, we would not have to wait for the branch instruction to be resolved if we predicted that it has been taken or not.

### B. Machine Learning in Branch Prediction

Perceptrons were first implemented in 1958 by the US Naval Laboratory for image recognition, and were based off of a human biological neuron [1]. It was discovered that perceptrons are great for linearly-seperable applications, meaning a line can be drawn through a graph with all of the points in one category falling on one side of the line.

Naturally, the binary nature of branch prediction would lead one to wonder whether the binary output of perceptrons may work for enhancing prediction accuracy. As early as 1995, applications of neural networks to branch prediction were being published, such as in the paper "Corpus-Based Static Branch Prediction." [2].

More recently, in the early 2000′s and still today, the increased accessibility of machine learning has led to a number of applications of perceptrons, deep learning, and convolutional neural networks (CNN) to branch prediction. These have included both static and dynamic methods, but tend to lean toward static because of the difficulty of using complex predictive models at runtime.

Other predictors were developed, such as Global Share Branch Predictor, and Tournament-Style Predictor, which has been popularly used in common processors since 2006. [3]

### C. Use of Machine Learning

Since branch predictors have to decide whether a branch has been taken or not, we sought to use a classifier algorithm to check on the performance of each traditional branch predictor that was already implemented. Our model gives a certain accuracy rating for each of these traditional predictors. As we discuss the g-share, bimodal, and the smith algorithms, we will see a comparison on how each performs on given traces.

Our choice of classifier is XGBoost. The XGBoost classifier framework is an gradient boosting framework optimized for efficiency with a high accuracy rate. Its built-in support for parallel processing expedites the training process. In the implementation, we will use the XGBoost classifier to predict the branch outcomes. We will extract relevant features such as past branch instructions, instruction type, and address of the branch instruction. To begin, each program execution trace is split into training and testing sets. Then, the XGBoost classification model will be trained on the training set and compared to the testing set for evaluation and tweaking.

Using machine learning for branch prediction is a well known challenge, because the cost of querying a machine learning model is significantly higher than the cost of a mispredicted branch. This is observed in existing ML solutions such as the perceptron, which predicts what will every branch outcome be. A natural improvement to execution time is to instead predict what the branch outcome will be for the next several branches. This allows us to query the model more sparsely. However, this creates a difficult problem to solve, as returning a set of outputs is much more difficult for a model to correctly predict than simply one value. Solving this problem will require the perceptron to be trained for significantly longer.

Our approach unveils a way to mitigate this challenge of returning multiple future predictions. Because we predict a predictor to use for the next several branches, the state space is reduced back to 1 output, and we can afford to make that improvement without a substantial loss of accuracy. Thus, our method can provide similar results while being much more performant than previous machine learning methods.

## III. Design

Our proposed solution to improving branch prediction will be a dynamic approach which will leverage feature training to determine the best branch predictor for a particular part of program execution. A lot of the current branch predictors employed today are already optimized for certain types of program branch execution. The goal here is to utilize a library of branch predictors and use a machine learning algorithm to 'curve fit' them to parts of the code execution where they are most optimal. We will focus on versatility, performance and simplicity in our design. This section will go into the Philosophy, the Proposed Implementation, Assumptions and additional concepts that can serve to augment the approach.

### A. *Philosophy*

This predictor will employ feature training to determine which of the basic branch predictors to employ for a section of the program execution. For the first implementation of this design, we will be choosing from either the Gshare, Bimodal, or Smith branch predictors. These are not only some of the most widely used predictors, but also help give us a benchmark from which to evaluate our implementation.

This branch predictor will operate in two phases. The first is the training phase whereby the predictor is 'taught' how to associate different features with different predictors. The second phase would be actual implementation on a program execution that it has not been run on before. This would require this predictor to be trained before distribution, the fidelity of the training determining the accuracy. i.e. we can expect better accuracy for the predictor trained on a specific class of programs, or even varied runs of a single program.

