Analysis and Accuracy Improvement for Perceptron-based Branch Prediction Method

Jiajing Liu and Shinji Kimura
Graduate School of Information, Production and System, Waseda University
2-7 Hibikino, Wakamatsu, Kitakyusyu, Fukuoka, 808-0135, Japan
E-mail: elspirit@asagi.waseda.jp, shinji_kimura@waseda.jp

Abstract: In the modern microarchitecture, high accuracy of branch predictors is essential for improving the overall performance of the processors. With the emergence of the Perceptron-based predictors, prediction based on long global branch history becomes possible, and this advantage gives this type of strategies much potential in achieving extremely high prediction accuracy. This paper devises two methods to obtain further accuracy improvement through slight modifications on the original structure, and presents the analysis of the limitation of indexing function refinement on Perceptron-based branch predictor. The performance of the two methods and the limitation analysis are evaluated with the championship branch prediction framework.
Keywords: branch prediction, Perceptron, accuracy improvement, limitation analysis

1. Introduction

Branch predictor is an essential component that enhances the instruction level parallelism in the modern processor architectures. With the trend towards deeper pipelines and wider instruction issue rates, the misprediction rate of the branch predictor becomes a critical factor which causes a significant degradation on the overall performance of the processors. Therefore, highly accurate branch prediction is very important to the modern microarchitectures. In recent years, Perceptron-based scheme [1] has been proposed and gives a brand new way to branch prediction. In contrast to the traditional counter-based mechanisms [7,8], Perceptron-based branch prediction not only adopts an intelligent neural network algorithm, but also reduces the memory requirement, which makes it possible to predict based on long history. These advantages make the Perceptron-based scheme promising in pursuing high accuracy [2].

This paper mainly focuses on exploiting the potential of the Perceptron-based branch predictors in accuracy improvement. First of all, some effective ways to enhance the performance of the predictors through modifying the Perceptron weight table are devised. Secondly, the limitation of some accuracy improvement methods on Perceptron-based scheme is clarified.

The remainder of this paper is organized as follows. Section 2 firstly describes the principle of the Perceptron-based Branch Predictor briefly. Then based on the original scheme, two methods called special weight and freshness partition are proposed respectively to yield higher prediction accuracy. Section 3 discusses the limitation of improving the performance of Perceptron-based branch predictor by refining the hash indexing function. Section 4 introduces the methodology of evaluation, and shows the simulation results of the accuracy improvement methods and the limitation analysis. And section 5 gives the conclusion of the paper finally.

2. Accuracy Improvement

2. 1 Perceptron-based Branch Predictor

Perceptron-based branch predictors use a simple linear neuron called Perceptron to make the prediction. The principle of this type of branch predictors is illustrated in Figure 1. A Perceptron holds a vector of weights which are trained with a machine learning algorithm based on the prediction outcomes and global branch history. When a branch instruction flows into the pipeline, an index is generated with the branch address and a hash function to access the Perceptron weight table and fetch the corresponding weight vector. Then the dot product of the global branch history and the weight vector is calculated, and the sign of the result decides whether a branch shall be taken or untaken.

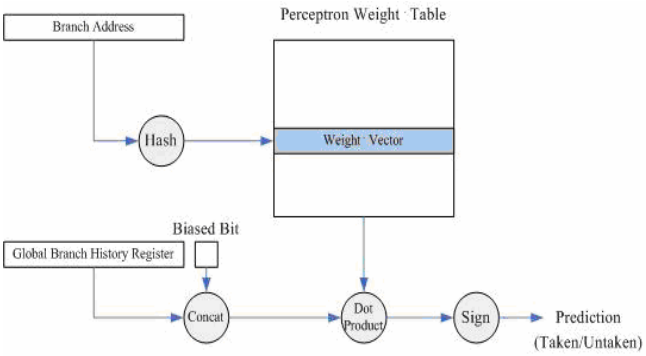


Figure 1. Principle of original perceptron-based branch predictor

When the actual outcome of a branch is decided, the training process is likely to be invoked. The outcome is compared with the global branch history bit by bit, and each corresponding weight value is updated according to the comparison. This training process makes the prediction more adaptable to both the branch address and the global branch history pattern.

