Performance Analysis of Branch Predictors

Contents

[1.1 List of Figures 2](#_Toc40629459)

[1.2 List of Tables 3](#_Toc40629460)

[1.3 Report Information 3](#_Toc40629461)

[2 Introduction 3](#_Toc40629462)

[3 Background 3](#_Toc40629463)

[3.1 Simulation Implementation 3](#_Toc40629464)

[3.2 Performance Measurement Metric 3](#_Toc40629465)

[3.3 Predictors 4](#_Toc40629466)

[4 Algorithms 5](#_Toc40629467)

[4.1 Local Predictors 5](#_Toc40629468)

[4.2 Correlating Predictor 5](#_Toc40629469)

[4.3 Global Predictors 5](#_Toc40629470)

[4.4 Custom Predictor 5](#_Toc40629471)

[5 Results 6](#_Toc40629472)

[5.1 Local Predictors: 1, 2, 3, and 4 Bit Predictors 6](#_Toc40629473)

[5.2 Local Predictors: 64, 128, and 256 Entry BHT 7](#_Toc40629474)

[5.3 Correlating Predictors: 1, 2, and 4 Shift Register Size 7](#_Toc40629475)

[5.4 Global Predictors 8](#_Toc40629476)

[5.5 Complex Predictor 9](#_Toc40629477)

[5.6 Overall Comparison 9](#_Toc40629478)

[9 Conclusion 10](#_Toc40629479)

## List of Figures

Figure 1 …………………………………………………………….…………………………………………………………….………………… 8

Figure 2 …………………………………………………………….…………………………………………………………….………………… 9

Figure 3 …………………………………………………………….…………………………………………………………….………………… 10

Figure 4 …………………………………………………………….…………………………………………………………….………………… 10

Figure 5 …………………………………………………………….…………………………………………………………….………………… 11

## List of Tables

[**Table 1: Report version information.** 5](#_Toc25605647)

## Report Information

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version Number** | **Author** | **Comment** |
|  | 0.0 |  | Initial |
|  | 0.1 |  | Additions |
|  | 0.2 |  | Additions – Results & Figures |
|  | 0.3 |  | Additions |

**Table 1: Report version information.**

# Introduction

This report discusses branch prediction techniques and their impacts on CPU performance. More specifically, it addresses the performance impact of local and global predictors when varying the number of entries in the Branch History Table (BHT), the number of bits used for prediction, and the number of predictors used (correlating predictor if more than one). It also discusses a custom predictor that was designed to increase prediction performance. This report discusses accurate information about these predictors but does not fully cover the topic of branch prediction or its impact on performance by any means.

# Background

## Simulation Implementation

To simulate branch prediction, we use a C++ framework that is able to read real instruction traces and provides a prediction class that can be changed to simulate different prediction techniques. In particular, it provides two main methods: one to provide a prediction (so that the program may continue speculating), and one to update any tables with the actual branch result (taken or not taken). When the program is run, it records the total Mispredictions per Kilo Instruction (MPKI) for each of the 20 benchmarks and provides an average.

## Performance Measurement Metric

The measurement metric used is the average of Mispredictions per Kilo Instructions (MPKI) over the 20 benchmarks run by the simulation. It counts MPKIs rather than missed predictions per benchmark because often benchmarks contain varying amounts of instructions; in our case, all traces are the same amount of instructions, but typically counting the missed predictions per benchmark would be an inaccurate measurement of the impact on performance because a benchmark with significantly more instructions is bound to produce more missed predictions if the prediction technique is kept constant. Furthermore, we use Misprediction per Kilo Instruction rather than Misses per Kilo Instruction, which measures the number of cache misses but can still occur on a correct prediction.

## Predictors

As mentioned in the introduction, we use a variety of branch predictors: local predictors, correlating predictors, global predictors, and a more complex predictor that uses the branch address in conjunction with its history to predict the branch. To begin, it is important to understand the naming convention of the predictors. A predictor denoted as a (0, n) predictor is a local n-bit predictor that uses the state of n bits to predict whether a branch will be taken or not taken. An (m, n) *local* predictor is a correlating predictor that picks from 2m n-bit [(0, n)] predictors, where the n-bit predictor is chosen based on the true history of the last m branches taken or not taken.

The global predictors are similar to correlating predictors in the sense that they use the last m branches as a shift register. However, rather than using the shift register to index a predictor to use, it is used to index directly into the BHT of a (0, n) predictor. The correlating predictor does this by using 5 specific bits of the branch address. Our complex predictor uses two variables to decide on what predictor is used for the branch prediction: it considers the branch address, and the history of that branch address. Nearly all of the predictors analyzed are also discussed (with the same naming convention) in the textbook Computer Architecture: A Quantitative Approach by John Hennessy and David Patterson.

