Branch Prediction Project

Chenhao Liu
PID: A59018246

Xinhao Li
PID: A59018256

1. Introduction

In computer architecture, branch prediction is a technology that predicts which branch will be executed before the execution of the branch instruction end to improve the performance of the processor's instruction pipeline. Branch prediction is critical for achieving high performance in modern microprocessors. Since the branch instruction is executed according to the result, the front-stage circuit of the pipeline is in a stall for the branch instruction, which will inevitably waste clock cycles. If the processor can predict whether the branch will transfer before the result of the previous instruction comes out, then the corresponding instruction can be executed in advance, thus avoiding the stall of the pipeline, and improving the computing speed of the processor. The advantages and disadvantages of branch prediction are obvious: If the prediction is correct, the processor efficiency will be improved as it no longer needs to stall before the real target address is determined; if the prediction is wrong, more cost will be induced due to pipeline flushing. Therefore, prediction accuracy is important for a branch predictor.

In this project, we implemented three branch predictors under 64K + 256bits memory usage by utilizing C, the following is what we did in the project:

(1) Implement *static*, *g-share*, and *tournament* branch predictors as baseline predictors;

(2) Analyze the perceptron algorithm and implement *perceptron* as our *custom* predictor;

(3) Conduct experiments for three branch predictors on six test traces and analyze their performance with different hyperparameters.

Branch prediction evolvement

With the constantly growing demand for computer with high speed, efficiency, and flexibility, branch predictors have developed rapidly in the past 50 years. People have invented different branch predictors. Initially, predictors used static branch prediction technology. They are very simple and only need a few system resources. The static branch predictor developed from only predicting one direction to a backward strategy where the branch is assumed taken (not taken) when the branch target is less (greater) than the current value of the PC. At that time, Some ISAs allow for a compiler interface through which Branch hints can be made. If such is the case, the compiler can make use of these hints by inserting the most likely outcome of the branches based on high-level information about the structure of the program. This is called program-based prediction [1]. Later, people found that there was a bottleneck in the accuracy of static prediction, which could not meet people's needs, so dynamic branch prediction technology was invented. Dynamic branch prediction algorithms take advantage of the run time information available in the processor and can react to the changing branch patterns. The benefit of dynamic branch prediction is that performance enhancements can be realized without profiling and existing binaries. There are many strategies, such as two-level prediction [11], which employs two separate levels of branch history information to make the branch prediction. Smith's Algorithm [12] consists of a table that keeps a record for each branch whether each or not the previous branches were taken or not so that the next time when it encounters the same branch it makes a better judgment. Stark and Jared [13] invented Variable Path Length Predictors. Juan and Toni [5] proposed Dynamic History Length Filtering Predictors. Wechsler and Ofri [16] proposed Loop Counting Predictors which are specifically for the iteration part of applications. Now people find that a program may be related to multiple different types of history, so some branch predictors start to use hybrid branch prediction [8] such as tournament that employs two or more single scheme branch prediction algorithms and combines these multiple predictions together to make one final prediction.

2. Implementation

Saturating counters

Our *gshare* and *tournament* branch predictors use two-bit saturating counters. A two-bit consists of four states: Strongly Not Taken(00), Weakly Not Take(01), Weakly Taken(10), and Strongly Taken(11), as shown in figure 1.

When a branch command is evaluated, the corresponding counter is updated. If the branch is not taken, the counter

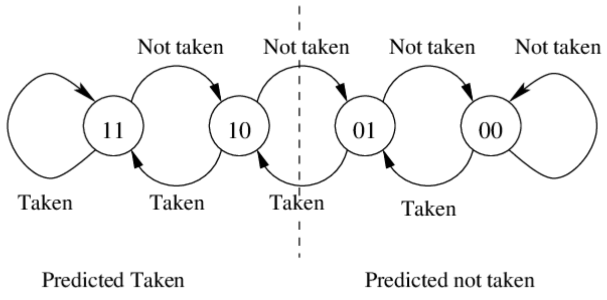


Figure 1. Two-bit saturating counter.

will be lowered in the direction of SN; if the branch is taken, the counter will be increased in the direction of ST. The advantage of this approach is that the conditional branch instruction must take a branch twice to flip from a strong state, thereby changing the predicted results.

