CSE240A (WI24): Branch Predictor Report

Samyak Mehta, Kunind Sahu

***Abstract*—This report presents the implementation and analysis of three branch prediction techniques - GShare, Tournament, and a novel Perceptron-based predictor - for enhancing processor performance. Branch prediction aims to anticipate conditional branch outcomes, reducing pipeline stalls and improving instruction throughput. The GShare predictor leverages global branch history, while the Tournament predictor combines local and global predictors through a choice logic. The Perceptron-based predictor employs a neural network approach to capture complex branch patterns. Detailed implementations are provided, highlighting data structures and algorithms. An extensive evaluation across various benchmarks and configuration parameters reveals the trade-offs and performance characteristics of each predictor.**

I. Introduction

In computer architecture, a branch predictor is a digital circuit that attempts to anticipate the outcome of a conditional branch instruction (such as an if–then–else structure) before it is definitively known. The primary purpose of a branch predictor is to enhance instruction pipeline flow and improve overall performance in modern pipelined microprocessor architectures.
Branch Prediction is used to obtain pipeline efficiency. Without branch prediction, the processor would have to wait until a conditional jump instruction (e.g., an if statement) has passed the execution stage before the next instruction can enter the fetch stage in the pipeline. Predicting branches allows the pipeline to continue fetching and executing instructions, reducing idle time.
High-performance microprocessors rely on branch prediction to minimize the impact of branch mispredictions, which can cause significant delays. What is even more important to know, is that, modern processors have very long pipelines (10 to 20 clock cycles), making accurate branch prediction crucial for maintaining efficiency.

A. General Working

When the processor encounters a branch instruction, it has to decide whether to predict that the branch will be taken or not taken. There are two types of predictors, static and dynamic.Static Predictors make fixed predictions based on statistical or heuristic analysis of branch instructions whereas dynamic predictors use historical information about the program's execution to adapt and make more accurate predictions.
Many modern processors use two-level adaptive predictors that consider the global history of branches (behavior across the entire program) and the local history (behavior of a specific branch).
During program execution, the branch predictor makes predictions based on historical information. If the prediction is correct, the predictor is considered "correctly trained," and the associated history information is updated. If the prediction is incorrect, the predictor adjusts its strategy to improve future predictions.

B. Advantages

An advantage of branch prediction is that it provides **improved performance**. Accurate branch prediction minimizes pipeline stalls, leading to faster execution times of a process. Predicted branches allow better utilization of execution resources hence it is a move towards Efficient Resource Utilization. Increased efficiency enables faster operations, especially in processors with lengthy pipelines which, require stalling for prolonged cycles, if branch prediction is not used.

C. Disadvantages

Incorrect predictions result in wasted cycles. Mispredicted branches cause pipeline stalls, impacting overall performance. Advanced branch predictors require additional hardware and complexity. Balancing accuracy and resource usage is challenging. There exists a trade-off between prediction accuracy and the cost of implementing sophisticated predictors. Additionally, overly complex predictors may not always yield significant performance gains.

D. Current Trends

Some recent research and developments explore the use of Neural Networks [4] for branch prediction. Neural predictors use machine learning techniques to adapt and learn patterns in branch behavior. Neural predictors aim to capture complex dependencies in branch patterns - for example, using both local and global history-based correlations that may be challenging to model using traditional techniques and other heuristics.
Hybrid predictors combine multiple prediction strategies to achieve better accuracy across different types of branches. These predictors might include a combination of local predictors, global predictors, neural predictors, or other strategies to leverage the strengths of each.
Reinforcement Learning and other adaptive algorithms are being explored to dynamically adjust prediction strategies based on program execution characteristics [7].

E. Our Work

In our project, we implemented three branch predictors viz. **GShare** [6] [5], **Tournament** [3] [2] and a **Perceptron-based** branch predictor [1]. We implement the algorithms in **C** and share some basic implementation details. Additionally, we analyze the performance of these predictors using different hyperparameter configurations to see how the choice of hyperparameters affect their performance and reason about why that might be.

II. Implementation

We have implemented G-share, Tournament and Perceptron models for Branch prediction.

