Homework 2 Report

李京叡, 0616329

***Abstract*—This electronic document is the homework 2 report for the course—Microprocessor Systems: Principles and Implementation at NCTU in 2020 fall.**

I. INTRODUCTION

A branch predictor is a circuit that attempts to guess which branch the processor will take before it certainly known. This is an important mechanism designed for pipelined processor, since the processor divides execution of an instruction into five stage and each stage has dependence on other stages. Without branch prediction, when the processor comes across a conditional jump instruction, it would have to wait until the instruction has passed the execute stage before the next instruction can enter the fetch stage. However, with branch predictor, we can save the time for such waiting. Branch predictor trying to guess whether the conditional jump will be taken or not taken according to the previous records, if the guess is correct, the processor can keep going executing without stopping and this is very efficient for pipelined processors. However, if the guess is incorrect, then the partially executed instructions will be discard and the pipeline starts over with the correct branch. This process will cause a delay. Nevertheless, a processor with branch predictor is more efficient than that without branch predictor. To be more specific, take Aquila as an example. Aquila have the performance of 0.71 DMIPS/MHz with branch predictor and 0.61 without branch predictor.

Branch prediction can be divided into two types: static branch predictor and dynamic branch predictor.

A. Static predictor

A static predictor is the simplest branch prediction since it does not consider the records of history of code executing. This technique predicts the branch to take based solely on the branch instruction. The simplest way to implement the prediction is to predict the branches are always taken or always not taken. In practice, a static branch predictor should always predict taken, since most branches are taken. The downside of this kind of prediction is that it has a large variance across different programs, and it requires the compiler to optimize for branches being taken.

B. Dynamic preditor

Dynamic branch prediction uses information about taken or not taken branches gathered at run-time to predict the outcome of a branch. There are several types of dynamic prediction, below I just name a few:

- One-bit prediction

The branch history table or branch prediction buffer stores 1-bit values to indicate whether the branch is predicted to be taken or not. The next time that branch is called, the result stored in the table is used as the prediction. Each time when the prediction is wrong, the predictor flips the prediction bit. So, the prediction bit flips quit frequently and the prediction accuracy of this technique is not very high but still usually better than static prediction.

- Two-bit prediction

The technique of two-bit prediction is similar to one-bit prediction but the predictor changes its prediction only on two successive mispredictions. Two bits are maintained in the prediction buffer and there are four different states, which are strongly taken/not taken and weakly taken/not taken. The predictor transfers between the four states according to the current state and the correctness of branch prediction.

- Tournament predictors:

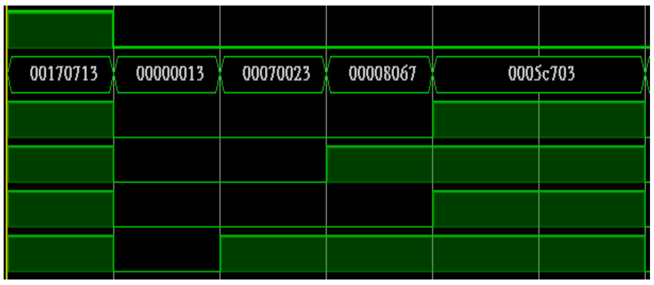
A tournament predictor maintains two different predictors, one is based on global information and one is based on local information. Typical practice is to use a two-bit saturating counter to select between two predictor. The switch between two predictors can be made when there are two successive mispredictions.

II. BRANCH HISTORY TABLE (BHT)

In Aquila, the predictor maintains a branch PC history table. The table stores 32 branch target addresses, and it is written in order from address 0 to 31. If the table is full, it will return back to write from the address 0, and so on. The predictor also maintains a branch PC table, which stores the program counter addresses of branch instructions corresponded to the target addresses stored in the branch history table. Each time when the processor encounters a conditional branch instruction, the predictor checks one by one to make sure if the program counter address has been stored in the branch PC table. If so, the branch predictor reads the branch target address directly from the branch PC history table. If it cannot find the program counter address in the branch PC table, it will write the program counter address into the branch PC table and write the branch target address into the branch PC history table.

III. FLUSH

When a misprediction for conditional branch instruction occurs, the pipeline control will send a flush signal to fetch and decoder module. At the time fetch module get the signal, it sets the instruction output as 0x00000013, which is NOP instruction, and resets some control signals. Similarly, decoder module resets all its control signals. I took fetch_valid_o in both module to observe, and the ILA waveform output is as follow:



, where the signals from top to bottom are "branch_misprediction" from execute module, "instruction_i" being sent to decode module, "flush_i" being sent to decode module, "fetch_valid_o" from decode module, "flush_i" being sent to fetch module, "fetch_valid_o" from fetch module relatively. In the first clock cycle, branch_misprediction occurs, and the pipeline control send flush signal to decode and fetch module, in the next clock cycle, fetch_valid_o in decode and fetch module are reset to 0.

IV. STATIC PREDICTOR

For static prediction, I simply set the branch decision to 1 in branch prediction module, which means that the predictor always choose to take the branch and the final result of Dhrystone is decreased to 0.67 DMIPS/MHz. In the beginning I observe the signal when the program counter hit the 0x1e88, which is "strcpy" function like homework 1 I did, there is a conditional backward jump which is a loop that check if the program has reached the end of the string, the program takes branch of "bnez" for 30 times, since the predictor always predict to take branch, it is correct in the beginning of 30 times. However, at the 31st time, misprediction occurs. Later, I found that observing the "strcpy" function is not worth studying for mispredictions, since the branch condition in the function is too simple, the function only contains one loop structure, so the result is the same as the original dynamic predictor. As a result, I turned to observe the signal of execution of main function and the difference between static and dynamic predictor is much apparent. The main function calls "malloc" function, and in the "malloc" function the original dynamic predictor only predicted wrongly for one time, while the static predictor predicted wrongly for three times.