2.1. gshare

Gshare is a kind of global branch predictor. Global branch predictors do not keeps a separate history record. Instead they keeps a shared history of all branches. Global branch prediction is used in AMD processors, and in Intel Pentium M [10], Core, Core 2, and Silvermont-based Atom [9] processors.

Gshare uses an m-bit global history register to keep track of the direction of the last m-executed branches. To simplify the implementation, when the instructions come to the branch predictor, this global history register is XORed with the PC to create an index into a 2m-entry pattern history table of n-bit counters. The result of this index is the prediction for the current branch. The predictor then compares this prediction with the real branch direction to determine if the branch was correctly predicted or not and updates the prediction statistics. The predictor then updates the n-bit counter used to perform the prediction. Finally, the branch outcome is shifted into the most significant bit of the global history register.The *gshare* predictor structure is shown in figure 2. The advantage is that dependencies between different branch instructions can be identified. The disadvantage is that the history is influenced by the execution of different irrelevant branch instructions.This approach is effective when the history buffer is long enough. However, the number of bits in the history table is an exponential order of magnitude in the number of bits in the counters. So it have to share this large pattern history table among all branch instructions which takes up a lot of space.

2.2. Tournament

Tournament is a hybrid predictor implemented on the Alpha 21264 microprocessor [6] in 1998, combining the local predictor and correlated predictor. It is similar to G-share, but it contains three pattern history tables including the local pattern history table, the global pattern history table, and the choice pattern history table used by the chooser which assigns branches to one predictor or the other. When the PC comes to the predictor, we calculate both local predictions and global predictions. Unlike global prediction calculating indexes through XOR manipulation, the local index is equal to the last few digits of the PC. After that, the chooser will choose local prediction or global prediction as the result. The predictor then compares this prediction with the real branch direction to determine if the branch was correctly predicted or not and updates the prediction statistics. No matter which prediction the chooser chooses, both the local and global predictors will be updated. At the same time, the counter of choice pattern history table will be updated in the direction of the correct predictor as well. Finally, the branch outcome is shifted into the most significant bit of the global history register. The *tournament* prediction process is shown in figure 3.

The advantage of the tournament predictor is that it integrates global and local historical information which improves the accuracy of the whole predictor, and overcomes the problem of mutual interference of unrelated instructions in *gshare*. Its disadvantage is that the space occupation is relatively large compared with *gshare* because it contains not only a two-level predictor, but also a chooser and a local predictor.

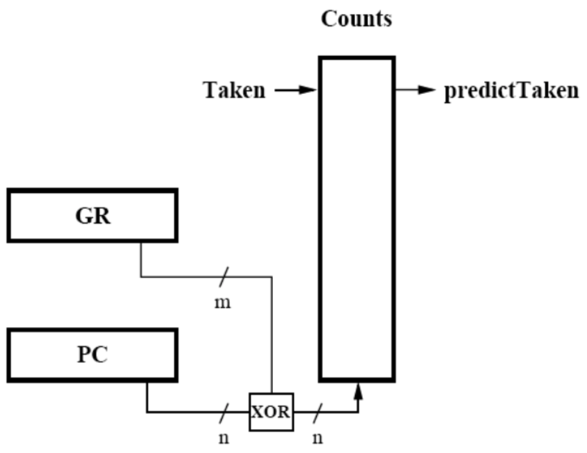


Figure 2. The structure of gshare.

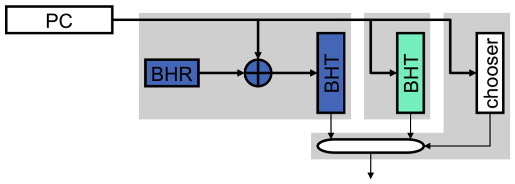


Figure 3. The process of tournament prediction.

Table 1. Branch prediction error rate (%).