A. *Distribution of Work*

- **Samyak Mehta** - Implemented GShare and Tournament Algorithms. Designed Experiments for Benchmarking
- **Kunind Sahu** - Implemented the Perceptron (Custom) Algorithm. Debuged GShare Algorithm. Ran Experiments for Benchmaking.
- Both authors contributed equally in paper writing.

B. *G-Share Predictor*

The g-share branch predictor is a dynamic branch predictor used in modern microprocessors. It leverages the history of recently executed branches to predict the outcome of the next branch instruction.

*1) **Data Structures**:*

```
int ghistoryBits
unsigned int ghistory
unsigned int ghistory_truncate
unsigned int* BHT_g
```

- `ghistoryBits` - The length of global history (most recent branch outcomes) to be used
- `ghistory` - Integer whose bit representation stores the actual global history
- `ghistory_truncate` - Helper integer initialized to $2^{\texttt{ghistoryBits}}$ - 1 to obtain the most recent ghistoryBits of history
- `BHT_g` - An array of length $2^{\texttt{ghistoryBits}}$ which stores the 2-bit predictions for all global history patterns

*2) **Working**:* In this we use a Global History Table that consists of 2-bit predictors as its entries. We XOR the `pc` and the `ghistory` (the bit representation of the Program Counter Index and the Global Branch History respectively), and extract its least significant `ghistoryBits` bits. It is extracted by performing a bitwise 'and' operation of the output of the XOR operation with `ghistory_truncate` (which is set to $2^{\texttt{ghistoryBits}}$-1). The final output is the index for the Global History Table. The value of the corresponding 2-bit predictor at that index determines the prediction for the branch - either taken (**T**) or not taken (**NT**). After the true outcome is known, the global predictor is trained, by updating the value of the 2-bit predictor at that index, and the Global History (`ghistory`) is updated.

*3) **Advantages**:* Gshare uses a global history register that combines information from multiple branches. It has a better chance of capturing complicated patterns in branch behavior over history independant techniques. This makes it effective for scenarios with intricate dependencies between branches. Gshare can be combined with other predictors (e.g., bimodal) to create hybrid predictors. Hybrid approaches leverage the strengths of different predictors for improved overall accuracy.

*4) **Disadvantages**:* Although GShare is an intuitive, yet powerful branch predictor, it has its shortcomings. It is inherently reliant on global correlations between branch outcomes of different brances. Its predictions will suffer if branching outcomes are more dependent on local patterns.

C. *Tournament Predictor*

The Tournament branch predictor is an advanced dynamic branch prediction technique that combines the strengths of multiple predictors to improve overall accuracy.

*1) **Data Structures**:*

```
// Same as GShare
int ghistoryBits
unsigned int ghistory
unsigned int ghistory_truncate
unsigned int* BHT_g

// To access local history
int lhistoryBits
unsigned int lhistory_truncate
unsigned int* BHT_l

int pcindexBits
unsigned int pc_truncate
unsigned int* PHT
unsigned int* chooser
```

- `lhistoryBits` - The length of local history used
- `BHT_l` - Array of length $2^{\texttt{lhistoryBits}}$ which stores the 2-bit predictions for all local history patterns
- `lhistory_truncate` - Helper integer initialized to $2^{\texttt{lhistoryBits}}$ - 1 to obtain the most recent lhistoryBits of history
- `pcindexBits` - Number of bits to store Program Counter Index
- `PHT` - Array of length $2^{\texttt{pcindexBits}}$ which stores local history of all program counter indices
- `pc_truncate` - Used to extract the the least significant `pcindexBits` from the input PC Index (`pc`)
- `chooser` - Array of size $2^{\texttt{ghistoryBits}}$ which stores 2-bit-based representations of whether to use local or global predictors

*2) **Working**:* The tournament predictor includes both a local predictor and a global predictor and a chooser which chooses which predictor's outcome should be taken based on

success history.
Local Predictor Working: We use the lower 'pcIndexBits' of the PC to index the Pattern history table. The value in that table gives us the 'localHistory' which we use to index the local History table. This table consists of 2-bit predictors for every possible 'localHistory'. The value in the 2-bit predictor is used to predict whether the branch is taken or not. After the true outcome is known, the local predictor is trained and 'localHistory' is updated.