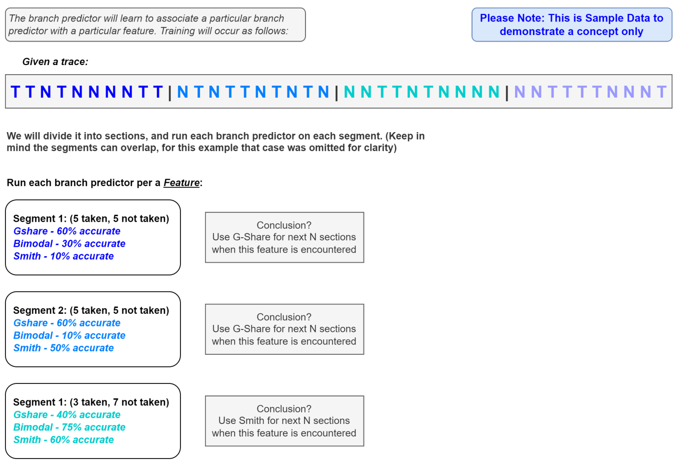
The branch predictor will learn to associate a particular branch predictor with a particular feature. Training will occur as follows:
Please Note: This is Sample Data to demonstrate a concept only
Given a trace:
T T N T N N N N T T | N T N T T N T N T N | N N T T N T N N N N | N N T T T T N N N T
We will divide it into sections, and run each branch predictor on each segment. (Keep in mind the segments can overlap, for this example that case was omitted for clarity)
Run each branch predictor per a Feature:
Segment 1: (5 taken, 5 not taken)
Gshare - 60% accurate
Bimodal - 30% accurate
Smith - 10% accurate
Conclusion? Use G-Share for next N sections when this feature is encountered
Segment 2: (5 taken, 5 not taken)
Gshare - 60% accurate
Bimodal - 10% accurate
Smith - 50% accurate
Conclusion? Use G-Share for next N sections when this feature is encountered
Segment 1: (3 taken, 7 not taken)
Gshare - 40% accurate
Bimodal - 75% accurate
Smith - 60% accurate
Conclusion? Use Smith for next N sections when this feature is encountered

Figure 1: *Feature Training* The graphic shows the relationship between the features and the predictors, and how a predictor is determined for a given feature set.

### B. *Proposed Implementation*

This predictor will utilize machine learning for the training phase of it's implementation. The user will specify a Section Length which will determine the granularity of the features. The Section Length will work similar to a sliding window allowing the predictor to gather local details about the program execution. These details are the features. For purposes of this predictor a feature can refer to either a set of two numbers, detailing the amount of branches taken and not taken whereby the sum would equal the Section Length, the volitility - how often the branch switches, and or other aspects ascertained from the obtained section of the global shared history. For purposes of this design section, we will focus on the feature as represented by taken and not taken items.

When describing this predictor, there are two key phases of its operation. Each phase of operation has a required input and a required output. It's important to know that information from the training phase of this predictor will be used in the operation of the runtime phase.

1) *Requisite Inputs:* The training inputs for this predictor will be a trace file or series of trace files from the execution of a number of different programs, or multiple runs of a single program. For this project, we will utilize the trace files supplied, as these will serve as a baseline from which we can evaluate performance later. In addition, we will use the Intel PIN Tool to collect additional traces for training and testing our Branch Predictor Technique Predictor. With the PIN Tool, we are able to collect more information about the branches, such as if the branch was conditional.

The predictor will have a number of standard branch predictors to choose from, in this case Gshare, Bimodal, and Smith. The predictor along with the trace files will take in a set of values detailing the Section Length and multiple other default parameters for the standard predictors from which it will choose. One additional variable that may need to be specified is the number $n$ specifying the amount of segments to utilize the branch predictor selected. In other words, when we update the chosen predictor after taking in the features, use that branch predictor for the next $n$ segments before querying the model again.

2) *Requisite Outputs:* The output would be calculated during the training session. The idea is to successfully get the model, to 'learn' to associate the features with what we have in the desired output column shown in Figure 2. The training outputs will be a set of features, (given by the amount of items taken, and amount of items not taken in a Section) and an associated branch predictor. For supervised learning, the predictor may have to be run over large datasets a number of times for it's output to completely align with the desired output.