Method	fp1	fp2	init1	init2	mm1	mm2	Harmonic Mean
static	12.128	42.350	44.136	5.508	50.353	37.045	16.801
gshare 13	0.825	1.678	13.839	0.42	6.696	10.138	1.331
tournament 9:10:10	0.991	3.246	12.622	0.426	2.581	8.483	1.412
perceptron 164:24	0.824	1.149	8.290	0.306	2.493	8.250	1.001

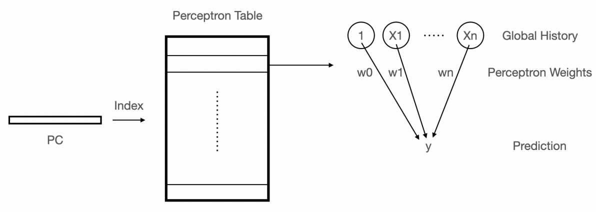


Figure 4. The framework of perceptron branch predictor.

2.3. Perceptron

Conventional dynamic branch predictors mainly inherit the design spirit of Yeh and Patt [17], whose prediction is made via indexing a pattern history table (PHT) of saturating counters by different combinations of branch address and global or local history. Despite notable progress made following this direction, these methods generally suffer from the pain point of short history length due to the exponential growth of table size under a limited area. To tackle this problem, inspired by the breakthrough in the machine learning and neural network community, Jimenez and Lin [3] proposed the first perceptron-based branch predictor.

The perceptron branch predictor consists of a table of perceptrons and a global history register. Each perceptron in the table is a one-layer perceptron with weight $\{w_i\}_{i=2}^n$ and bias w_1, and the global history register $\{x_i\}_{i=1}^n$ records the recent $(n-1)$ branch outcome in $\{x_i\}_{i=2}^n$ and x_1 is always set to 1 to fully factor in bias. Figure 4 illustrates the working mechanism of perceptron.

For inference, a perceptron is first chosen by using the current PC address as the index to the perceptron table

$$w = TABLE[pc\%table\_size] \tag{1}$$

and then the prediction result is calculated as

$$y = w \cdot x. \tag{2}$$

If $y \geq 0$, then the branch is predicted taken, and otherwise not taken.

For training, as the ground-truth branch outcome t (1 for taken and -1 for not taken) becomes available, the chosen perceptron is updated if its prediction is wrong or $|y| < \theta$, where θ is a hyperparameter threshold. The updating of w follows:

$$w_i \leftarrow w_i + tx_i, \tag{3}$$

which intuitively makes a certain weight larger if its corresponding history outcome is aligned with the current outcome and smaller otherwise. Jimenez and Lin [3] empirically set $\theta = \lfloor 1.93(n-1) + 14 \rfloor$ to achieve the best result.

The most notable advantage of perceptron branch predictor over conventional two-level predictors with saturating counters is that it scales linearly regarding history length, which enables it to maintain a longer history under a limited area, whereas conventional predictors scale exponentially. Besides, since the perceptron used by perceptron branch predictor has only one layer, its behavior is totally interpretable as the weights capture correlation between the current branch and previous branches. However, the disadvantage of perceptron branch predictor is also obvious, as its decision boundary is a hyperplane, it cannot precisely model non-linear prediction function, whereas given sufficient training time the two-level PHT scheme like *gshare* can. Despite this limitation, since many functions describing branch behaviors are linearly separable in reality, and the perceptron branch predictor can evolve with time, it can achieve strong performance alone or as part of a hybrid predictor.

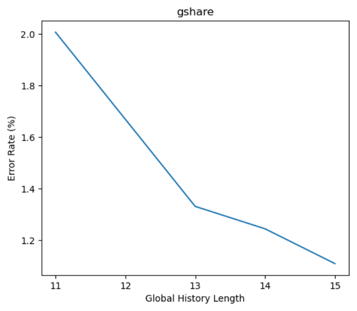
Following the work of Jimenez and Lin [3], many improved branch predictors based on perceptron have been proposed [4]. Tarjan and Skadron [14] propose a hashed perceptron predictor by combining the merits of gshare, path-based, and perceptron-based predictors. Jimenez [2] proposes a piece-wise linear branch predictor that leverages a set of linear functions for different program paths to a branch, greatly enriching the expressiveness of a single linear function. Very recently, some works exploiting more sophisticated neural networks for branch prediction have also emerged [7, 15].

