# Branch Predictor Project

Final Project CSE240A Wi/2023

## I. Introduction

Branch predictors are an important aspect of CPU designs. Their goal is to predict the outcome of conditional branches in a running program. Correct predicted branches significantly improve performance as the processor can prefetch and execute code in advance. A miss prediction slows down the execution because the processor must discard all speculative work beyond the branch. Using an accurate branch predictor in high-performance, deeply pipelined superscalar processors is essential [1].

One approach to branch predictors is to capture and use the correlation between previously executed branches in some form to guess the outcome of a future branch. A so-called "gshare" uses the history of the most recent outcomes on a global scale, while a local predictor saves the history of each branch and makes its decision based on the saved "local" about the history. Combining these two approaches leads to a hybrid predictor that also learns which predictor performs better for a certain branch. Another more complex approach is to train functions that associate a weight to each entry in global history, thus allowing it to outperform the previously mentioned predictors. All mentioned predictors, except the local predictor, will be discussed in detail in this report.

As compute increases, advances in machine learning also have found their way into branch predictors [7]. For example, deep learning algorithms can find hidden correlations [4] and are able to outperform the traditional branch predictors.

### A. Pattern history table

All predictors discussed use a pattern history table (PHT) to make a final decision. The PHT can be seen as a four-stage state machine with the following states:

1) Strongly not taken
2) Weakly not taken
3) Weakly taken
4) Strongly taken

For each prediction, the branch predictor looks up an address, based on some unique logic, in the PHT and then predicts the branch. After that, the predictor updates the PHT table based on the branch's actual outcome, whether weakening or enforcing their current belief.

### B. G-Share

The global share branch predictor uses the last n bits of the program counter (PC) with an XOR combination from the history of the last m bits, shown in 1. The XOR increases the entropy, meaning that the created address is more unique. With this method, the gshare predictor can capture $2^n$ different global patterns and over time adjust its PC accordingly over time. Advantages of the gshare are that it is able to learn to interfere the outcome based from the direction of previous branches [5], and that it is less complex than the other predictors. One disadvantage is that the size of the PC table grows exponentially with an increase in n. Due to its simplicity the gshare leaves room for improvement one studied found that not training the predictor on "simple" to predict branches can improve performance by 38% [1]. Overtime the gshare also found its way in to hybrid predictors one of which we will discuss in the next section.

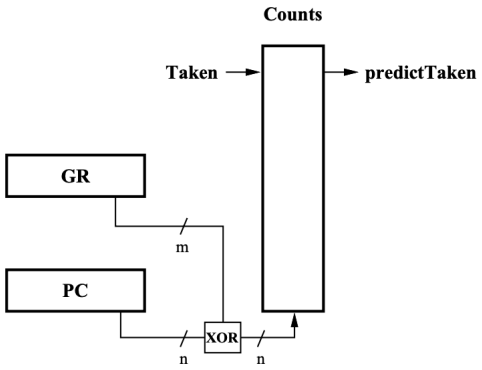


Fig. 1. G-share branch predictor [6]

### C. Tournament

We implemented Alphas 21264 tournament branch predictor, a tournament branch predictor consisting of multiple predictors. Each individual predictor makes a prediction, and the tournament predictor then chooses dynamically which predictor to follow for a particular PC, visualized in 2. Our predictor is an ensemble out of the previously mentioned gshare and a local predictor. The local predictor works by using the last n bits of the PC to address a local history table. The local history table saves the outcomes of the truncated PC address and than uses a PHT to make decisions. The tournament predictor uses another PHT which is indexed using the global path history to make a choice of which predictor to choose. The predictor outperforms both individual predictors with larger tables 90-100% of the time [5]. One drawback of the Alpha 21264 is that its performance is up to 25% reduced when it tries to predict out of order executions, since decisions have to be made before the global history register is updated [2].

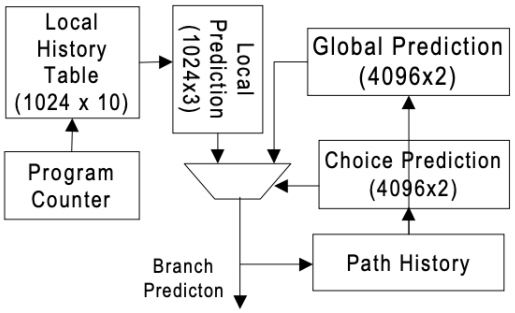


Fig. 2. Tournament branch Predictor, with local predictor on the left and gshare as global predictor [5]