**After Training on a few traces:**

| Training Segment | # Taken | # Not Taken | Our Output | Desired Output |
|---|---|---|---|---|
| 1 | 7 | 3 | Bimodal | G-Share |
| 2 | 8 | 2 | Bimodal | G-Share |
| 3 | 2 | 8 | Bimodal | Smith |
| 4 | 5 | 5 | Bimodal | Bimodal |

**After Training on more traces:**

| Training Segment | # Taken | # Not Taken | Our Output | Desired Output |
|---|---|---|---|---|
| 1 | 7 | 3 | Bimodal | G-Share |
| 2 | 8 | 2 | G-Share | G-Share |
| 3 | 2 | 8 | Smith | Smith |
| 4 | 5 | 5 | Smith | Bimodal |

**After Training Is Complete:**

| Training Segment | # Taken | # Not Taken | Our Output | Desired Output |
|---|---|---|---|---|
| 1 | 7 | 3 | G-Share | G-Share |
| 2 | 8 | 2 | G-Share | G-Share |
| 3 | 2 | 8 | Smith | Smith |
| 4 | 5 | 5 | Bimodal | Bimodal |

Figure 2: *Training* Tables of example data illustrate what the training process for the predictor might look like.

**Feature Future applied to new Features**

| Training Segment | # Taken | # Not Taken | Our Output | Desired Output |
|---|---|---|---|---|
| 1 | 6 | 4 | Bimodal | |
| 2 | 1 | 9 | Smith | |
| 3 | 10 | 0 | G-Share | |
| 4 | 2 | 8 | Smith | |

Figure 3: *Feature Association* The table shows the predictor being run on a new set of inputs after being successfully trained.

3) *Runtime Inputs:* The runtime Phase would utilize a different set of inputs for the predictor during execution. This would include the set of features found in the most recent segment as well as the corresponding branch predictor to use for the next $n$ segments.

4) *Runtime Outputs:* The output of the runtime phase should be the program's execution, with each branch predictor being utilized over a set amount of segments of the programs execution, ideally with greater accuracy than any of the individual predictors over the entire execution.

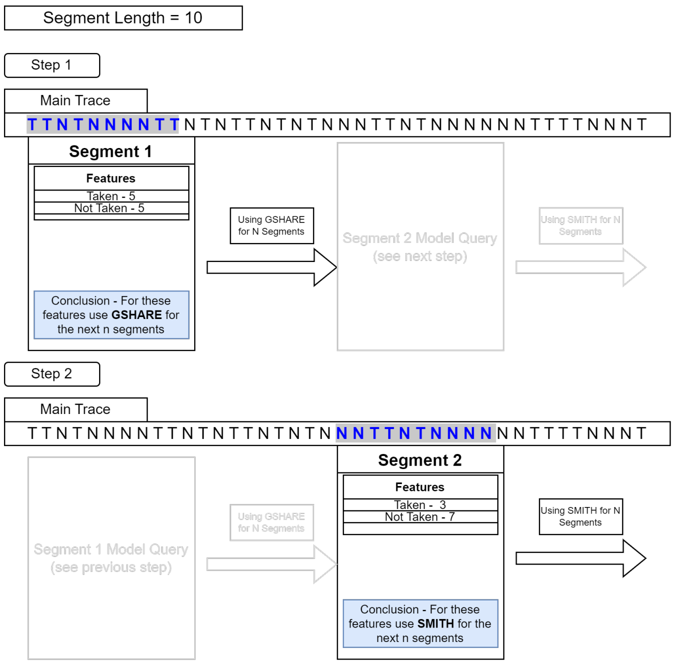
Segment Length = 10
Step 1
Main Trace
T T N T N N N N T T N T N T T N T N T N N N T T N T N N N N N N T T T T N N N T
Segment 1
Features
Taken - 5
Not Taken - 5
Conclusion - For these features use **GSHARE** for the next n segments
Using GSHARE for N Segments
Segment 2 Model Query (see next step)
Using SMITH for N Segments
Step 2
Main Trace
T T N T N N N N T T N T N T T N T N T N N N T T N T N N N N N N T T T T N N N T
Segment 1 Model Query (see previous step)
Using GSHARE for N Segments
Segment 2
Features
Taken - 3
Not Taken - 7
Conclusion - For these features use **SMITH** for the next n segments
Using SMITH for N Segments

Figure 4: *Predictor Execution* The graphic illustrates how the prediction window set during training by the user moves throughout the program execution, identifying features and determining which to utilize.

