Branch Predictor

Soham Satyadharma* and Vaibhav Kant Tiwari*

1. INTRODUCTION

Branch prediction is an important problem in computer architecture, especially while designing a pipelined processor. A branch predictor improves the flow in a pipeline. We can use a simple example to understand the need for a branch predictor. Let us consider a 5 stage pipeline diagram with stages as F (fetch), D (decode), X(execute), M(memory), W(writeback). Assuming the first instruction is a branch instruction, branches are resolved in the D stage and XF forwarding is available, the pipeline needs to be stalled for 1 cycle as shown in Table 1. This is because the address of the target instruction can only be fetched after the branch instruction is resolved in the D stage in cycle 2.

Instruction	1	2	3	4	5	6	7
Branch Instruction	F	D	X	M	W		
Target instruction			F	D	X	M	W

Table 1: Without branch prediction, we have to stall the next instruction till the branch instruction is resolved.

For every branch instruction, we need to stall for 1 cycle. This increases the latency of the program unnecessarily. To avoid this, we can predict the target of a branch instruction and fetch it to continue our execution without stalling. If the the branch target is the same as the predicted instruction, then the pipeline continues normally. Otherwise, the correct target instruction is fetched when the branch is resolved and the incorrectly predicted instruction is flushed from the pipeline as shown in Table 2.

Instruction	1	2	3	4	5	6	7
Branch Instruction	F	D	X	M	W		
Predicted Instruction		~~F~~					
Target Instruction			F	D	X	M	W

Table 2: Branch prediction allows us to predict the result of the branch instruction before it is resolved. If the predicted instruction is incorrect, we flush it from the pipeline.

If done correctly, branch prediction removes the need to stall the pipeline until the branch is resolved. If the predicted instruction is the target instruction, the instructions are executed without any stalls. However, incorrect branch prediction affects the performance of a pipeline. As we can see in the above example, a wrong prediction effectively means that there will at least be the same number of stalls as there would be without a branch predictor.

Branch predictors can be either static or dynamic. Static branch predictors predict branches at compile time, which means they do not adapt to program behaviors. Dynamic branch predictors predict a branch as taken or not taken by looking at the history of previous branches from a branch history table (BHT). The earliest dynamic branch predictors were 1 bit branch predictors. They checked if a branch was taken the last time it was executed. If it was taken the last time, they predicted taken, otherwise they predcited not taken. Current branch predictors have improved a lot and achieve >99% accuracy on 99% of static branches on benchmarks like SPEC 2017 [1], [2]. CNN methods proposed by Tarsa et al [3] and Zangeneh et al [4] claim to further improve branch prediction by reducing the misprediction rates on hard to predict branches.

In our project, we first implemented two traditional branch predictors - GShare [5] and Tournament [6]. Then, we implemented a custom branch predictor based on the Perceptron branch predictor [7] to beat their best implementations. In the following section we will describe the evolution, workings and our implementation of these predictors.

2. IMPLEMENTATION

2.1 Gshare branch predictor

Gshare is an improvement on bimodal branch predictors. A bimodal branch predictor does not take local or global branch history into account and just tries to be resilient against unusual changes in outcomes by using a 2- bit saturating predictor, indexed by program counter (PC). Yeh and Patt [8] improved this by using global history to make a prediction. The direction of most recent n conditional branches are recorded in a single shift register. This tackled of patterns where direction taken by a branch depends on other recent branches.

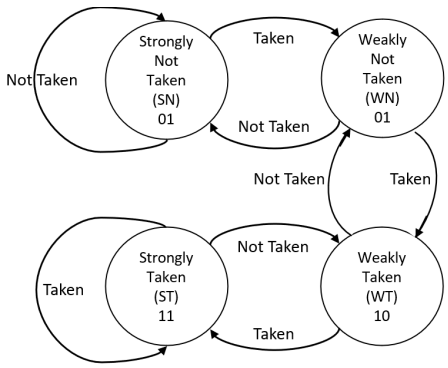

Figure 1: State diagram of 2-bit predictor. A 2-bit predictor adds hysteresis to branch prediction so that the prediction does not change on a single different outcome.