Based on the original Perceptron-based branch prediction strategy, a variety of evolved schemes are proposed to get better performance. For example, [5] combines the path information and pattern history to make prediction, which results in improved accuracy; [6] obtains its extra accuracy by merging the path and GSHARE function to generate different index when accessing each weight of the Perceptron vector. Although those schemes

prove to be superior, they have not fully resolved the problems of the original Perceptron-based branch predictor. By considering other aspects of the matter, some new methods can be devised to improve the accuracy.

2. 2 Special Weight

Through some statistical analysis of the runtime behavior of the original Perceptron-based branch predictor, we firstly notice that the predictor has the tendency to make wrong predictions on some extremely biased branches, whose outcomes hardly change throughout the whole execution process. These extremely biased branches widely exist in the infinite loops, assertions, exception handlers, and other places of the programs. Therefore, it is meaningful to exploit the prediction accuracy on those branches. This problem has been noticed in the research on counter-based branch predictors, and some approaches that adding flag bit to each branch address to decide whether the prediction is reliable or not have been proposed [9]. However, such aggressive approaches do not deliver a good solution to the Perceptron-based branch predictor. In order to make the Perceptron-based prediction more adaptable to those extremely biased branches, we propose a relatively conservative approach, which attaches a special weight which to some extent indicates the confidence to each Perceptron. This special weight keeps the difference of prediction success times and failure times of the Perceptron, constrained by an upper limitation in case of giving much negative influence on the common branches. When making the prediction, the predictor checks the sign of the dot product of the Perceptron weight vector and the global branch history. If the dot product is greater or equal to zero, the special weight will be added to the dot product; otherwise, the special weight will be subtracted from the dot product. This process, as is shown is Figure 2, can be regarded as a compensation to the calculation result. In this way, we can to some extent prevent the prediction distract from the right directions of the extremely biased branches.

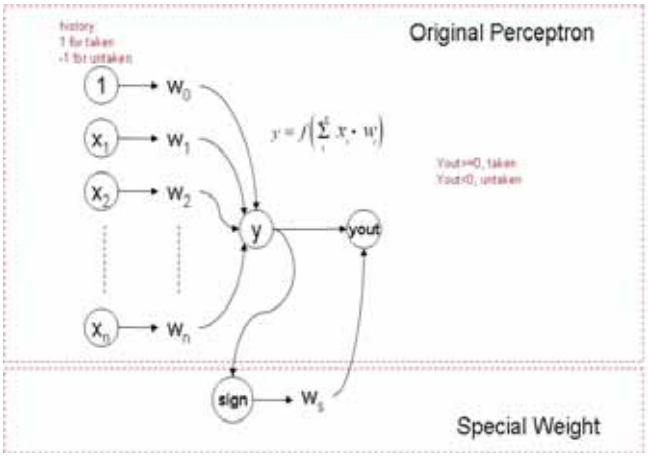


Figure 2. Attaching the special weight to the original weight vector

2. 3 Freshness Partition

The second modification is to divide the weights vector into two parts and attach different importance to those weights according to freshness of the history they are related to. Since the weights related to newer history usually have stronger impact on the decision, the devised scheme lets those weights have larger contribution to the Perceptron calculation result. To implement this idea, we need to set a partition standard value N, according to which the weights vector fetched out from the Perceptron weights table is divided into a newer group and an older group. Since the newer group of weights should be attached more importance, they are left-shifted by one bit, namely doubled, before the dot product calculation. On the other hand, older group of weights are directly used for the calculation without any pretreatment. The principle of the freshness partition method is illustrated in Figure 3.