### D. Perceptron

The perceptron branch predictor leverages the perceptron as the method for learning correlations between ranch outcomes from a global history, and the current branch [3]. The perceptron can be described as follows:

$$y = w_0 + \sum_{i=1}^{n} x_i w_i \tag{1}$$

In the use case of branch prediction, each x represents a branch outcome in the global history with the value 1 if taken and -1 if not taken. If y is greater than 0, then the branch prediction is taken, else it's not taken. There exists a table of perceptrons that use the lower bits of the PC. The number of PC bits used is a hyperparameter determining the number of perceptrons used. The greatest benefit of the perceptron and why it outperforms GShare:13 and Tournament:9:10:10 is because it can look at much larger bit strings of global history. Accurate branch prediction is paramount to perform deeply pipelined processors. According to Eric Sprangle and Doug Carmean [8], branch misprediction latency is the largest contributor to performance degradation as pipelines grow deeper. Deeper pipelines can increase processor frequency by 100% [8] but this is dependent on accurate branch prediction. One of the coolest features that come implicitly built into the perception is that result y from the perceptron also acts as a measure of confidence in the next branch prediction. The dot product is a way to measure how strongly vectors correlate. For example, y values close to zero mean the vectors are orthogonal and prediction is uncertain. Future iterations on this perceptron design may leverage this information to use a tournament predictor to choose another branch predictor if certainty is near 0. Additionally, the perception certainty can be used to speculatively execute both branches if confidence is low [3].

*1) Memory and Performance:* For this specific implementation, the major reason why the perceptron is able to use a large global history is due to the amount of memory required grows $n \log n$ with respect to the amount of history. First there is a linear relationship between the history length and the number of bits required to represent a single weight. For a single perceptron, the number bits needed to represent the weight is equal to the floor of $\log_2(1.93 * historylength + 14)$ plus 1 for the sign bit [3]. A perceptron contains a global history length number of weights. Therefore, the total number of bits required to represent a perceptron grows $n \log n$ with respect to the global history length. The last major factor on the memory is the number of entries in the perceptron table. Finally, we need to add the bits required for the global history register. GShare grows exponentially with the length of global history. This is due to the PHT being indexed by an XOR of the lower PC bits and the global history register. To map everything global history to a single entry in PHT requires $2^{globalhistorybits}$. This explains why for similar amounts of memory, the perceptron will outperform GShare when correlated branches are far apart. For our implementation that beats GShare:13 and Tournament 9:10:10, the number of perceptron entries is 256 entries and 30 bits of history (in this paper, this will be denoted as Perceptron:entry_number:bits of history). Using the process described above, the amount of memory used for the perceptron is 55583 bits. Experimenting with several history lengths and number of perception entries, the high number of perception entries helps mitigate the amount of destructive aliasing that occurs.

*2) Limitations::* Perceptrons lack in their ability to learn linearly inseparable branches that GShare is capable of learning [8]. Linear separability can be understood best by using the example of a classifier. If a set of points can be partitioned with a line into a set of True points and a set of False points, then the function is linearly separable. For example, the perception can learn logical AND but not exclusive OR. In cases with linearly inseparable branches where the branches require short histories [8], GShare can outperform the perceptron. This makes sense because if there is no correlation with the larger history size, then it doesn't make a contribution on branches that can be correlated with short histories. The main advantage of the perceptron is weakened, while the ability for GShare to learn a larger set of boolean functions allows it to perform better. This explains the results we see with "fp_1" as GShare:13 performs nearly as well Perceptron:256:30 with a misprediction rate of 0.825 figure 3 compared to 0.818 figure 6.

## II. Implementation

We implemented the three described predictors in C and used the boiler plate code provided in the Github repository [1]. We compared the three predictors using six different traces and reported the results section.

### A. Contributions

We Fritz Girke and Connor Kuczynski, mostly worked together on all project parts. Connor Kuczynski worked heavily on the perceptron algorithm, while Fritz Girke focused more on the branch and tournament predictor implementations.

[1] https://github.com/rskpdev/CSE240A/tree/master/src

### B. G-share

We initialize the length of the global history, with the console input n. And create a PHT array with size $2^n$, different from 1; we keep m and n always equal. We initialize the created PHT tables with ones that correspond to weakly taken. For a prediction we use bit wise XOR to create the PHT address which and tan translate the states 0-3 to a one for taken and a Zero for not taken. In the training step, we first update the PHT with by comparing our prediction with the actual outcome and changing the PHT state accordingly. Then, in the final step, we update our global history register.

### C. Tournament

Through the terminal, we set three input values corresponding to the PC index length, which determines the length of our local history table, the local history length, which sets the width of the local PHT array and the global register length, which is used to initialize the gshare and the PHT table of the tournament Predictor. All arrays are created with length $2^n$ (n for each of the mentioned numbers) . In the predict step we get the prediction from the gshare and local predictor, we than look up which predictor gets selected by checking the value in the tournament PHT if its $< 2$ we select the local predictor otherwise the output of the gshare gets returned. In the training step we first train the gshare, the local predictor. We than check if the two predictions made by the two predictors are different from each other, if that is the case we update the tournaments PHT table.