Pan et. al [9] further improved upon this by using a concatenation of global history and branch address bits to index the predictor table. This method is known as gselect and it significantly outperforms simple global prediction since branch address bits more efficiently identify branches.

However, there can be a lot of redundancy in the counter index used by gselect. If there are enough address bits to identify a branch, we can expect the frequent global history combinations to be rather sparse. Hence, we can hash the global history and the branch address using XOR, so that the hash has more information than either component alone. This strategy is known as gshare and it is one of the branch predictors that we implement. Consider the 4/4 gselect vs 8/8 gshare strategy comparison referenced in figure 2.

Branch Address	Global History	gselect 4/4	gshare 8/8
00000000	00000001	00000001	00000001
00000000	00000000	00000000	00000000
11111111	00000000	11110000	11111111
11111111	10000000	11110000	01111111

Figure 2: Gshare can more effectively differentiate between all 4 cases compared to gselect. Figure taken from [5]

It is evident that gshare can separate all four cases above since the gselect predictor cannot take advantage of the distinguishing history in the upper four bits. For smaller predictor sizes, gshare outperforms 2-level correlation predictors (Figure 4) and has comparable performance for larger predictor size. Since, gshare only requires a single array access, it has less delay and is easier to pipeline than a tournament predictor (Figure 6).

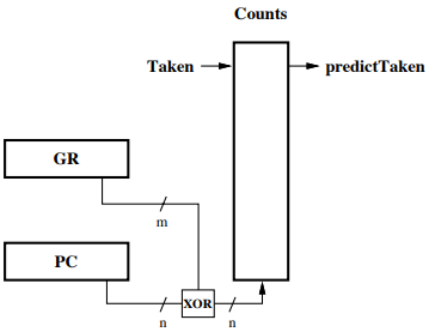

Figure 3: Global history predictor with index sharing (gshare). GR and PC registers are XORed together to index the counter table. Figure taken from [5].

Implementation Details

Initialization: In our implementation of the gshare predictor we take the global history bits *(ghistoryBits)* from the user and set the size of a gshare branch history table *(gshare_BHT)* as an array of $2^{(ghistoryBits)}$ elements. We also maintain *global_history* to index into the *gshare_BHT*.

Training: We update the saturating counter according to the actual outcome of the branch and update the *global_history* with the latest outcome.

Prediction: We XOR the *global_history* with *pc* and take the last *ghistoryBits* to index into the *gshare_BHT* table and predict the outcome. We predict NOTTAKEN if the counter is 00/01 (SN/WN), otherwise we predict TAKEN as shown in Figure 1.

2.2 Tournament branch predictor

Tournament branch predictor is a combination of a local branch predictor and a global branch predictor. The local branch predictor [8] takes into account the history of branch patterns and tries to predict outcomes according to each individual local history pattern by using a saturating bimodal branch predictor for each potential pattern. This predictor consists of a history table, each of whose entry records the direction taken by the most recent *n* branches whose addresses map to this entry, where *n* is the length of the entry in bits. The second table is an array of *m* bit saturating counters as used in a bimodal predictor. Thus, it is also referred to as local or *(m, n)* correlation branch predictor.

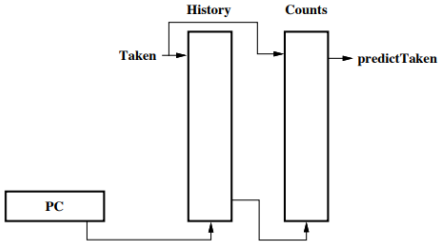

Figure 4: (m, n) Correlation branch predictor structure where first table holds branch pattern histories and second table maintains prediction counts. Figure taken from [5]

We can combine 2 different types of predictors in a single scheme and choose between the optimal predictor using a selector. Let the combined predictor contain two predictors P1 & P2. This combined predictor will also contain additional counter array, which is a 2-bit saturating counter, which serves to select the best predictor to use by keeping track of the more accurate predictor. If P1c and P2c represent whether predictors P1 and P2 are correct, then the selector is updated by P1c-P2c as shown in figure 5.