Global Predictor Working: This is similar to G-share where the 'globalHistory' is used to index the global History Table which consists of 2-bit predictors for every possible 'globalHistory'. The value in the 2-bit predictor is used to predict whether the branch is taken or not. After the true outcome is known, the global predictor is trained and 'globalHistory' is updated.

Chooser: It consists of 2 bits. The chooser helps in choosing which predictor to use depending on the previous history of correct and wrong predictions made by each predictor. If both predictor guess correct or wrong, chooser remains unchanged, if global predictor makes a correct guess and local predictor makes a wrong guess then chooser value is incremented (ensuring maximum value is 3) and if local predictor makes a correct guess and global predictor makes a wrong guess then chooser value is decremented (ensuring minimum value is 0) If the chooser has value 00 then choose local predictor strongly, if the value is 01 then choose local predictor weakly, if value is 10 then choose global predictor weakly and if value is 11 then choose global predcitor strongly.

*3) **Advantages:*** The tournament predictor adapts to different branch behavioral patterns.By combining local and global predictors, it achieves better overall accuracy.
The choice predictor selects the most suitable predictor for each branch.

*4) **Disadvanatages:*** The tournament predictor combines local and global predictors. However, this fusion can lead to higher collisions in the pattern history table (PHT)
Implementing a tournament predictor requires additional hardware resources. The global history register, local history table, and choice logic add complexity to the design. (more than G-Share)
The choice between local and global predictors is not always straightforward. While tournament predictors aim for better overall accuracy, they may not outperform specialized bimodal predictors in certain scenarios.

*D. **Perceptron-based Predictor** (Custom)*

A perceptron branch predictor [1] is a machine learning-based dynamic branch predictor that uses a simple neural network model to predict branch outcomes in a CPU architecture. It calculates a weighted sum of features derived from the global history of branches, and if this sum exceeds a threshold, it predicts the branch will be taken.

*1) **Data Structures:***

```
uint8_t **perceptron;
int *global_history_feat;
int perceptron_output;
unsigned int perceptron_idx_truncate;
```

- `perceptron` - A 2D array of size ($2^{\texttt{ghistoryBits}}$, `ghistoryBits`+1), stores the weights and a bias of all perceptrons
- `global_history_feat` - A feature vector for global history (stores +1 if branch in history was taken else -1)
- `perceptron_output` - The perceptron output as shown in equation 1
- `perceptron_idx_truncate` - Helper integer initialized to $2^{\texttt{pcindexBits}}$ - 1 to obtain an index to choose a perceptron after XOR of `pc` with `ghistory`

*2) **Working:*** In our case, we have an array of multiple perceptrons. We first XOR the `pc` with the `ghistory` (the bit representation of the Program Counter Index and the Global History, respectively), to get an index, through which we select the perceptron for prediction from our list of perceptrons. The chosen perceptron maps a vector of input features of size `ghistoryBits` to one boolean output unit which represents whether or not a branch is taken. An element x_i of the input feature vector represents the value of the corresponding element in the global history register - which is either $+1$ (branch is taken) or -1 (branch is not taken). The final output for the perceptron can be given by:

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

Where, n = `ghistoryBits` and $x_i \in \{-1, +1\}$, represents whether or not the i^{th} branch in history was taken (+1) or not taken (-1). Therein, we predict branch is taken (**T**) if $y \geq 0$, else not taken (**NT**)
After the true outcome is known, we update the global history and the global history features. In addition, the weights are updated according to the following algorithm:

```
if sign($y$) or $|y| \leq \theta$ then:
    for $i := 0$ to $n$ do:
        if $|w_i| < \theta$ then:
            $w_i := w_i + t x_i$
        end if
    end for
end if
```

Here, t is the actual outcome, and θ is the a threshold on the weights (so the weights don't increase in magnitude beyond it). It is set to $1.93 \times$`ghistoryBits`$+14$ as in [1]