## C. *Assumptions*

For this design and approach there are a number of assumptions to consider. First, it is assumed that the global number of taken and not taken are not a focus and as such are relatively independent of feature detection., at least for our purposes. Features are determined largely by the relationship between values. In addition, the order of taken and not taken is highly important. The predictor will assign weights and biases to the features that will then influence choosing a particular predictor.

## D. *Supplementary Concepts*

It is assumed that greater accuracy can be achieved in a future implementation that can average out the results of each segment to provide a sort of momentum for a particular predictor. For example, if a 5 segments in a row have a high accuracy with Gshare, then we should have a bias that should saturate for the Gshare predictor. This will help take into account small anomalies that the traditional predictors in and of themselves are normally able to account for with a saturating counter.

## IV. Implementation

## A. *Dataset Preprocessing*

We will start by loading the dataset into memory and performing necessary preprocessing steps. This may include feature extraction, feature scaling, and handling missing values. We will also split the dataset into training and testing sets, typically using a 70-30 or 80-20 split.

### *B. Feature Engineering*

Next, we will extract relevant features from the dataset that can be used as input to the XGBoost classifier. This may include features such as the instruction type, the address of the branch instruction, the outcome of the branch, and any other relevant features are shown to potentially help in predicting branch outcomes.

### *C. Feature Analysis (Windows and Segments)*

To analyze the features, we use a sliding window segment of the previous predictions. This is set by us. The window size we chose was 12, meaning the last 12 branches are taken into account when querying the model. Our segment size was 4, meaning we queried the model every 4 branches. Segments do not overlap. The segment size is larger than 1 to reduce the overall number of calls that need to be made to help negate some of the cost of analyzing the branches.

### *D. Oracle*

To provide a baseline for a perfect branch predictor, we created a predictor based off future sliding feature windows. It also chose a predictor for every branch, in order to increase the chances of the optimal predictor as much as possible. This is not physically possible, but it is a useful point of comparison.

### *E. Model Training*

We will use the training set to train an XGBoost classifier. We will configure the hyperparameters of the classifier, such as the learning rate, the maximum depth of the trees, and the number of boosting rounds, based on experimentation and cross-validation. We will fit the classifier to the training set and monitor its performance during training using appropriate evaluation metrics, such as accuracy, precision, recall, and F1 score. The trained models would be stored in main memory on a host computer for querying at program runtime.

### *F. Libraries*

The following Python libraries were used in our simulator: NumPy [4], Pandas [5], Scikit [6], XGBoost [7], and Matplotlib [8].

## V. Evaluation

### *A. Model Evaluation*

Once the classification model is trained, we will evaluate the model's performance using the testing traces to assess its accuracy and generalization ability. If necessary, we will fine-tune the hyperparameters of the classifier based on the testing set performance.

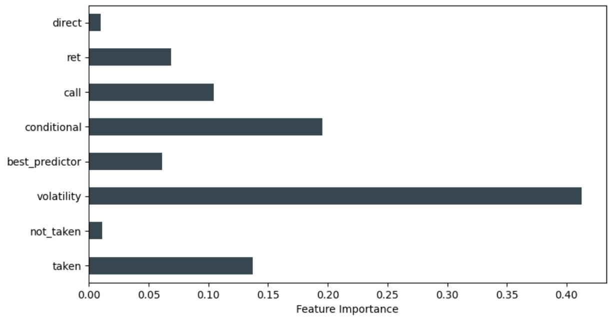

Figure 5: *Feature Importance* The various features used and their relative importance, for a given execution of GZIP.

From Figure 5, we see that volatility is the most important feature for determining which predictor to choose in the case of this certain trace. Whether the branch was conditional is also a significant factor. The feature determining whether the branch was from a return or call to a function also has some impact. A branch being direct seems to have no impact for this trace. The 'not taken' feature is deemed unimportant since 'taken' was likely already integrated into the decision tree, where it is seen to have some impact.

Other feature importance graphs for other traces vary widely, as shown in Figure 6 and Figure 7. While one graph shows similar importance for all features, the other shows high importance for whether a branch is a return or call to a function. In all of our test scenarios, it was found that volatility is consistently the most important feature.