V. MODIFICATION OF BHT PARAMETER

I resize the branch PC table and the branch PC history table to be the half of the original one, 16 entries. The result is that the Dhrystone performance drops to 0.68 DMIPS/MHz. The reason I think is that since the table size is smaller than the original one, and the mechanism of updating the table is writing into each address one by one in order, the entries in the smaller table would be more often to be overwritten than those in the bigger table. As a result, the records of the program counter addresses and the corresponded branch target addresses stored in the tables are more likely to be overwritten before they hit again. I also tried setting the tables size to be twice of the original one, 64 entries. The result is the same as the original one. This result indicates that 32 in length is enough to handle the conditional branch instructions for Dhrystone program.

VI. ONE-BIT PREDICTOR

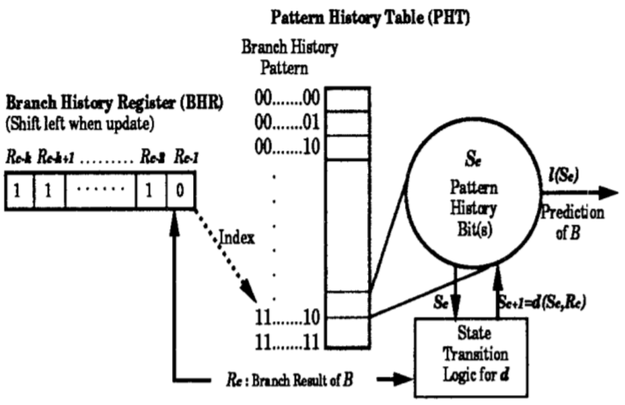
I set the branch_likelihood in bpu module to be one bit and flips every time when misprediction occurs, the result is a little worse than the original two-bit dynamic prediction. In fact, the difference is not obvious. I have to look for more decimal places to find the difference between two predictors. The original 2-bit predictor perform 0.713521 in DMIPS/MHz, and the performance of 1-bit predictor is 0.712330 DMIPS/MHz. The improvement from 1-bit to 2-bit predictor is very slight, at least for Dhrystone program.

VII. IMPROVEMENT

In the beginning, I thought the way in which the predictor looks up for the branch history table is not efficient, since it checks every entry one by one each time to find whether the program counter has been stored in the branch_pc_table. The time complexity of this method is O(n), where n is the length of branch_pc_table. I implemented another approach, in which I set lower five bits of the program counter as tag, so each program counter address is stored in the table corresponded to its tag. The predictor can quickly find which of the addresses the program counter is stored by the tag. The time complexity of a finding is O(1). However, the performance of the approach is not that good as I expected. In fact, the DMIPS/Mhz even dropped a lot to 0.61. I have tried choosing some other five consecutive bits as tag, and get some better results, and the best one is 0.65 DMIPS/MHz, but still worse than the original predictor. I think the major reason is that through this way of choosing tags, each program counter's tag is not distributed evenly, which makes the table stored unevenly. In other words, some slots of the table have never stored any entry, while some other slots repeatedly stores the entries and constantly changes its value. As a result, when the predictor tries to look up for the predict target through searching by tag, the entry might have been overwritten by other program counter with the same tag. Also, 32 is not a huge number for hardware today, checking table entries one by one for 32 times is not a big deal. As a consequence, considering the possible factors above, the original predictor outperforms the predictor using tags for searching.

VIII. TWO-LEVEL PREDICTOR

For two-level predictor, I refers [1] and [2] and implement the GAg scheme two-level adaptive branch predictor. The predictor uses two levels of branch history information to make prediction. The first level is a k-bit shift register, called branch history register (BHR), which records the last k branches encountered. (Here I set k to be 5) The second level is the pattern history table (PHT). There are 2^k patterns in the table, and each can be indexed by the BHR. The table keeps updating every time when encounters conditional branch instruction and it works like the branch_likelihood in the original predictor does, which has four states and transfers between them according to the current state and the conditional branch result. The structure of the predictor is as following (from [2]):


Pattern History Table (PHT)
Branch History Pattern
Branch History Register (BHR)
(Shift left when update)
R_{c-k} R_{c-k+1} .... R_{c-2} R_{c-1}
1 1 ...... 1 0
00.......00
00.......01
00.......10
11.......10
11.......11
Index
S_c
Pattern History Bit(s)
$\lambda(S_c)$
Prediction of B
S_c
$S_{c+1}=\delta(S_c,R_c)$
State Transition Logic for δ
R_c : Branch Result of B

Before implementing the predictor, I expected that a two-level branch predictor should definitely performs better than a one-level predictor, because it has a more complex structure and technique for branch prediction. However, the result is not same as I expected, the two-level predictor runs Dhrystone with only 0.66 DMIPS/MHz, which is worse than the original one-level predictor, but still performs better than running without branch predictor. Observing the signal, there are significantly more mispredictions for two-level predictor than one-level predictor. Assume that all my implementation details are correct, I think that the reason is that Dhrystone benchmark is not a good benchmark intended to evaluate the branch predictor. As a result, the two-level predictor didn't perform better than one-level predictor in terms of Dhrystone benchmark.

IX. REFERENCE

[1] T. Yeh, Y. Patt, "A comparison of dynamic branch predictors that use two levels of branch history", Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 257-266, 1993.
[2] T-Y Yeh, Y.N. Patt, "Two-Level Adaptive Branch Prediction", Proceedings of the 24th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture, pp. 51-61, 1991-Nov.