*3) **Memory Requirements:*** The memory (**in bits**) required by our perceptron can be given by the expression:

$$2^{\texttt{pcindexBits}} \times (\texttt{ghistoryBits} + 1) \times 8$$

This is because, for each perceptron, we have, weights equal to the `ghistoryBits` and 1 bias unit for each perceptron. Each weight is assumed to take up 8 bits in memory (the lowest possible you can go in C). Additionally, we use $2^{\texttt{pcindexBits}}$ perceptrons to make predictions, hence the expression follows.
To beat the gshare:13 and tournamnent:9:10:10 predictors on all traces, we use a perceptron with `pcindexBits` = 8 and ghistoryBits = 31. Thus the memory usage of our perceptron implementation is $2^8 * (31 + 1) * 8 = 65,536$ bits = 64 Kbits, which is well within the memory limits.

4) *Advantages:*

Perceptrons can handle longer branch histories, which improves prediction accuracy over traditional methods.
Robust to Aliasing: Due to the ability to use longer histories, perceptrons are less prone to aliasing, where different branches interfere with each other's predictions.
Perceptrons have been shown to improve misprediction rates on benchmarks compared to other predictors like gshare and tournament [1].

5) *Disadvantages:*

Implementing a perceptron predictor is more complex than traditional two-bit counters, requiring more sophisticated hardware.
Perceptrons require a training period to adjust their weights properly, which can lead to initial inaccuracies.
The time it takes to access the weight table and compute the weighted sum can introduce delays, affecting the predictor's speed. In addition, the perceptron has a higher overhead - to store the weights of the

III. Observation

A. *Gshare*

We found out the misprediction rate for various 'global History bits' as shown in table I. The misprediction rate for this model for various traces and configurations can be visualized in the figure 1.

Traces	5	10	13	15	20
fp_1	2.675	1.220	0.825	0.827	0.834
fp_2	19.592	6.131	1.678	0.985	1.149
int_1	35.574	22.024	13.839	11.220	8.172
int_2	1.975	0.734	0.420	0.364	0.320
mm_1	31.252	13.104	6.696	4.441	3.479
mm_2	24.094	13.314	10.138	8.039	5.875

TABLE I
G-SHARE MISPREDICTION RATE

B. *Tournament*

We found out the misprediction rate for various combination of 'global History bits' : 'local History bits': 'PC index bits' as shown in table II, III, IV. The misprediction rate for this model for various traces and configurations can be visualized in the figure 2 where we are varying only global History bits, figure 3 where we are varying only local History bits, figure 4 where we are varying only PC Index bits.