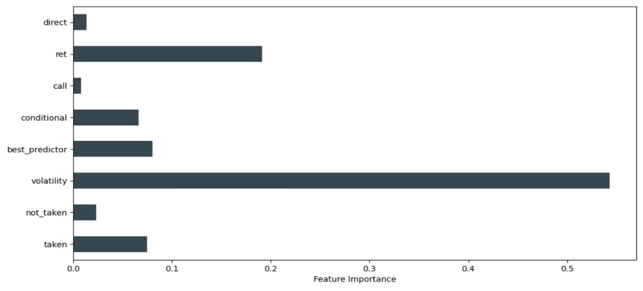

Figure 6: *Feature Importance* The various features used and their relative importance, for a given execution of GCC.

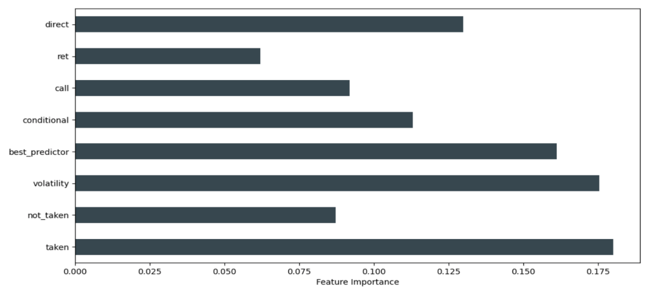

Figure 7: *Feature Importance* The various features used and their relative importance, for a given execution of JPEG.

## VI. Results

Results for a percentage of branches accurately predicted are summarized for each test file in the following table.

| File | JPEG | GCC | GZIP |
|---|---|---|---|
| Smith Predictor | 55.52% | 58.77% | 67.67% |
| Bimodal Predictor | 86.70% | 80.38% | 73.83% |
| Gshare Predictor | 86.22% | 75.09% | 72.45% |
| Oracle Clairvoyance | 89.96% | 83.74% | 77.89% |
| Our Method | 86.99% | 80.72% | 75.58% |

The bimodal predictor performed the best out of all of the basic predictors we tried. Compared to Bimodal prediction technique, our method showed an improvement in the number of branches predicted correctly of 0.29, 0.34, and 1.75 on JPEG, GCC, and GZIP respectively.

The average improvement in prediction accuracy was 0.793%.This means, on average, that our method is 1.0104 times as accurate as the Bimodal predictor.

### *A. Branch Predictor Comparison*

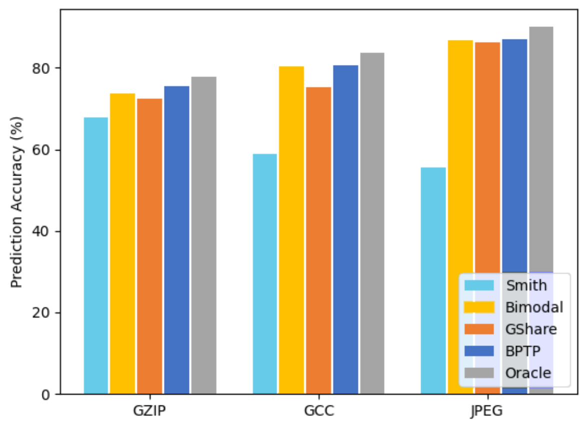

Figure 8: *Prediction Accuracy vs Prediction Method* The prediction accuracy of various branch prediction methods for multiple trace files.

As seen in Figure 8, our BPTP predictor performs higher than the traditional predictors for all traces. In some cases, this difference is very small, but this can be improved with both parameter and hyperparameter tuning. Still, it is very close to reaching the accuracy of the Oracle predictor, which is the ideal case of our model performing perfectly. Since our predictor is limited by which predictors we used in our model, we could include other various predictors to improve the overall result.