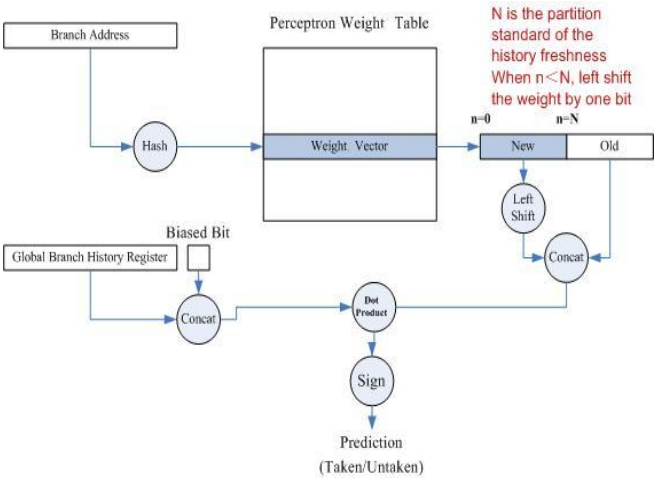


Figure 3. Revising the Perceptron-based branch predictor with freshness partition

In the original Perceptron-based branch predictor, the training threshold value is $\theta = 1.93n + 14$, in which n is the global branch history length. When misprediction occurs or the Perceptron calculation result is less than θ, the training process will be invoked. With the use of the freshness partition method, since the calculation formula is changed, the threshold value θ should also been adjusted to adapt to the weights variation. As we suppose the history length is n and the partition standard value is N, the θ should be linearly modified to

$$\theta = \frac{(1.93n + 14)[n + 1 + N]}{n + 1}$$

3. Limitation Analysis

Some research shows that using refined hash function to generate the index to the prediction table successfully improves the performance of the counter-based branch predictors [3]. In the Perceptron-based scheme, the destructive aliasing still causes performance degradation. We try to estimate how the hash function refinement affects the accuracy improvement of the Perceptron-based branch predictor, by eliminating some constraints of practical hardware implementation and modeling some ideal conditions in the simulation. As the on-chip memory space

is very limited in the hardware implementation, hash function is very important to utilize that space efficiently. However, in the software simulation, we can take advantage of the main memory of the computer to provide sufficient space for the Perceptron weights table. In this case, the branch addresses can be directly used as the indices without any hash function. Therefore, no destructive aliasing will exist in this ideal environment. This experiment will show how the destructive aliasing degrades the performance, and the limitation of improving the accuracy by minimizing the destructive aliasing.

4. Methodology and Result

4.1 Simulation Framework

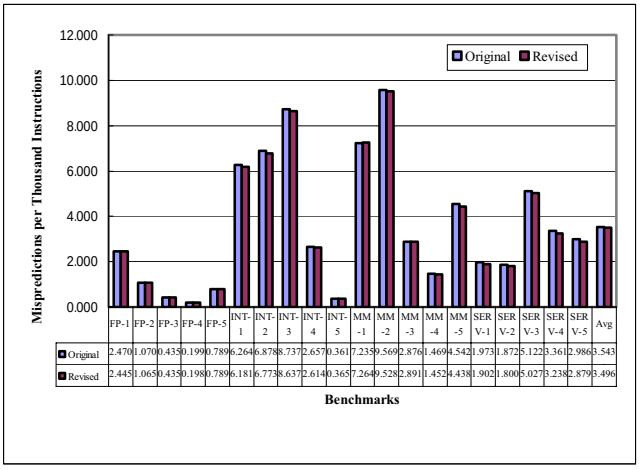
The accuracy of the schemes we devised and the limitation of the indexing function refinement are evaluated with the championship branch prediction framework version 3 by Intel Corporation and IEEE TC-uARCH [4]. This is a trace-driven simulation framework, which contains trace data of 20 benchmarks. The 20 benchmarks are classified into 4 categories, including SPECfp, SPECint, multimedia, and server application programs. The performance is measured by the misprediction rates on these 20 benchmarks, in unit of misprediction per thousand instructions.