P1c	P2c	P1c-P2c	
0	0	0	(no change)
0	1	-1	(decrement counter)
1	0	1	(increment counter)
1	1	0	(no change)

Figure 5: Predictor selection scheme according to which entries of selector table are updated. Figure taken from [5]

One such combined predictor was used in Alpha 21264 which uses local history and global history to predict future branch directions since branches exhibit both local correlation and global correlation. This predictor, shown in figure 6, is referenced as the tournament branch predictor. [9,]

One of the major advantages of the tournament predictor is its ability to leverage both local and global history information and thus easily outperforms both independent types of

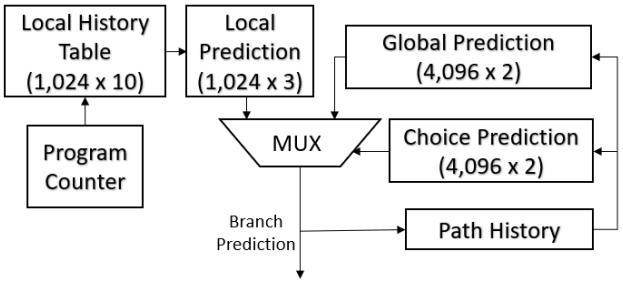

Figure 6: Tournament branch predictor structure as used in Alpha 21264 processor includes a selector choosing between predictions of global and local branch predictors

predictors. But, the two level array access increases latency compared to gshare predictor and thus is harder to pipeline.

Implementation Details

Initialization: In our implementation of the tournament predictor we take global history bits *(ghistoryBits)*, local history bits *(lhistoryBits)* and program counter (PC) index bits *(pcIndexBits)* from the user to set sizes of global branch history table *(global_BHT)*, local pattern history table *(local_PHT)*, and local branch history table *(local_BHT)* respectively. We also maintain a *selector* table of same size as the *global_BHT*. A *gshare_BHT_index* is also maintained to keep track of global history.

Training: We first update the *selector* table entry, referenced by *ghistoryBits* of *pc*, according to the correctness of the two predictors. The next step involves updating the predictors themselves. The *local_BHT* and *global_BHT* are updated according to the difference in their prediction and the actual outcome and the two bit saturating counters are updated. Then, the local and global histories are modified to note the latest outcome.

Prediction: From the PC *(pc)*, *phistoryBits* are used to index into *local_PHT* and *ghistoryBits* are used to index into *global_BHT* table and global prediction, *global_outcome*, is noted. From *local_PHT*, the pattern history value is used to index into *local_BHT* and local prediction *local_outcome* is noted. *ghistoryBits* of *pc* are also used to index into *selector* table and accordingly *local_outcome* or *global_outcome* is chosen. If the entry in *selector* table is 0 or 1, we choose *global_outcome* otherwise *local_outcome*

2.3 Perceptron branch predictor

So far, we have explored the use of history tables of 2 bit saturating counters for branch prediction. The main drawback of these methods is that the length of the shift register is restricted to the logarithm of the number of counters. To improve branch prediction using these architectures, we need to increase the length of the shift register to accommodate more complex branch history.

We use a machine learning solution to improve the aforementioned branch predictors. Machine learning has been successfully applied to branch prediction and using methods like recurrent neural networks [10] and convolutional neural networks [3]. In our project, we use the simplest machine learning architecture - the single layer perceptron - to design a dynamic branch predictor following the method of Jiménez et al [7]. Before the perceptron branch predictor was introduced, neural networks had been used to perform static branch prediction. [11]

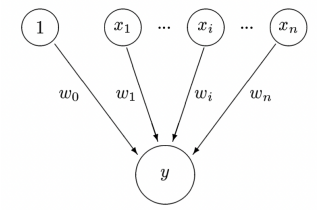

Figure 7: A perceptron takes *n* bits from the global history register as input and computes their weighted sum. Figure taken from [7]

A single layer perceptron contains one artificial neuron connected to several input units through weighted edges to one output unit. A perceptron branch predictor takes n bits of global history shift register $\{x_1, x_2, ..., x_n\}$ as input and learns a function $f(x_1, x_2, ..., x_n)$ to compute the branch prediction using learnt weights for every input bit. The output of the perceptron can be calculated according to Equation 1. The diagram of a single perceptron is shown in Figure 7.