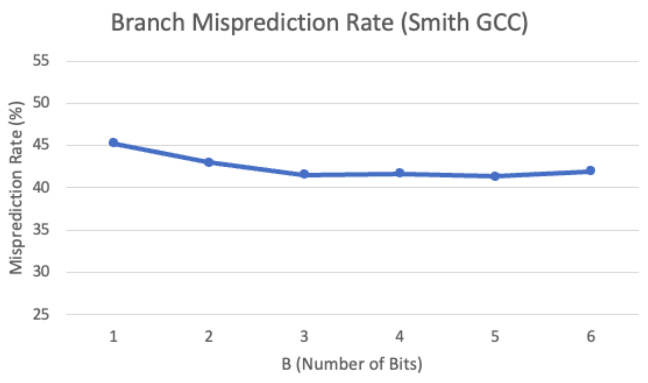

Figure 9: *Smith Branch Misprediction Rate (GCC Trace)* The Smith branch predictor's accuracy on GCC Trace.

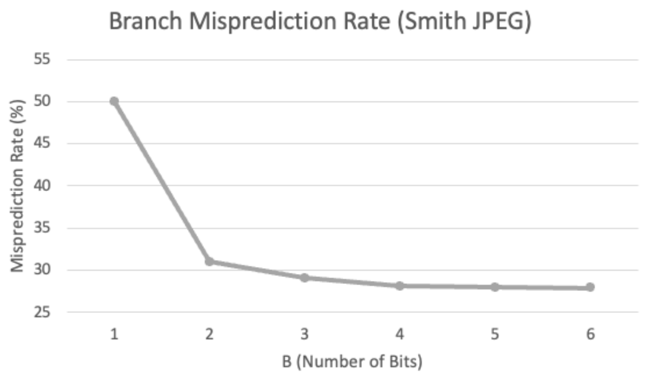

Figure 10: *Smith Branch Misprediction Rate (JPEG Trace)* The Smith branch predictor's accuracy on JPEG Trace.

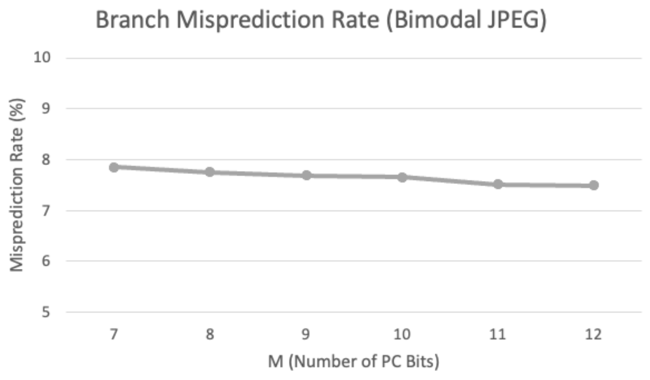

Figure 13: *Bimodal Branch Misprediction Rate (JPEG Trace)* The Bimodal branch predictor's accuracy on JPEG Trace.

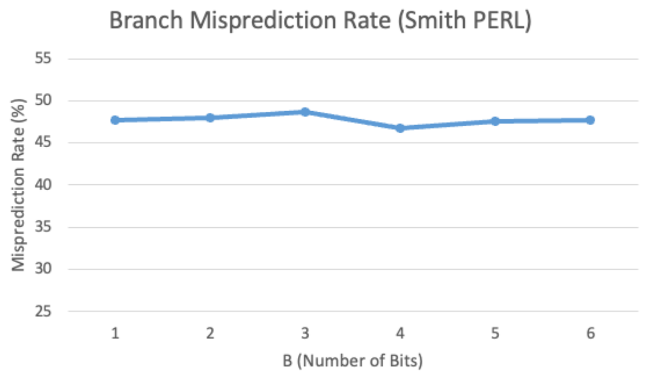

Figure 11: *Smith Branch Misprediction Rate (PERL Trace)* The Smith branch predictor's accuracy on PERL Trace.

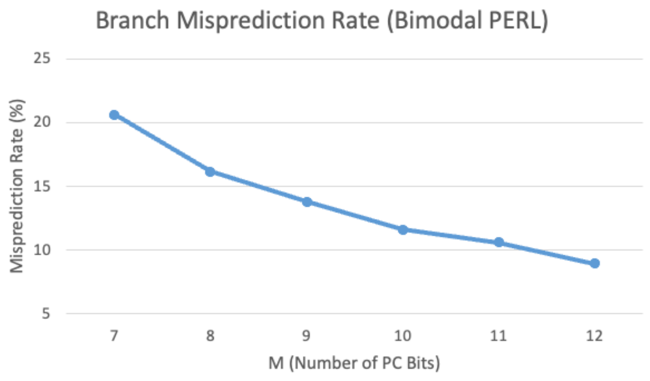

Figure 14: *Bimodal Branch Misprediction Rate (PERL Trace)* The Bimodal branch predictor's accuracy on PERL Trace.