4.2 Evaluation of Accuracy Improvement

The branch predictors are modeled in C++ and embedded into the framework to do the simulation. We have written 4 types of predictors: an original Perceptron-based branch predictor, three revised predictors respectively based on special weight, weights freshness partition, and a combination of these 2 methods, each of which is in about 200 lines of C++ code. We evaluated the accuracy of the original Perceptron-based branch predictor as the evaluation baseline, and then modeled the proposed schemes with 16 bits and 32 bits history length respectively to measure the accuracy improvement. Different parameters like the upper limitation of the special weight and the freshness standard for the weight partition are adjusted in the simulation to seek for the optimal ones. The experimental results are shown in Figure 4 and Figure 5.

Figure 4 shows the misprediction rate comparison of original Perceptron-based branch predictor and our devised scheme.Figure 5 shows the misprediction rate reduction of the devised scheme comparing with the original one as a baseline. Evaluation result shows that using 5 bits to store the special weight and doubling the newest one fourth of the weight vector are good choices of these two parameters. Under these conditions, the methods we devised reduce the misprediction rate by 1.33% on average.

Though the average improvement is slight, it does have a relatively strong effect on the server type benchmarks and achieve up to 3.85% improvement on that type of programs. In our statistical analysis, we found that the server type benchmarks have a relatively large portion of extremely biased branches. And after the modification, the failure rate of the predictions on those branches decreases obviously.



	FP-1	FP-2	FP-3	FP-4	FP-5	INT-1	INT-2	INT-3	INT-4	INT-5	MM-1	MM-2	MM-3	MM-4	MM-5	SERV-1	SERV-2	SERV-3	SERV-4	SERV-5	Avg
Original	2.470	1.070	0.435	0.199	0.789	6.264	6.878	8.737	2.657	0.361	7.235	9.569	2.876	1.469	4.542	1.973	1.872	5.122	3.361	2.986	3.543
Revised	2.445	1.065	0.435	0.198	0.789	6.181	6.773	8.637	2.614	0.365	7.264	9.528	2.891	1.452	4.438	1.902	1.800	5.027	3.238	2.879	3.496

Figure 4. Misprediction rate comparison between original and revised Perceptron-based branch predictor (32 bits history length)

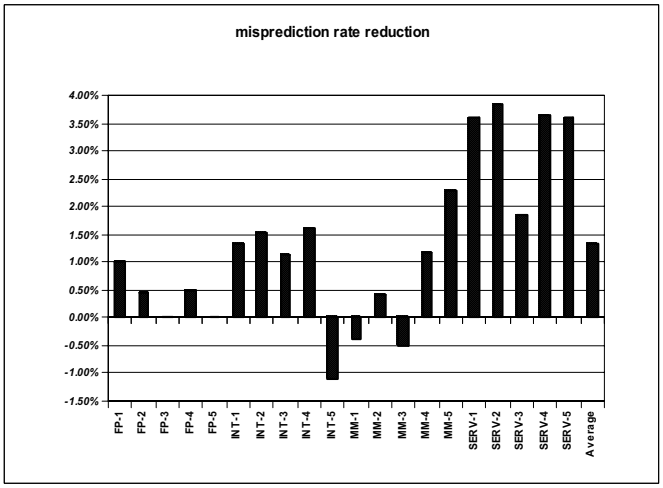


Figure 5. Misprediction rate reduction of the devised methods comparing to the original Perceptron-based branch predictor (32 bits history length).

4.3 Evaluation of Accuracy Limitation

Based on the same simulation framework, the effect of the indexing function refinement is also evaluated. Firstly, a predictor with a common hash function, which computes the modulus of the 2-bit-right-shifted branch address and the Perceptron table size, is modeled. Then the ideal situation in which no aliasing exists is modeled to show accuracy improvement limitation of minimizing the destructive aliasing. Simulation result is shown in Figure 6, and it shows in the case of 32 bits history length, an ideal indexing function that causes no aliasing will reduce the misprediction rate by about 4.42% on average.