For the local (non-correlating) predictors, the report analyzes 1-bit, 2-bit, 3-bit, and 4-bit predictors where the number of entries in the BHT is held constant at 32. It also analyzes a 2-bit predictor when using 64, 128, and 256 entries in the BHT. For the correlating predictors, the number of bits used for prediction is kept constant, while changing the number of predictors used for prediction. More specifically the report analyzes the impact of using 2, 4, and 16 2-bit predictors. For the global predictors, the number of bits used for prediction is also kept constant, but the size of the shift register varies; this paper discusses global 2-bit predictors with a shift register of sizes 2, 4, and 8 bits, which implies BHT sizes of 4, 16, and 256 respectively. For the complex predictor, this paper discusses the performance for two sets of configurations: keeping the number of history bits constant and keeping the number of table entries constant. In specific, the first set has the history bits as a constant, and the number of table entries tested were 64, 128, and 256. The second set has the table entries as a constant, and uses 2, 4, and 8 bits to keep track of the history.

Another important detail that must be understood before continuing is the functionality of our 2-bit predictors. For all configurations in the following paper, the 2-bit predictor differs slightly from normal 2-bit predictors. For all other configurations, the bit predictors increase their value by one if the branch outcome is taken and decreases if the branch outcome is not taken. Of course, all values are constrained by the number of bits used. For the 2-bit predictors used, there are 4 (2^2) states allowed which are named: strongly not taken, weakly not taken, weakly taken, and strongly taken. Essentially, the 2-bit predictors discussed in this paper will jump from weakly taken to strongly not taken upon a miss, and from weakly not taken to strongly taken upon a hit. This kind of 2-bit predictor is also discussed in the textbook mentioned above.

# Algorithms

## Local Predictors

The local predictor works by setting a branch history table (BHT) size and predictor bit-size in order to determine how large of a table and how large a prediction will be. In the case of the first two parts of this project, the local predictor is tested while holding the BHT size constant and changing the size of the predictor to decrease miss rate and increase the efficiency of branch prediction and while holding the size of the predictor constant at 2-bits and changing the BHT size to gauge the MPKI performance of the branch prediction algorithm. The local predictor works by storing predictions in the BHT for instructions dependent on the branch instruction’s address. The address gives information of the location of the prediction within the table and the prediction is used to decide whether the branch is taken or not taken. An incorrect prediction results in a miss and the prediction changes to move one state closer to the target prediction (I.e. the correct prediction for the branch).

## Correlating Predictor

The correlating predictor works by enabling multiple BHT for instructions encoded differently. In other words, branch instructions are mapped to different categories. Then each BHT is mapped to one or many, but usually not all, of the branch categories. This enables more fine-tuned prediction of the branches and the MPKI reduces for the overall prediction mechanism. This mechanism of prediction ensures that predictions are attributed to the history of a single branch category, instead of for all branches, as is done in the local predictor. This enables branch predictions to be made based on accurate history for a branch making the predictions more reliable.

## Global Predictors

The global predictor is like the correlating predictor with the one difference that the shift register used to attribute the branch history to branch categories is used to select a BHT instead of a predictor, as is done in the correlating predictor. This provides flexibility with regards to which branch instructions get predicted based on which BHT. The larger BHT can be saved for branches that occur more often with the smaller BHT for branches that are used less often but are more likely to affect the code performance if missed.

## Custom Predictor

The custom predictor is similar to the correlating predictors in the sense that it uses both the branch address and the history of a specified number of branches to index into a predictor. The biggest difference is that the custom predictor uses the history pertaining to a specific branch (results of the past m branches on a specific branch), whereas the correlating predictors use the global history (the past m total branches) to decide on a predictor. To implement this, the custom predictor needs to hold an array of history; the history is required for each table entry. Furthermore, the predictor is a 2D array much like for a correlating predictor.

When making a prediction, the predictor first finds the address corresponding to the branch. If the number of bits used to specify the address is said to be x, the history array will have 2^x entries. Unlike a correlating predictor, the first index into the predictor is the branch address. The result after indexing with the branch address is an array that contains different predictions for different history possibilities. The history for that specific branch is found by using the branch address to also index the history array. Once found, that specific history is used to predict the next branch outcome. At the time of the branch evaluation, the predictor will be able to see if the actual outcome of the branch was taken or not taken. It updates the appropriate place in the history array by indexing into it via the branch address.

To summarize, this predictor keeps track of some number of bits that represent branch history and has some number of table entries. Each table entry has a branch history and a 2-bit predictor associated with it, so the branches history is the main determinant of the final prediction. The number of branch history bits and the number of table entries are the two dependent variables when analyzing the overall performance of the predictor. Another factor that could be analyzed is using different bit predictors, rather than the 2-bit predictor used in this custom predictor.