In this project, we focus on the seminal idea of perceptron branch predictor and reproduce the work of Jimenez and Lin [3] to compare with other vintage designs such as *gshare* and *tournament*.

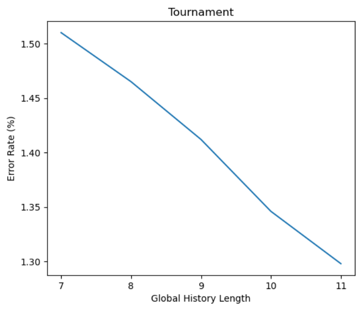
3. Observation

3.1. Implementation Details

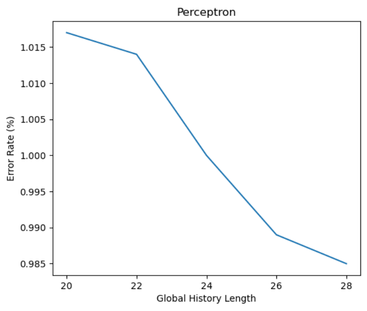
We implement *static*, *gshare*, and *tournament* as three baselines. For the results reported in table 1, the static predictor always predicts branch taken. The *gshare* predictor maintains a history of length 13, which is the largest pos-



(a)

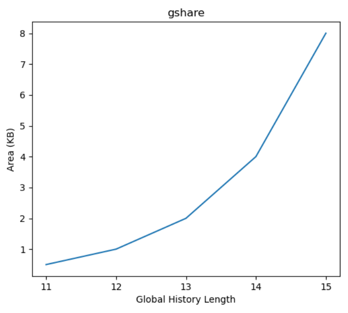


(b)

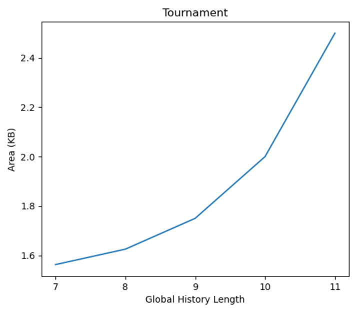


(c)

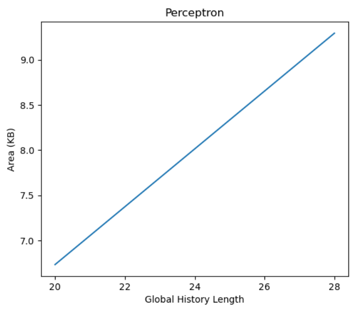
Figure 5. Harmonic mean of misprediction rate using different global history length.



(a)



(b)



(c)

Figure 6. Area of predictor under different global history length.

sible one that fits the 16kb budget. The *tournament* uses 9 bits of global history, 10 bits of local history, and 10 PC bits, which is about 14kb.

For our custom implementation of *perceptron branch predictor* in table 1, we set the number of perceptrons in the perceptron table to 164 and set the length of global history to 24. Each perceptron weight is represented by an int16 variable, and the training threshold θ is an int32 variable. Therefore, the area needed is $164 \times (24 + 1) \times 16 + 24 + 32 = 65656$ bits, which is less than the budget cap of $64Kb + 256b = 65792$ bits.

3.2. Comparative Study

Table 1 shows the branch misprediction rate of three baselines and our custom branch predictor on six test traces. We can observe that *static* predictor achieves the worst result on all traces as it cannot adjust itself with respect to branch history. Conventional dynamic branch predictors such as *gshare* and *tournament* generally achieve better performance, but neither of them is consistently better than the other on all traces. In contrast, perceptron branch predictor consistently outperforms all baselines on all traces, especially lowering misprediction rate by 34.32% and 27.14% in init1 and init2 compared to the runner-up. We reckon that the improvement is brought by the longer history maintained by perceptron branch predictor.