The Smith Branch Predictor consistently had the lowest prediction accuracy among the branch predictors we utilized. As shown in Figures 9, 10 and 11, the misprediciton rate decreased as the number of bits used increased, if only slightly. While the Smith Branch Predictor is simple to implement, its prediction accuracy is comparatively low.

The Bimodal Branch Predictor seemed to perform the best between the three benchmark prediction methods we used. This predictor had the second highest prediction accuracy rate when compared to our Branch Predictor Technique Predictor. We see that both the Smith and Bimodal predictors performed the best when run on the JPEG Trace.

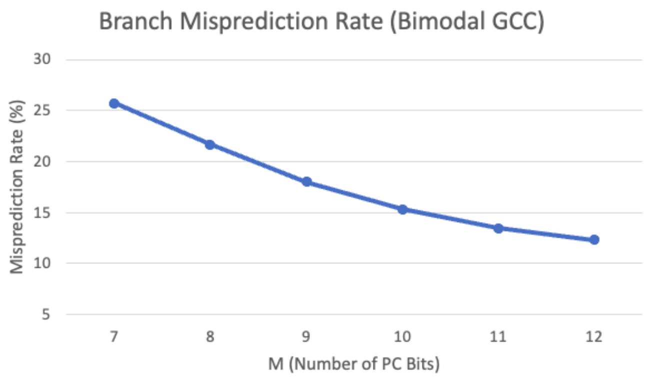

Figure 12: *Bimodal Branch Misprediction Rate (GCC Trace)* The Bimodal branch predictor's accuracy on GCC Trace.

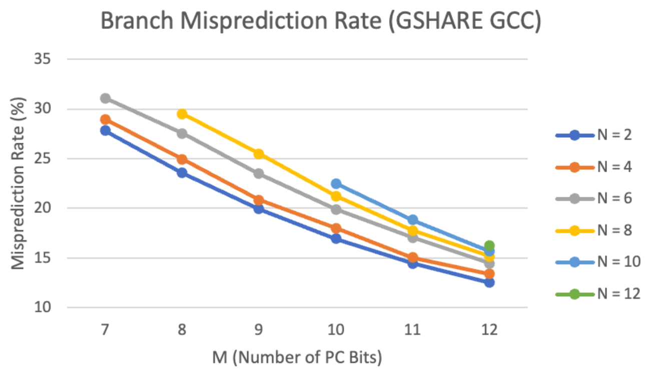

Figure 15: *G-Share Branch Misprediction Rate (GCC Trace)* The G-Share branch predictor's accuracy on GCC Trace.

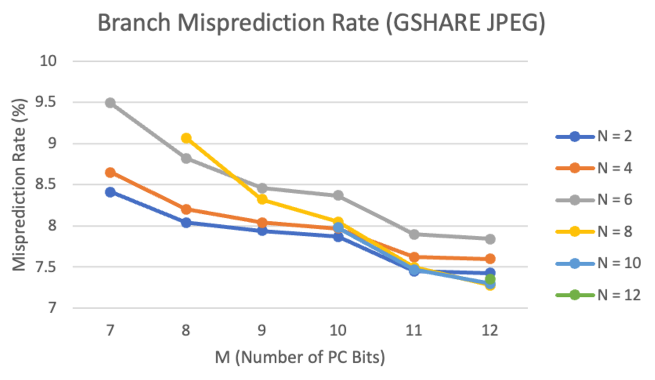

Figure 16: *G-Share Branch Misprediction Rate (JPEG Trace)* The G-Share branch predictor's accuracy on JPEG Trace.