# Results

The following section will go into detail about the results from the various branch predictors explained in the previous sections.

## Local Predictors: 1, 2, 3, and 4 Bit Predictors

For the first task, our group tested four different local predictors: 1 bit, 2 bit, 3 bit, and 4 bit, all using a 32-entry Branch History Table (BHT). We ran these branch predictors on different trace benchmarks to analyze their results. As seen in Figure 1, there was significant improvement after using more than 1 bit for the prediction. The results for the 2 bit, 3 bit, and 4 bit predictor are relatively similar with 15% - 20% decrease in the average Mispredictions per Kilo-Instruction (MPKI) compared to the 1 bit predictor. Thus, we can conclude that having more than one bit in branch prediction can increase the accuracy significantly, while adding more bits from there won’t necessarily improve the prediction.

Figure : shows the results of the 2, 3 and 4 bit predictors using a 32-entry Branch History Table, run on the trace benchmarks. It can be seen that over the entirety of the benchmarks the 3-bit predictor was the most accurate in branch predictions.

## Local Predictors: 64, 128, and 256 Entry BHT

Once we analyzed the results of the local predictors with a 32-entry BHT, we tested a 2 bit local predictor with several different sized BHTs: 64, 128, and 256 entry. We ran the predictors on the same trace benchmarks and the results can be seen in Figure 2. As Figure 2 shows, there was significant decrease in MPKI when increasing the BHT entry size. There was slightly more than 15% improvement from 32 entry to 64 entry BHT. The 128 entry BHT branch predictor showed a decrease of around 27.5% and the 256 entry had a decrease of almost 40%. Our results show that branch predictors can have meaningful improvement when increasing the entry size of the BHT.

Figure : shows the output of the benchmarks ran on a 2-bit predictor with varying entry sizes for the Branch History Table (64, 128, 256). It can be seen that the 256-entry table performed the best, leading to the biggest decrease of mispredictions over the 32-entry predictor.

## Correlating Predictors: 1, 2, and 4 Shift Register Size

Next, we kept the number of branch history tables fixed at 32 and instead implemented a (1,2), (2, 2) and (4,2) predictor. As you can see from the table using a bigger shift register increased performance, as Figure 3 shows a correlation between both shift register size and bits per predictor in terms of decrease in MPKI. The (4,2) predictor performed the best, having a decrease MPKI of 45% over the (0,1) predictor, even though both predictors had the same BHT size.

Figure : shows the output of the benchmarks ran on (1,2), (2,2) and (4,2) branch predictors where the first number indicates the size of the shift register used and the second the number of bits used.