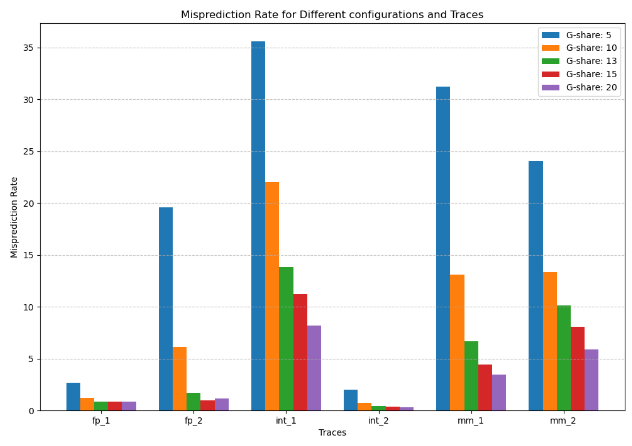

Fig. 1. Gshare Misprediction rate comparision

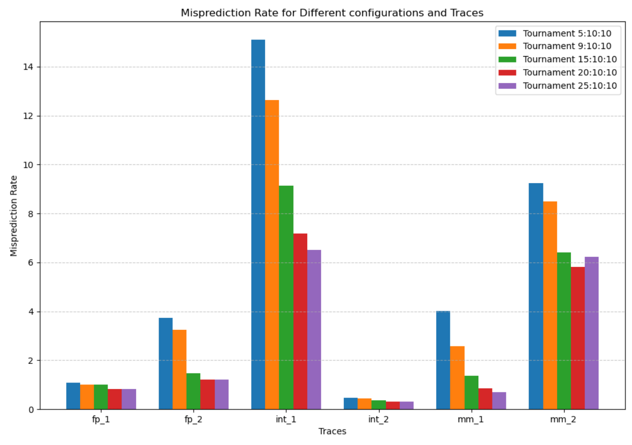

Fig. 2. Tournament Misprediction rate by varying Global History bits

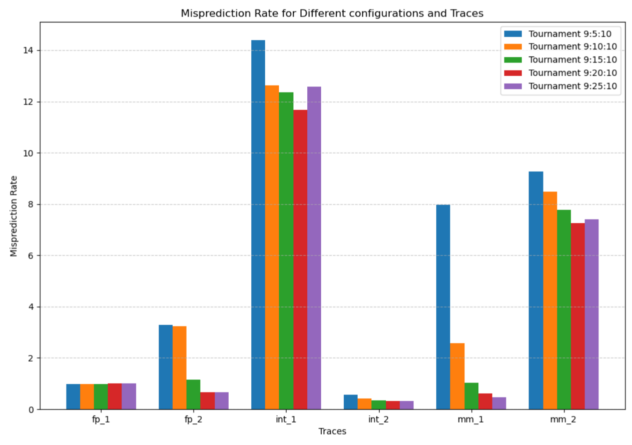

Fig. 3. Tournament Misprediction rate by varying local History bits

Traces	5:10:10	9:10:10	15:10:10	20:10:10	25:10:10
fp_1	1.091	0.991	0.994	0.827	0.834
fp_2	3.730	3.246	1.464	1.211	1.211
int_1	15.096	12.622	9.137	7.180	6.521
int_2	0.465	0.426	0.349	0.307	0.303
mm_1	4.018	2.581	1.355	0.838	0.701
mm_2	9.237	8.483	6.409	5.806	6.233

TABLE II
TOURNAMENT MIS-PREDICTION RATE BY VARYING GLOBAL HISTORY BITS

Traces	9:5:10	9:10:10	9:15:10	9:20:10	9:25:10
fp_1	0.992	0.991	0.992	0.993	0.993
fp_2	3.285	3.246	1.148	0.664	0.664
int_1	14.383	12.622	12.362	11.673	12.581
int_2	0.560	0.426	0.338	0.325	0.316
mm_1	7.977	2.581	1.028	0.614	0.463
mm_2	9.257	8.483	7.760	7.266	7.413

TABLE III
TOURNAMENT MIS-PREDICTION RATE BY VARYING LOCAL HISTORY BITS

Traces	9:10:5	9:10:10	9:10:15	9:10:20	9:10:25
fp_1	1.450	0.991	0.989	0.988	0.988
fp_2	4.262	3.246	2.069	2.069	2.069
int_1	19.654	12.622	11.457	11.457	11.457
int_2	0.715	0.426	0.377	0.377	0.377
mm_1	9.813	2.581	2.121	2.121	2.121
mm_2	13.081	8.483	6.830	6.719	6.714

TABLE IV
TOURNAMENT MIS-PREDICTION RATE BY VARYING PC INDEX BITS

Traces	8:8	16:8	31:8	64:7	128:6
fp_1	0.996	0.825	0.824	0.841	0.191
fp_2	5.132	0.982	1.016	0.282	0.229
int_1	10.138	9.131	8.208	8.668	9.589
int_2	0.411	0.327	0.298	0.297	0.324
mm_1	4.748	3.684	1.965	0.814	0.141
mm_2	20.141	10.260	7.586	9.364	11 830

TABLE V
PERCEPTRON MIS-PREDICTION RATE

C. *Perceptron*

We found out the misprediction rate for various combination of 'global History bits' : 'PC index bits' as shown in table V. The misprediction rate for this model for various traces and configurations can be visualized in the figure 5.

Now, we compare Gshare 13, Tournament 9:10:10 and Perceptron 31:8 accross all traces as shown in Figure 6 and Table III-C.