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

In Equation 1, w_0 is the bias term. Each x_i is either 1 or -1. If $y \geq 0$, then branch is predicted to be taken, otherwise it is predicted to be not taken. We also use θ as threshold to decide when to train the perceptron. If t is the target prediction and y is output predicted by the perceptron, then the training algorithm is given in Algorithm 1. Finally, number of perceptrons are combined into a perceptron table indexed by a hash of the global history register as shown in Figure 8.

if $sign(y) \neq t$ *or* $|y| \leq \theta$ **then**
  for $i \leftarrow 1$ **to** n **do**
    $w_i \leftarrow w_i + t x_i$
  end
end

Algorithm 1: Perceptron training algorithm

It is easy to implement dynamic perceptron branch predictor in hardware unlike other machine learning models like convolutional neural networks, which have enormously large training times and we have to use offline training for them on large datasets. Besides this, a perceptron branch predictor is more intuitive unlike other machine learning models. From Algorithm 1, we can see that a perceptron makes a keeps track of correlations between global history and the branch being predicted. The algorithm increases the w_i when the corresponding branch is correctly predicted and decreases it when it is not. In this way, the perceptron increases the weight when there is a positive correlation and penalizes it when there is a negative correlation.

A major drawback of a perceptron predictor is its inability to handle non linear relationships between branches. From Equation 1, we can say that a perceptron divides an n dimensional space into two by predicting taken or not taken. Hence, a perceptron cannot capture non linear relationships between global history bits and the branch to be predicted. To handle all such cases, we need to use more complex models or introduce non-linear activation functions like ReLU [12] in a model.

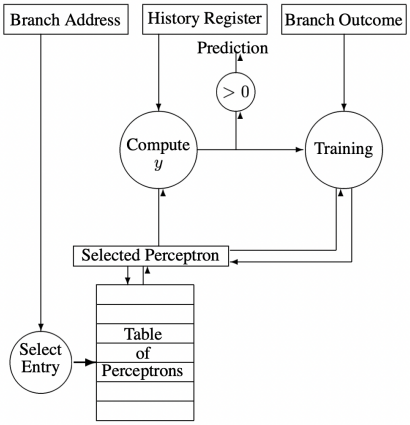

Figure 8: A hash of the branch PC is used to select a perceptron and the prediction is computed. The perceptron is trained using Algorithm 1 and written back to the table. Figure taken from [7]

Implementation Details

We fit the perceptron predictor to the 64K bit budget. We used a perceptron table of length 256, with each perceptron having 31 weights in addition to one bias term. We restricted the value of the bits between -128 and 127 to ensure that each weight does not exceed 8 bits. We arrived at this final design after conducting experiments detailed in the next section.

Initialization: We initialize each element of the perceptron table of *256x31* and the bias table of *256x1* to 0. We took the threshold θ as 32.

Training: We calculate the index of the perceptron table by using the last 256 bits of the PC. Using that index, we compute the perceptron sum as shown in Equation 1. Using the outcome of computed sum, we update both the perceptron and bias tables according to Algorithm 1, making sure that the weights remain between -128 and 127.

Prediction: We calculate the index of the perceptron and its sum similar to the method used in training. If the computed sum is greater than 0, we predict TAKEN, otherwise we predict NOT TAKEN.

3. EXPERIMENTS AND OBSERVATIONS

We were provided with 6 instruction traces, which include integer(INT), floating point (FP) and multimedia (MM) workloads, with instruction counts varying from 1.5M to 9.7M per trace. We experimented with various designs of gshare, tournament and perceptron predictors on these traces. Table 3 shows more details about these traces.

Traces	Branches	NOTTAKEN	TAKEN	TAKEN%
int_1	3771697	1664686	2107011	55.86
int_2	9755315	206849	3548466	36.37
fp_1	1546797	187589	1359208	87.87
fp_2	2422049	1025735	1396314	57.65
mm_1	3014850	1518079	1496771	49.64
mm_2	2563897	94796	1614101	62.95

Table 3: Statistics of traces used

3.1 Gshare and Tournament predictors

We firstly identify the best performing versions of the gshare and tournament predictors by varying their hardware budget. Since, we are allowed to only use a maximum of 13 global history bits for gshare and 9 bits, 10 bits and 10 bits as global history bits, local history bits and PC index bits respectively for tournament predictor, we limit our experiments within the specified range.