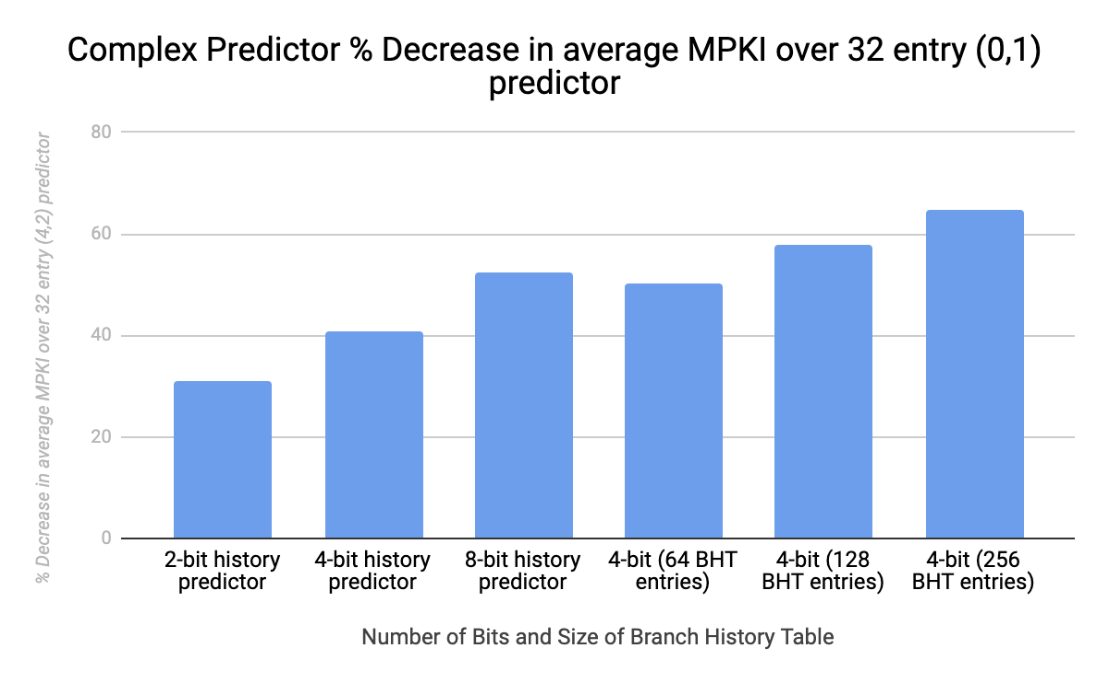
## Global Predictors

Next, we tested global predictors which use a shift register to index the BHT versus the branch instruction bits. From Figure 4 it can be seen that a larger shift register to index the BHT leads to a higher decrease in MPKI, which makes sense because you are also increasing the size of the BHT, giving you a greater history with which to make predictions.

*Figure 4: shows the decrease in MPKI over the benchmarks ran on a (4,2) and (8,2) global predictor.*

## Complex Predictor

Finally, we tested our complex predictor which is discussed above in section 7.4. The complex predictor outperformed the 32 entry (0,1) in every with case, with a range in decreased MPKI between 30-65%. The complex predictor also outperformed every other predictor we tested in terms of mispredictions except for the correlating predictor, although the decrease in MPKI was close.



## Overall Comparison

Analyzing the data as a whole shows a couple of key observations. First, for a fixed size BHT, the data indicates that more bits in your predictor does not always lead to better predictions and seems to reach a limit at around 3-bits after which performance actually decreases. Our findings also show that typically the larger the branch history that a predictor can store, the more effective the predictions.

One way to analyze the actual benefits of these various predictors is to compare their performance to the cost of the physical predictor. One way to measure the cost of a predictor is to analyze the amount of memory required by each. There is a clear trend shown in the results above that using more memory typically results in better performance. In terms of the predictors shown above, the local predictors use the least amount of memory, where an n-bit predictor needs n bits for each Branch History Table entry. For the first configuration shown in section 8.1, the number of table entries is held constant at 32. Section 8.2 has the same configurations as 8.1 but varies in the amount of table entries.

For section 8.3, each correlating predictor selects a 2-bit predictor depending on the history of the branches. This means for a correlating (m,2) predictor, m-bits of history are tracked and are indexed by the history bits and the branch address. The number of branch address corresponds to the number of table entries, called k. Therefore, we need a total of 2m \* 2k total 2-bit predictors, whereas the local predictors require 2k 2-bit predictors.

On the other hand, the global (m,2) predictors explained in section 8.3 actually require less memory compared to the correlating predictors. The global (m,2) predictors use the history of the last m branches to index a 2-bit predictor, and therefore require 2m 2-bit predictors. Using a (2,2) correlating predictor results in an MPKI of 18.3, whereas using a (2,2) global predictor results in an MPKI of 41.645. This is a clear example where the extra memory used by the correlating predictor increases the performance benefit.

The custom predictor is very similar to the correlating predictor in the sense that it also requires 2m \* 2k total 2-bit predictors. However, the custom predictor uses additional memory to keep track of the history for specific table entries. So, if the predictor is keeping track of m history bits and has k table entries, the total number of history bits is m \* k. The (2,2) correlating predictor with 32 table entries results in an MPKI of 18.3, but the custom (2,2) predictor with 32 table entries has an MPKI of 20.19. This is interesting because although it uses more memory, it is still slower than the correlating predictor in this case. This correlation remains when keeping the number of history bits and number of table entries between the two predictors constant relative to each other.

# 9 Conclusion

In this project we analyzed the performance of different branch prediction techniques over twenty different benchmarks to measure their performance. Analysis was mainly done by examining the benefit of the branch predictions to their base prediction states for a particular type of branch prediction technique. For this project the average performance of the twenty benchmarks was used, as we were interested in finding the performance of the techniques as a whole, not specifically catered to a specific program or benchmark. The performance was then collected and put into graphs which allowed us to more easily analyze the data and compare the outputs of different branching techniques.

One major takeaway from this paper is that using more memory in a seemingly smart way does not guarantee better performance in the predictions. This is proven by the custom predictor discussed above, which uses more memory (compared to a correlating predictor) to keep track of specific branch history. While in theory the prediction should be more accurate, the performance is decreased compared to predictions made by correlating predictors.

However, while increasing the amount of memory may not always increase performance, the results discussed above show that using a correlating predictor results in the best performance compared to all other configurations discussed in this paper. In terms of applying these techniques, the best choice is dependent on two main factors: the cost (usually measured by complexity and required memory) limit, and the performance desired.