3.3. Global History Length Analysis

The key insight of perceptron branch predictor is that it can scale linearly with global history length, and maintaining a longer global history can improve dynamic branch prediction result. To verify this argument, it is desirable to understand the performance of *gshare*, *tournament*, and *perceptron* under different history length. Despite different design, the three methods share a common factor, which is a global history register. Therefore, in this section, we focus on performance and area change regarding different *global history length*, while fixing other hyperparameters the same as described in section 3.1.

From figure 5 (a), (b), and (c) we can observe that all methods benefit from longer global history length, and perceptron consistently outperforms the other two baselines. This phenomenon is understandable as longer history enable to predictor to capture more correlation between branches. Meanwhile, from figure 6 (a), (b), and (c) we can see that the area consumption of *gshare* and *tournament* grow exponentially with global history length, while *perceptron* grows linearly. These observations verify the effectiveness and scalability of perceptron branch predictor.

4. Results and Conclusion

In this project, we first review the development of the branch prediction technique in history and then discuss the design of three vintage branch predictors: *gshare*, *tournament*, and *perceptron*. We implement the two conventional dynamic branch prediction methods, i.e., *gshare* and *tournament* as baselines, and re-implement *perceptron branch predictor* as our custom predictor. We investigate their branch misprediction rate on six trace testbeds and also track their area usage when maintaining different global history lengths. Verified by quantitative experiments and visualization, all three methods can largely improve over the naive static predictor, and our custom perceptron branch predictor is the most performant and scalable among them. Based on the theoretical design in the original paper as well as our experimental result, we verify that the advantage of perceptron branch predictor results from using a linear perceptron to capture branch correlation in a scalable manner.

5. Team Member Responsibility

This project is a collaborative effort of Chenhao Liu (PID: 59018246) and Xinhao Li (PID: A59018256). Chenhao Liu is in charge of implementing *gshare*, *tournament*, and writing sections 1, 2.1, and 2.2 of this report. Xinhao Li is in charge of implementing *tournament* (jointly with Chenhao Liu), *perceptron*, and writing sections 2.3, 3, and 4 of this report.

References

[1] Thomas Ball and James R Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, 1993. 1

[2] Daniel A Jiménez. Piecewise linear branch prediction. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 382–393. IEEE, 2005. 3

[3] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001. 3

[4] Rinu Joseph. A survey of deep learning techniques for dynamic branch prediction. *arXiv preprint arXiv:2112.14911*, 2021. 3

[5] Toni Juan, Sanji Sanjeevan, and Juan J Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 155–166. IEEE, 1998. 1

[6] Richard E Kessler, Edward J McLellan, and David A Webb. The alpha 21264 microprocessor architecture. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No. 98CB36273)*, pages 90–95. IEEE, 1998. 2

[7] Yonghua Mao, Huiyang Zhou, Xiaolin Gui, and Junjie Shen. Exploring convolution neural network for branch prediction. *IEEE Access*, 8:152008–152016, 2020. 3

[8] Scott McFarling. Combining branch predictors. Technical report, Citeseer, 1993. 1

[9] Carlos Rosales, Antonio Gómez-Iglesias, Si Liu, Feng Chen, Lei Huang, Hang Liu, Antia Lamas-Linares, and John Cazes. Performance prediction of hpc applications on intel processors. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1325–1332. IEEE, 2017. 2

[10] Efi Rotem, Alon Naveh, Micha Moffie, and Avi Mendelson. Analysis of thermal monitor features of the intel pentium m processor. In *TACS Workshop at ISCA-31*, volume 25, 2004. 2

[11] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013. 1

[12] James E Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998. 1

[13] Jared Stark, Marius Evers, and Yale N Patt. Variable length path branch prediction. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 170–179, 1998. 1

[14] David Tarjan and Kevin Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM transactions on architecture and code optimization (TACO)*, 2(3):280–300, 2005. 3

[15] Stephen J Tarsa, Chit-Kwan Lin, Gokce Keskin, Gautham Chinya, and Hong Wang. Improving branch prediction by modeling global history with convolutional neural networks. *arXiv preprint arXiv:1906.09889*, 2019. 3

[16] Ofri Wechsler. Inside intel® core™ microarchitecture: Setting new standards for energy-efficient performance. *Technology*, 1, 2006. 1

[17] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, 1991. 3