Comparison of the Efficiency and Theoretical Hardware Consumption of Four Different Dynamic and Static Branch Predictors

Boyang Li

Computer science and technology, Harbin Institute of Technology, Harbin, Heilongjiang, 15001, China

Corresponding author’s email: gaoming@cas-harbour.org

Abstract. When CPU enters the era of instruction level parallelism, the execution cycle of instructions has become shorter and shorter. Meanwhile, the processing speed of CPU has become faster and faster. However, only from the execution process of CPU instructions, there is a key factor that limits the speed of CPU processing instructions, namely, the branch instruction processing. The branch predictor is used to predict the execution result of the current branch instruction (taken/non-taken), which can make the program counter get the target instruction to continue to execute according to the prediction result before calculating the target jump address. If the prediction is correct, it will greatly improve the execution efficiency of CPU. This paper starts from the two basic categories of static branch prediction and dynamic branch prediction. Then analyse their prediction principle, prediction accuracy and component consumption used for prediction. For static branch predictor, its hardware consumption is less than that of dynamic branch predictor, and its accuracy is also lower than that of dynamic branch predictor. Moreover, their shortcomings are pointed out in this paper. At last, based on the original branch predictor, better branch predictors developed in recent years are introduced. In the future, considering the hardware consumption, I think the development of branch predictor will continue to optimize the branch predictor combining global history and local history, from the number of stored entries, the correlation between branch instructions and so on. On the premise of executing a large number of instructions, every little improvement in the efficiency of the predictor will bring huge benefits.

1. Introduction

The traditional branch predictor introduced in this paper is divided into static branch prediction and dynamic branch prediction. The static branch predictor includes taken and not-taken, and the dynamic branch predictor includes bimod and 2-level. The second section briefly introduces the working principle of static branch predictor and dynamic branch predictor. Section 3 discusses these four basic branch predictors from the hardware cost required to implement the prediction process. Besides, the effect of the branch predictor will be discussed from both hardware and software aspects. After the most basic branch predictor is released and put into use, in order to improve the accuracy of prediction, branch predictor is still continuously developed. Section 4 introduces different improvement directions of branch predictors and a related improvement example from different perspectives of improvement. Section 5 shows the conclusion and the relevant articles referred to are recorded in Section 6.
2. Static and dynamic branch predictors

2.1. Static branch predictor
As the name suggests, the information for prediction of static branch predictor will not change during the execution of the program, but the branch result will be directly predicted through the instruction itself. For a taken-branch predictor, each time a branch instruction is encountered, the target address of the current branch instruction is put into the PC. In contrary, for the not-taken branch predictor, PC just take the address of the next instruction.

2.2. Dynamic branch predictor

2.2.1. Bimod branch predictor
It should be noted that program branches do not appear randomly in the program. Most branch instructions are usually taken or usually not taken [1], which means that we can record the history of the branch instruction execution to make inferences. The simplest method is to use a k-bit saturated counter to record the historical execution, and then index to the corresponding counter by the low-order of PC for prediction, as shown in Figure1. Experiments have shown that the prediction effect of a counter with a length of 2 is the best [2]. Here, a counter with a length of 2 is taken as an example to illustrate the bimod branch predictor. For each successfully executed branch instruction, the counter according to its PC index must be plus 1. On the contrary, each branch instruction that is not executed makes the corresponding counter minus 1. In addition, this counter is saturated, which means that its value will only change between 00 and 11 in binary. When encountering the same branch instruction again, the corresponding saturated counter needs to be found according to the PC, and whether the branch instruction is taken should be predicted according to the highest bit value of counter (1 means taken, 0 means not-taken). And then the PC can be updated according to the prediction result, so that the program can continue to execute without idling. After knowing the final execution result of the branch instruction, the predictor updates the counter according to the strategy mentioned above.

![Figure 1. Bimod predictor [1].](image)

2.2.2. Two-level branch predictor
The two-level branch predictor uses two-level branch history information to make branch predictions. The first level is called the branch history register (BHR), which is used to store the execution history of the k branch instructions recently encountered. According to different model implementations, these k instructions can be k branch instructions actually encountered in the whole program (Global), the last k times of the same branch instruction (Local), or the last k branch instructions that occurred under the same set (Set) [2]. All BHRs are stored in the branch history table (BHT). When a branch is taken, 1 is recorded in the corresponding BHR, otherwise, 0 is recorded. The second-level predictor records the pattern that occurred in the last j of the k records of the first-level branch history. The current branch is
predicted based on this pattern. For a k-bit branch history register, there may be $2^k$ different patterns, each of which is associated with a second-level predictor, and these second-level predictors are stored in the pattern history table (PHT). After each branch instruction is executed, the two-level branch predictor will be automatically updated according to the execution result.