### D. Perceptron

First, the index into the perception table is created by hashing the PC. We experimented with a few hashes namely, using the number of history bits and using modulo with the number of perceptrons but that led to destructive aliasing. The current implementation uses a simple mask created from the number of perceptrons in the table. An improved hash function may better correlate local branches while still minimizing the amount of destructive aliasing. Next the dot product of the recent global history register and the indexed perceptron create the output y. If y is greater than 0, the branch is taken, else it is not taken. Training occurs if one of two cases are true, (1) the prediction is not equal to the real outcome of the branch, or (2) the absolute value of y is less than the maximum training threshold. This threshold is found using:

$$threshold = 1.93 * historylength + 14 \quad (2)$$

If the real outcome is NOT TAKEN then each weight of that perceptron is decremented by -1 times its respective bit of recent global history. Conversely, if the real outcome is TAKEN then each weight of that perceptron is incremented by 1 times its respective bit of recent global history. This has the effect of strengthening weights that correlate correctly with the outcome and weakening the weights that correlate incorrectly with the outcome. Finally the recent global history shift register is shifted and updated with the most recent outcome.

## III. Observations

A general trend is traces that have less than 1% mispredicted rate like fp_1 and int_2, see all three branch predictors performing exceptionally well. All three branch predictors can learn easy-to-correlate patterns at a very high level of accuracy. The perceptron outperforms GShare and Tournament when ways are difficult to understand. For example, Perceptron performs around 7% better than the other two for int_1.

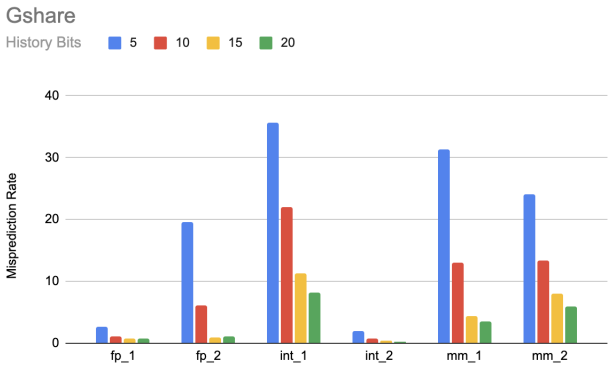


Fig. 3. Gshare with varying global history length

In figure 3, we display the results of gshare with different global history lengths. For example, we can observe that a larger history length corresponds to a lower miss prediction rate. This can be explained by the increased information the predictor can capture due to the exponentially enlarged history table size.

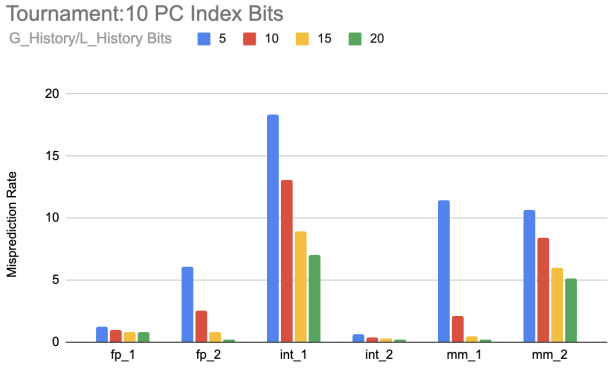


Fig. 4. Tournament with 10 PC Index and varying global and local history length

Figure 4 shows the experiments we ran for the tournament predictor, fixing the PC Index length to 10 and changing the global history and local history in the same increments. We see a similar behavior to gshare, where an increase in table size corresponds to an improvement in performance. Furthermore, comparing the results to figure 3, we can see that given the same global history length tournament outperforms gshare on all test cases.