As expected, increasing the hardware budget for these predictors generally leads to improved performance of the predictors. This can be observed from performance charts of gshare and tournament predictors in Figures 9 and 10 respectively. From these observations we establish gshare:13 and tournament:9:10:10 as our predictors to benchmark our custom predictor against. Gshare:a refers to a gshare predictor of a global history bits while tournament:a:b:c refers to a tournament predictor with a global history bits, b local history bits and c PC index bits.

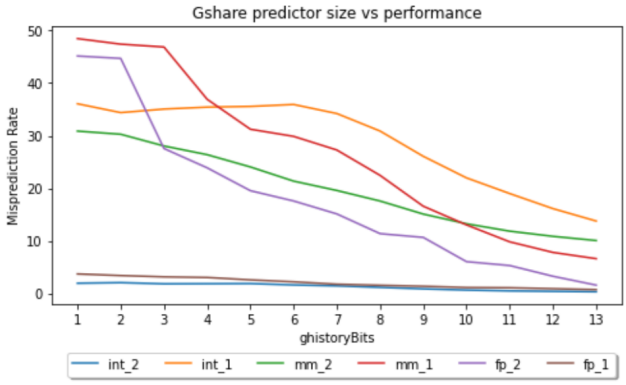

Figure 9: Gshare predictor size vs Performance. The performance improves on increasing the global history bits.

3.2 Perceptron predictor

We experimented with different perceptron designs within the 64Kbit budget by changing the values of perceptron table length, weights in a perceptron and the maximum bits allowed for a single weight. We tried the following combinations - 8:5:3, 8:6:2, 8:4:4, 7:5:4, 7:6:3. Here a perceptron predictor a:b:c refers to 2^a as perceptron table length, 2^b weights in a single perceptron and 2^c bits for the value of each weight. Note that $(a+b+c) = 16$. This means a total of 2^{16} bits or 64 Kbits are used.

Though there was not a clear trend in the misprediction rates for the different designs, the 8:5:3 design gave us the best results on all 6 traces. Intuitively, we can say that it performs better than the 7:5:4 and 7:6:3 designs as it uses double

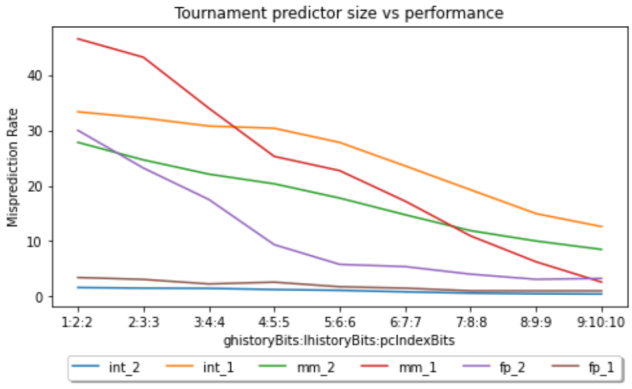

Figure 10: Tournament predictor size vs Performance. a:b:c on the X axis refers to *a* global history bits, *b* local history bits and *c* PC index bits.

the global history bits and is able to glean more information from the global history shift register. Similarly, we can speculate that the 8:6:2 has too few bits per weight to make a prediction and the 8:4:4 has too few weights in a perceptron to make better predictions. The results are graphically represented in Figure 11.