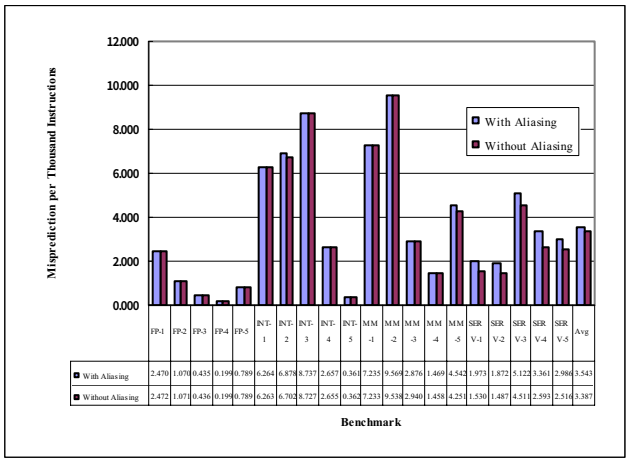


Figure 6. Comparison of the misprediction rates of the Perceptron-based branch predictors with and without prediction table aliasing (32 bits history length).

It is obvious that the server type benchmarks suffer most from the destructive prediction table aliasing, so that the indexing function refinement has the potential to reduce the misprediction rates of those programs by up to 22.45%. Notice that the result of MM-3 benchmark shows an unusual behavior that prediction without aliasing performs worse on the contrary. The reason might be that non-destructive aliasing, in which the branches with similar history patterns and the same directions share the same Perceptron vector, take a large portion of the overall aliasing in that benchmark. However, such situation does not widely exist, and the aliasing problem is destructive in most of the program so that it usually tarnishes the predictor performance, as illustrated in the experimental results.

5. Conclusion

In this paper, two methods are proposed to improve the accuracy of the original Perceptron-based branch predictor. A special weight is attached to each of the Perceptron weights vector to compensate the misprediction tendency on the extremely biased branches. Freshness partition lets the newer part of the weights vector give more contribution to the Perceptron, which makes the calculation more reasonable. The two methods prove to be effective in the simulation. On the other hand, the effect of the hash indexing function on the Perceptron-based branch predictor is estimated. The simulation shows the limitation of improving the accuracy by hash function refinement. In the future work, we will also evaluate other factors that have influence on the Perceptron-based branch predictor.

Acknowledgement

This work was partly supported in part by a grant of Knowledge Cluster Initiative implemented by Ministry of Education, Culture, Sports, Science and Technology (MEXT).

References

[1] D. A. Jimenez and C. Lin, “Neural Methods for Dynamic Branch Prediction”, ACM Transactions on Computer Systems, 20(4):369-397, 2002

[2] C. Y. Ho, K. F. Yau and Anthony S. S. Fong, “A Study of Dynamic Branch Predictors: Counter versus Preceptron”, In Proceedings of 4th International Conference on Information Technology, pp. 528-536, 2007.

[3] Yi Ma, Hongliang Gao, and Huiyang Zhou, “Using Indexing Functions to Reduce Conflict Aliasing in Branch Prediction Tables”, IEEE Transaction on Computers, VOL. 55, NO. 8, pp. 1057-1061, 2006

[4] The official CBP-3 website: http://www.jilp.org/cbp

[5] D. A. Jimenez, “Improved Latency and Accuracy for Neural Branch Prediction”, ACM Transaction on Computer Systems, 23(2): 192-218, 2005

[6] D. Tarjan and K. Skadron, “Merging Path and Gshare Indexing in Perceptron Branch Prediction”, ACM Transactions on Architecture and Code Optimization, 2(3): 280-300, 2005

[7] J. E. Smith, “A Study of Branch Prediction Strategies”, In Proceedings of the 8th Annual International Symposium on Computer Architecture, pp. 135-148, 1981

[8] T. Y. Yeh and Y. N. Patt, “A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History”, In Proceedings of the 20th Annual International Symposium on Computer Architecture, pp 257-266, 1993

[9] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba, “The Bimode++ Branch Predictor”, In Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems, Page(s). 8pp, 2005