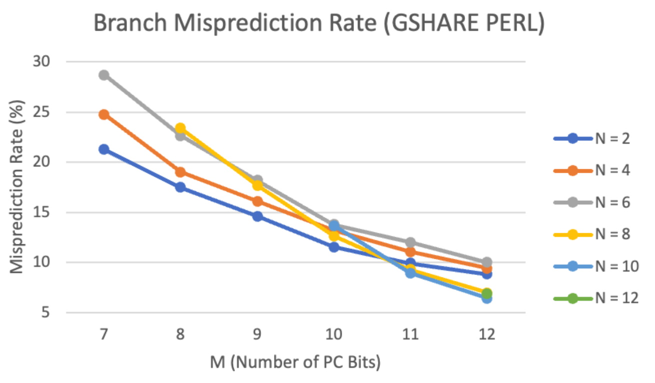

Figure 17: *G-Share Branch Misprediction Rate (PERL Trace)* The G-Share branch predictor's accuracy on PERL Trace.

The G-Share Branch Predictor accuracy improved as the number of PC Bits used increased for all trace files we tested it on. Again we see that the JPEG Trace produced the lowest misprediction rate, ranging between 7 to 10 percent, while the other traces' misprediction rate ranges from about 5 to 30 percent.

## VII. Summary

### A. Limitations

The limitations of our research include:

- small sample size of program trace files
- the number of prediction techniques
- time constraints preventing us from implementing our method in a containerized application
- arbitrary machine learning parameters

If the speed of branch prediction techniques improve in the future, or new branch prediction techniques are developed, our method would need to be trained more, but it would provide a great way to leverage the possible benefits of these new techniques. With more training over varied datasets, higher accuracy can be attained.

### B. Future Work

The machine learning parameters were chosen arbitrarily, so by altering them, significant optimizations could be made. Further optimization could be achieved by continuing to tweak the individual standard predictors used in our model, averaging over multiple runs of each sub/standard predictor, and using a larger set of traces to train the model. We would also like to implement more standard predictors in the machine learning model, such as Yeh/Patt, GEHL and TAGE branch predictors.

### C. Conclusion

Our goal was to develop a method for using a machine learning model to determine the best choice of branch predictor for a particular program based off of trace analysis. We achieved this by training on program trace files, and our model associated several variables with their benchmarked performance for each of a few types of branch prediction strategies.

It is important to note that, as the model only decides when to switch standard predictors, it may be practical to use in a dedicated branch predictor computer in a large-scale hypervisor setting that sees wide variation in branching patterns. It is possible the efficiency granted by the branch prediction method we have developed would outweigh the cost of running a dedicated computer to host our model on in some practical applications.

With respect to the most important predictors for which strategy will result in the best outcome, Volatility appears to be a key factor. Using a method based upon analysis of the sequence of branch predictions and their predictive power allows our method to be slightly more accurate than Gshare, Smith, or Bimodal within a reasonable degree of confidence.

### D. Source Code

The source code for this paper may be found at *https://github.com/eblissss/branch-prediction*

## References

[1] M. Olazaran, "A sociological study of the official history of the perceptrons controversy," *Social Stud. Sci.*, vol. 26, p. 611, 1996.

[2] B. e. a. Calder, "Corpus-based static branch prediction," *Acm Sigplan*, 1995.

[3] A. P. Shanthi, "Dynamic branch prediction," 2018. [Online]. Available: https://www.cs.umd.edu/~meesh/411/CA-online/chapter/dynamic-branch-prediction/index.html

[4] C. R. Harris, K. J. Millman, et al., "Array programming with NumPy," *Nature*, vol. 585, p. 357, 2020, doi: 10.1038/s41586-020-2649-2.

[5] W. McKinney, and others, "Data structures for statistical computing in python," in *Proc. 9th Python Sci. Conf.*, vol. 445, 2010, pp. 51–56.

[6] F. Pedregosa, G. Varoquaux, et al., “Scikit-learn: machine learning in python,” *J. Mach. Learn. Res.*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[7] T. Chen, and C. Guestrin, “XGBoost: a scalable tree boosting system,” in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining* in Kdd '16, San Francisco, California, USA, 2016, pp. 785–794, doi: 10.1145/2939672.2939785. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[8] J. D. Hunter, “Matplotlib: a 2d graphics environment,” *Comput. Sci. & Eng.*, vol. 9, no. 3, pp. 90–95, 2007.