IV. ANALYSIS

A. *Gshare*

We can see in Figure 1 that for each trace, in general, the misprediction rate decreases as Global History bits increases. This is because as global history bits are increasing more temporal locality patterns can be captured and hence better performance.
Adding more global history bits in Gshare helps the computer better guess what the program will do next by looking at a longer history of past decisions. This reduces mistakes because the computer can understand more complicated patterns and can tell the difference between different branches more easily. So, with more history bits, the computer makes fewer wrong guesses about what the program will do, making it better at predicting overall.

B. *Tournament*

As shown in figure 2; when you increase the number of global history bits while keeping local history bits and PC index bits fixed, the predictor can remember more past behaviors, which helps it make better predictions for branches that share similar historical patterns. Consequently, the misprediction rate decreases as the predictor becomes more accurate in foreseeing the outcomes of these branches. However, there might be a point of diminishing returns where increasing the global history bits further doesn't significantly improve prediction accuracy, and other factors like local history bits and PC index bits start to play a more significant role in

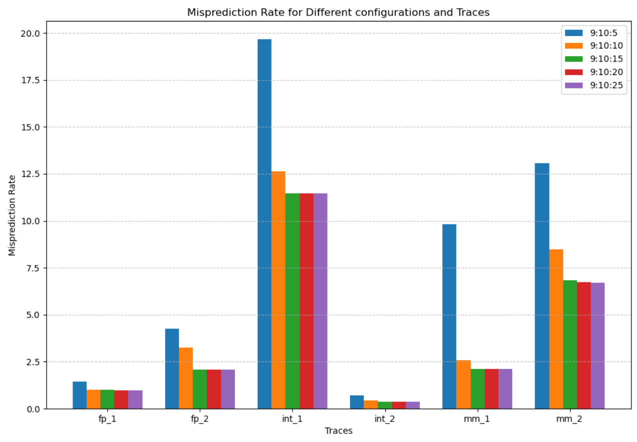

Fig. 4. Tournament Misprediction rate by varying PC Index bits

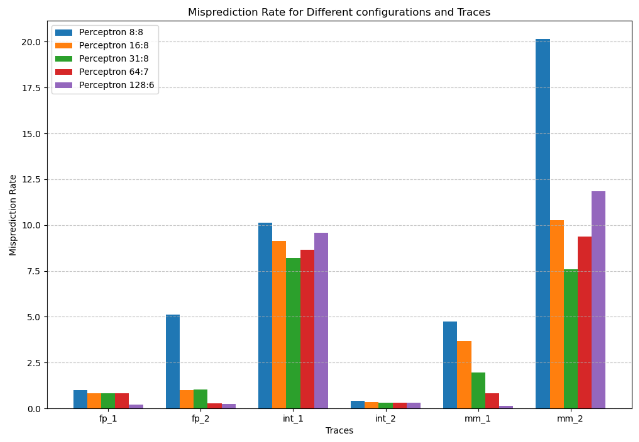

Fig. 5. Perceptron Misprediction rate comparision

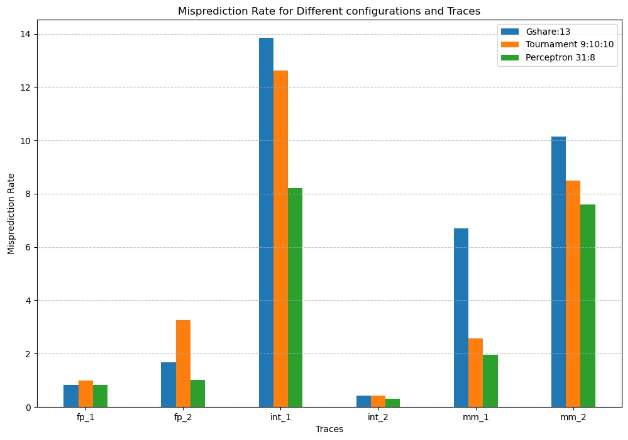

Fig. 6. All three models comparision

Traces	GShare:13	Tournament 9:10:10	Perceptron 31:8
fp_1	0.825	0.991	0.824
fp_2	1.678	3.246	1.016
int_1	13.839	12.622	8.208
int_2	0.420	0.426	0.298
mm_1	6.696	2.581	1.965
mm_2	10.138	8.483	7.586