The process of the 2-level branch predictor to execute branch instructions is as follows: For the branch predictor whose first level is the global history, every time a branch instruction is encountered, the global history register is used to index the second level pattern history, which is shown in figure 2. If the first level is local history, it will need to be indexed to the history register of the corresponding instruction according to the low-order of PC, and then indexed to the corresponding pattern history according to the local branch history, as can be seen in figure 3. Then the pattern history is used to construct a saturated counter make the final prediction. The prediction principle is as same as bi-modal. Updating the corresponding history register and pattern history after the prediction is completed, where the pattern table is the entry indexed before updating the global history. If the prediction is found wrong after the current branch instruction is completed, the two-level predictor will need to be restored to the previous state and updated according to the correct result.

![Figure 2. Global branch history predictor [1].](image)

![Figure 3. Local branch history predictor [1].](image)

3. Analysis of four branch predictor

3.1. Static branch predictor

For the taken/not taken static branch predictor, its implementation is very simple and hardly requires additional hardware resources. As long as the branch instruction is encountered, the PC can be directly updated according to the taken/not taken strategy. Therefore, the hardware resource consumption of the two static strategies is the lowest, but its correct rate is not high, because its prediction does not refer to
any historical execution information. In addition, it is impossible to make accurate predictions in a complex branch instruction environment.

3.2. Bimod branch predictor
For the bimod branch predictor, its hardware consumption is related to the number of saturated counters. For the bimod branch prediction model with n saturated counters, it can store the execution status corresponding to n branch instructions. When a saturated counter is k bits, a total of n*k bits is needed to record the execution data of the branch prediction. Since it refers the past k execution results of a branch instruction, it can accurately predict the branch instruction correctly. The bimod branch predictor also has two obvious shortcomings. For one thing, considering the limitation of hardware consumption, n cannot be very large. It means only the lower bits of the PC can be indexed to the corresponding saturated counter. So there is an obvious problem in larger programs that the lower bits of the different branch instruction may be indexed to the same saturated counter, which will greatly reduce the performance of the bimod predictor, because a predictor may be modified due to several unrelated branch instructions, so that the counter loses the reference value for the branch instruction currently to be predicted. In addition, even if a k-bit saturated counter has the same value, its branch instruction environment will be different. From another point of view, there may be too few k bits. The execution result records of two longer branch instructions may have a big difference in the front part, while the last few bits of them are always the same. This means that even if the patterns of the two saturated counters are the same, its actual execution direction may be completely different under two different long branch histories. This will cause the same saturated counter fail in making accurate predictions.

3.3. Two-level branch predictor
The 2-level branch predictor alleviates the shortcomings of the bimod branch predictor to a certain extent. It first looks up the branch history table for local branch prediction [1]. Meanwhile, it needs to be indexed by PC. For global branch prediction [1], there is only one global history entry. According to the branch history, the different execution patterns of the latest k branch instructions can be indexed, so that the historical execution information of the branch instructions is combined when the saturated counter is used for branch prediction. In terms of hardware resource consumption, for local branch prediction, an n-bit register is used to record the past n execution history records of a branch. That is to say, for n different historical patterns, each pattern must correspond to a k-bit saturated counter. In general, if the first level can store m branch histories, m n-bit registers are needed to store the first level data, and 2^n k-bit saturated counters are needed for branch prediction. For the 2-level branch predictor that uses the global history, the first-level only requires an n-bit global history register, while the second-level requires 2^n k-bit saturated counters to make predictions.

3.4. The impact of software on branch prediction
In addition to different branch predictors themselves, software also affects the accuracy of predictions. For example, there is such a program: The program first generates 2048 integers in the range of 0-512, traverse all random numbers without sorting, add all the numbers less than 256, and output the final sum. If no processing is done at compile time, the prediction result will be random every time when judging whether the current number is less than 256. The core of dynamic branch prediction refers to the previous branch instruction execution history. Furthermore, the irregular branch history results cannot help predictors make accurate predictions. In order to solve these problems, relevant optimizations can be performed at compile time, which can also improve the accuracy of branch prediction.

4. The development of branch predictors

4.1. Improve the accuracy of static branch prediction by referring to execution information
No reference to the execution of the program for analysis and prediction is the limitation of static branch prediction. If the branch prediction is based on the running results of a previous program, the accuracy
of its prediction will be greatly improved. Fisher and Freudenberger's experiment has strongly showed that since branches go in one direction most of the time, the static history-based prediction schemes which use previous runs of program are effective across the different data sets of most applications [3].