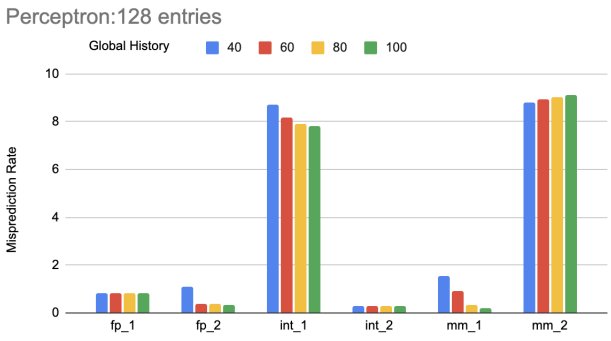


Fig. 5. P256 Perceptrons with varying number of weights

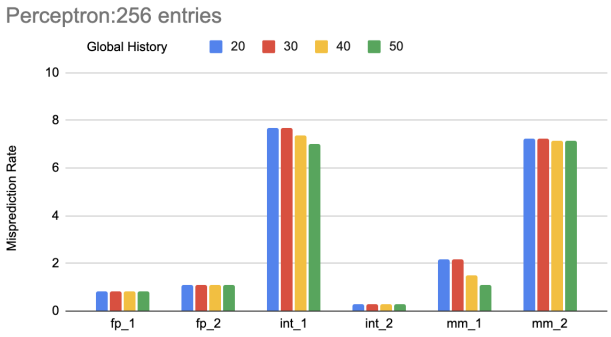


Fig. 6. 256 Perceptrons with varying number of weights

Next, we study the performance differences of more memory toward longer global histories versus more perceptrons. One of the interesting patterns is that more global history does not necessarily guarantee lower mispredictions. For example, with mm_2 we see that when using Perceptron:128, misprediction stays roughly the same around 9% even with double the amount of global history. This may be because the correlations learned by the perception occur at lower address distances so adding more history doesn't improve the correlations. Furthermore, adding more global history appears to dilute these correlations slightly. Perceptron:256 however performs roughly 2% better. The correlation–even at just 20 bits of global history–performs better than Perceptron:128. This is likely due to less destructive aliasing occurring because there are less collisions from the hash function. Perceptron:128 does outperform Perceptron:256 for trace mm_1. It roughly halves the misprediction rate, meaning this trace correlates well with branches that are 80 to 100 branches apart. Perceptron:256 does not improve significantly with more global history bits which is contrary to previously suspected. This may be because too many branches map to different branches so a smaller number of branches are correlated compared to Perceptron:128. We can conclude that it is better in this instance that more perceptrons is a better use of resources compared to more bits of history. Past a certain point, the gains from more global history diminish compared to adding more perceptrons. Further study in different hashing functions may improve branch correlation similar to how GShare uses XOR to reduce collisions.

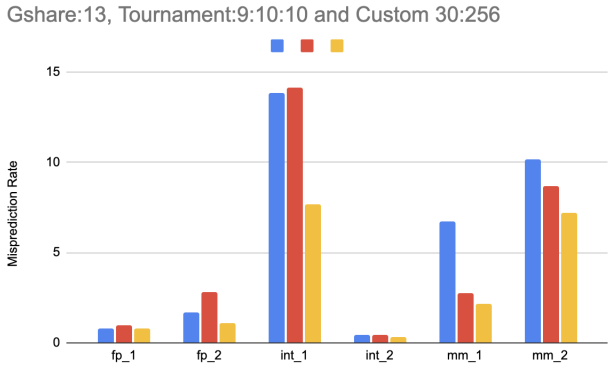


Fig. 7. Miss prediction rate of branch predictors for each trace

From table I, visualized in figure 7, we observe that our perceptron predictor can do better than gshare:13 and tournament:9:10:10 on all traces. In addition we can see that gshare:13 is able out to perform the tournament predictor in the "int_1" case. The reason for this could be that in this particular trace, more accurate interference between the global history leads to better results, making gshare:13 beat the tournament predictor which only keeps the last 9 outcomes in the global history register.

| | Gshare:13 | Tournament:9:10:10 | Custom 256:30 |
|---|---|---|---|
| fp_1 | 0.825 | 0.993 | 0.818 |
| fp_2 | 1.678 | 2.82 | 1.107 |
| int_1 | 13.839 | 14.13 | 7.692 |
| int_2 | 0.42 | 0.427 | 0.295 |
| mm_1 | 6.696 | 2.775 | 2.186 |
| mm_2 | 10.138 | 8.699 | 7.22 |

TABLE I
MISS PREDICTION RATE OF THREE PREDICTORS

## IV. Conclusion

In this project, we successfully implemented three predictors, gshare, tournament, and perceptron. We observed that an increase in information that is available to a predictor leads to better performance. Furthermore, an increase in complexity also improves prediction accuracy, making perceptron the most accurate predictor, followed by tournament and then gshare.

## References

[1] P.-Y. Chang, M. Evers, and Y. N. Patt, "Improving branch prediction accuracy by reducing pattern history table interference," *International Journal of Parallel Programming*, vol. 25, no. 5, pp. 339–362, Oct 1997. [Online]. Available: https://doi.org/10.1007/BF02699882

[2] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 67–76.

[3] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 197–206.

[4] R. Joseph, "A survey of deep learning techniques for dynamic branch prediction," *CoRR*, vol. abs/2112.14911, 2021. [Online]. Available: https://arxiv.org/abs/2112.14911
[5] R. Kessler, E. McLellan, and D. Webb, "The alpha 21264 microprocessor architecture," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*, 1998, pp. 90–95.
[6] S. McFarling, "Combining branch predictors," 1993.
[7] A. Smith and S. Diego, "Branch prediction with neural networks: Hidden layers and recurrent connections," *Department of Computer Science University of California, San Diego La Jolla, CA*, vol. 92307, 2004.
[8] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 25–34, 2002.