TABLE VI
COMPARING THE THREE MODELS

determining the misprediction rates accordingly within the tournament framework.
As shown in figure 3, when you vary the number of local history bits in a tournament predictor while keeping global history bits and PC index bits fixed, you're essentially adjusting how much historical information is stored for individual branches. Increasing local history bits allows the predictor to remember more specific patterns of behavior for each branch. This leads to better predictions for branches with distinct, localized patterns, potentially reducing their misprediction rates. However, excessive local history may increase conflicts and overhead, impacting prediction accuracy negatively. Conversely, decreasing local history bits may result in oversimplified predictions, leading to higher misprediction rates, particularly for branches with intricate, branch-specific behaviors. Hence we see that the misprediction rate first decreases and then increases as we increase the local history bits keeping rest constant.
As shown in figure 4, increasing the number of PC index bits allows for finer granularity in indexing the prediction tables, which can lead to improved prediction accuracy for branches with distinct program counter values. However, there's a finite address space for indexing these tables. As you continue to increase the number of PC index bits, you eventually reach a point where the address space is fully utilized, and adding more bits doesn't provide any additional benefit. This point is where saturation occurs.

C. *Perceptron*

As shown in figure 5, as we increase the number of bits allocated to capture global branch history (keeping PC Index bits constant), our misprediction rate decreases. This is expected becauses as we increase the number of global history bits, we are able to condition the prediction for next branch on a longer history. This allows the perceptron to capture more complex and long range patterns in branch prediction.
The latter three perceptron configurations i.e 31:8, 64:7 and 128:6 require the same amount of memory for their implementation according to the formulation mentioned in section II-D. Changing the number of global history bits changes the length of branch outcome history we give to the perceptron to predict the next outcome. whereas, changing the PC Index Bits changes the overall number of perceptrons we train for predicting the outcomes. Essentially, keeping the memory fixed, there is a trade-off between the amount of history to draw correlations from (global history bits) and how we draw correlations from the history using different perceptrons (PC Index bits). The optimal values for both these parameters are heavily dependent on the data we are working with. As shown in figure 6, our model perceptron:31:8 (`ghistoryBits` = 31 and `pcindexBits` = 8) beats gshare:13 and tournament:9:10:10 on all the traces.

V. RESULT AND CONCLUSION

As we can see in Figure 6 perceptron 31:8 beats Gshare 13 and Tournament 9:10:10 in all the traces. We observe that for fp_1, fp_2 and int_2 gshare performs better than Tournament and Tournament performs better than Gshare for the rest of the traces. Traces with limited diversity in branch behavior or with stronger global correlations may favor GShare predictors due to their specialization in global history prediction, resulting in better overall performance in such cases, hence this could be a reason for Gshare 13 performing better than tournament 9:10:10 in those traces.
We observe that perceptron 31:8 outperforms Gshare 13 and Tournament 9:10:10 in all traces. Perceptron predictors often outperform GShare and Tournament due to their neural network-based approach, which captures complex and non-linear patterns in branch behavior. Unlike GShare's global history or Tournament's combination of predictors, perceptrons adapt dynamically, effectively modeling dependencies over long distances. Their ability to learn from historical branch outcomes and adjust weights allows them to accurately predict branches with irregular patterns.

VI. ACKNOWLEDGEMENTS

We are thankful to Prof. Jishen Zhao and the TAs of CSE240A for designing such an eventful learning experience for us in the form of this project and also helping us round the clock with all our queries.

REFERENCES

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

[2] R. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[3] S. McFarling, "Combining branch predictors," Western Research Laboratory, Tech. Rep., 1993.

[4] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 118–130.

[5] J. Zhao, "Principles in computer architecture [branch predictor discussion session video]. cse240a," Winter 2024. [Online]. Available: https://sites.google.com/ucsd.edu/cse240a-w24/

[6] J. Zhao, "Principles in computer architecture [lecture slides]. cse240a," Winter 2024. [Online]. Available: https://sites.google.com/ucsd.edu/cse240a-w24/

[7] A. Zouzias, K. Kalaitzidis, and B. Grot, "Branch prediction as a reinforcement learning problem: Why, how and case studies," 2021.