4.2. Reducing the interference of different instructions in 2-level branch predictor
For local branch prediction, there are still the following shortcomings: Firstly, different branch instructions may still be mapped to the same branch history. In addition, because all branch histories share a pattern array, there are still different branches that use the same saturated counter for prediction. Therefore, the core of the problem is to construct a mapping as sparse as possible from each branch instruction to pattern history table. One feasible method is to use gshare branch prediction [1]. Through exclusive ORing branch address and global history, the current branch instruction is mapped to different entries in PHT as much as possible. As a result, the interference between different branch instructions will be reduced. In addition to XOR, there are many other hash methods that can achieve the function of sparse mapping.

4.3. Adaptive dynamic branch prediction length adjustment
Another problem with the 2-level branch predictor is that not all bits of the branch history are related to the execution of the current branch instruction. The length of the fixed branch history obviously cannot achieve optimal prediction. According to Jong Wook Kwak and Chu Shik Jhon, a method of dynamically adjusting the history length of each instruction (DpBHLA) is proposed to improve the accuracy of local branch prediction [4]. The core of the algorithm is to identify whether there is data related to the current branch instruction in the branch history and to identify strongly correlated branches. After the recognition is completed, the length of the history register is dynamically adjusted to its optimal length, namely, eliminating non-strongly related instructions as much as possible in history register. The experiments show that the misprediction rate of dynamically adjusted length is lower than that of fixed length under different benchmarks.

4.4. Branch predictor based on neural network
For the static branch predictor, the neural network can be used as the input of a trained neural network, according to the control flow and opcode information provided by the program at compile time. Moreover, the output is the direction of branch prediction. Through this technology the accuracy rate can achieve 80% compared to that of static heuristics with 75% [5].

For dynamic branch predictors, if two related branch instructions are far apart, most 2-level branch predictors have no way to consider long history lengths [6], but simply rely on increasing branches. The length of the history is unrealistic, because longer training times may be required to achieve these tasks. Hence, it is proposed to realize branch prediction through perceptron [5]. Its hardware consumption is smaller than other neural network-based branch predictors, but compared to the traditional 2-level branch predictor, hardware resource consumption is much more. [5] Branch prediction with perceptrons’ principle is to predict whether a branch will occur through n inputs (x1, x2,...xn, where xi are the bits of a global branch history shift register). The model of the perceptron is \( y = w_0 + \sum_{i=1}^{n} x_i w_i \), where \( w_i \) is a signed integer. By training the weight \( w \), a branch predictor with better performance can be obtained.

5. Conclusion
The branch predictor is based on the two basic directions of static and dynamic. After years of development, it has been able to provide tremendous help in improving processor efficiency. Furthermore, the future developing direction of static branch processors lies in making predictions with reference to previous execution history information. Although its accuracy rate has not reached very high, it can be used for program compilation and provide effective information for dynamic branch prediction [5]. For dynamic branch predictors, the main room for improvement lies in the size of 2-level tables and historical information. Determining an appropriate table size makes each branch instruction use its pattern history entry as independently as possible without causing a large number of idle entries,
which can improve the prediction efficiency. In addition, how to match the length of historical information with the maximum distance between the current branch instruction and its strongly related branch instruction to make accurate predictions has been an optimization direction. At the same time, the improvement of prediction accuracy often means the improvement of hardware resource consumption. In order to maximize the role of branch predictor, it is necessary to trade off the execution efficiency and hardware resources.

References
[1] Predictors B, Mcfarling S. Combining Branch Predictors[J]. Wrl Technical Note Tn Digital Equipment Corporation, 1998.
[2] Yeh T Y, Patt Y N. A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History[C] ACM SIGARCH Computer Architecture News. IEEE, 1993.
[3] Fisher J A, Freudenberger S M. Predicting conditional branch directions from previous runs of a program[C] International Conference on Architectural Support for Programming Languages & Operating Systems. ACM, 1992:85-95.
[4] Jong, Wook, Kwak, et al. Dynamic per-branch history length adjustment to improve branch prediction accuracy[J]. Microprocessors and Microsystems, 2007, 31(1):63-76.
[5] Jimenez D A, Lin C. Dynamic branch prediction with perceptrons[C] International Symposium on High-performance Computer Architecture. IEEE Computer Society, 2001:197-206.
[6] Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In Proceedings of the 25th Annual International Symposium on Computer Architecture, July 1998.
[7] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In Proceedings of the 24th International Conference on Computer Architecture, June 1997.