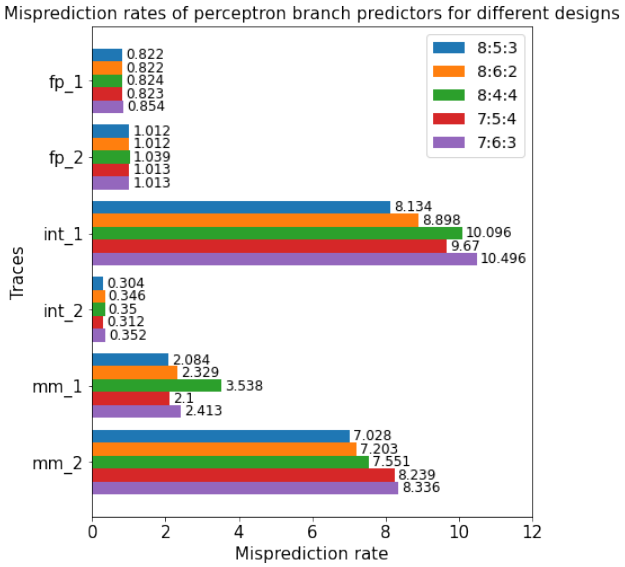

Figure 11: Comparison of perceptron different designs of perceptron branch predictors. A design of a:b:c means a perceptron table length of 2^a, 2^b weights in a single perceptron and 2^c bits for each weight value. The 8:5:3 design performs best on all traces.

After finalizing the 8:5:3 design, we also experimented with varying values of the threshold θ. The value of θ decides how often the branch predictor is trained. It is evident from Algorithm 1 that greater the value of θ, more is the predictor trained. We chose θ from the set $\mathbf{\Theta} = \{0, 16, 32, 48, 64, 80, 96, 112, 128\}$. The smallest value is 0 as θ is compared to the absolute value of the perceptron sum. We stopped at 128 as the predictor was performing worse on 4 out of the 6 traces - *fp_1*, *fp_2*, *int_2*, *mm_2* - with increasing values of θ.

We have shown the results for the misprediction rates for θ = 0, 32, 128 in Figure 12. θ = 0 expectedly gives the worst results as the predictor is without a threshold and it is trained only when the outcome is incorrect. From our set $\mathbf{\Theta}$, we got the best overall results for θ = 32. Increasing θ after 32 makes the predictor perform worse on 4 of the 6 traces, as mentioned earlier. This is expected because like other machine learning algorithms, the perceptron too does not get better by simply training it more. There is a point after which the results get worse due to overfitting. The same phenomenon will also happen with *int_1* and *mm_1* traces, but at a value of θ greater than 128.

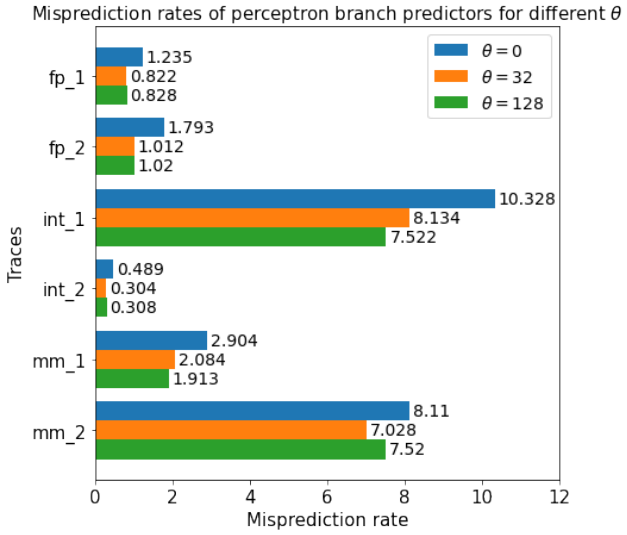

Figure 12: Comparison of perceptron branch predictors on 3 values of θ - 0, 32, 128 with the 8:5:3 design. We got the best overall results for θ = 32.

4. RESULTS

Based on our observations, we chose a 8:5:3 design for our custom perceptron predictor with θ = 32. We compared our custom predictor with gshare:13 and tournament 9:10:10 predictors, the best performing gshare and tournament implementations respectively. Our perceptron predictor beats both the other predictors in all the six traces provided to us. The results are tabulated in Table 4 and graphically shown in Figure 13.

Traces	Gshare	Tournament	Perceptron
fp_1	0.825	0.991	0.822
fp_2	1.678	3.246	1.012
int_1	13.839	12.622	8.134
int_2	0.420	0.426	0.304
mm_1	6.696	2.581	2.084
mm_2	10.138	8.483	7.028

Table 4: Comparison of the perceptron predictor with gshare:13 and tournament:9:10:10 predictors. The perceptron predictor beat both of them in all six traces.

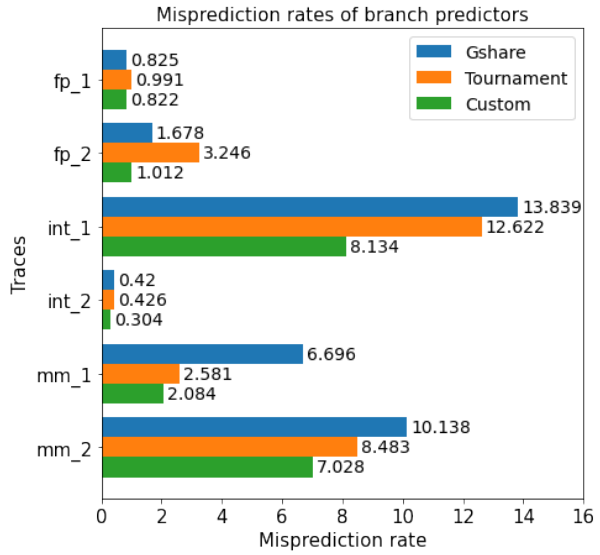

Figure 13: Graphical representation of the results shown in Table 4

On average, we are able to achieve a 34.86% of performance improvement over our gshare implementation and a 31.21% performance improvement over our tournament branch predictor implementation. Table 5 details the performance improvement achieved by our custom branch predictor as a percentage for every trace. This is intuitive because unlike gshare and tournament predictors, a perceptron predictor can learn to recognize the global history bits that are more important for prediction compared to other bits by assigning them greater weights and ignore the unimportant ones by assigning them lower weights. Moreover, as the weights in the perceptron vary from -128 to 127, the perceptron predictor adds much more hysteresis to branch prediction compared to the gshare and tournament predictors that use 2 bit counters.

Predictor	fp_1	fp_2	int_1	int_2	mm_1	mm_2
gshare	0.36	39.69	41.94	27.61	68.87	30.67
Tournament	17.05	68.82	36.34	28.63	19.25	17.15

Table 5: Percentage improvement of our perceptron predictor over gshare:13 and tournament:9:10:10 predictors for each trace.

5. CONCLUSION

In this project we studied and explored the intricacies of traditional branch predictors starting from various types of global predictors, including gselect and gshare, followed by correlating predictors and a combination of global and correlation predictors, called the tournament predictor. Implementing the gshare and tournament predictors allowed insights into their working which we used to beat them by building our custom branch predictor based on the perceptron branch predictor. After fine tuning our design we were able to achieve an average performance improvement of 34.86% and 31.21% over our implementation of gshare and tournament predictors respectively.

6. INDIVIDUAL CONTRIBUTION

This was a group project and both members contributed equally in the project. Having said that, there are a few tasks that I explicitly focused on. I am summarizing it as follows:

- Conducted literature survey regarding implementation of potential custom predictors.
- Implemented the perceptron branch predictor as the custom predictor.
- Explored hardware budget optimizations for perceptron predictor within the 64Kbit threshold and analyzed the same to incorporate in the final perceptron predictor design. The graphs were made using matplotlib. [13]
- Explored θ optimizations for perceptron predictor and analyzed the same to incorporate in the final perceptron predictor design. The graphs were made using matplotlib. [13]
- Wrote the Introduction, Implementation (Perceptron branch predictor), Experiments and Observations (Perceptron predictor), Results sections of the report and reviewed the other sections.

7. REFERENCES

[1] A. Seznec, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[2] C.-K. Lin and S. J. Tarsa, "Branch prediction is not a solved problem: Measurements, opportunities, and future directions," *arXiv preprint arXiv:1906.08170*, 2019.

[3] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, "Improving branch prediction by modeling global history with convolutional neural networks," *arXiv preprint arXiv:1906.09889*, 2019.

[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)*, pp. 118–130, IEEE, 2020.

[5] S. McFarling, "Combining branch predictors," tech. rep., Citeseer, 1993.

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

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

[8] T. Yeh and Y. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," *Proc. 20th Annual Intl. Symp. on Computer Architecture*, 1993.

[9] S. Pan, K. So, and J. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, 1992.

[10] A. Smith, "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.

[11] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 1, pp. 188–222, 1997.

[12] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[13] `https://matplotlib.org/`.