ECE552
Assignment 2

Canna Wen [996117205]
wencanna@ecf.utoronto.ca

Introduction: Objectives and Motivations

Branch prediction is a field of study in computer science in which researchers try to come up with algorithms to accurately "guess" how a program will execute. This guess is impacted by a variety of factors, including (but not limited to) the current PC value, the history of the specific branch, and the global history all branches.

Correctly predicting if a branch will be taken or not taken results in performance improvements, as the pipeline does not need to be stalled while waiting for the branch result to actually be calculated. Incorrectly predicting a branch will only be found out after the branch result has been calculated (usually a few clock cycles). The "guessed" instructions that happened within that time frame would need to be discarded, as they would be inaccurate. The program will then have to move the PC to the correct location, and start executing the proper code; doing this may add even more delays to the pipeline, thus causing the program to run slower.

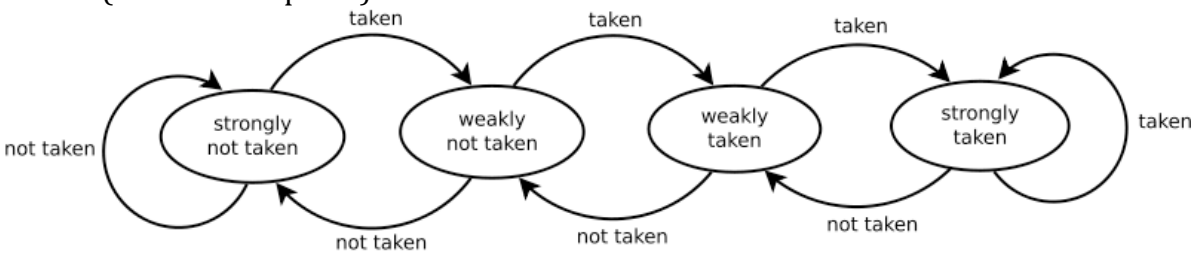
Branch prediction is an important mechanism in making programs run faster, and it is helpful (and very interesting!) to understand how the various algorithms of branch prediction work.

Design: Description of open-ended branch prediction

There was a lot of experimentation in coding the open-ended branch predictor, much of which actually made the predictions worse than the two-level 8-PHT predictor implemented earlier in the lab. (For example, a three-bit saturating counter is worse than a two-bit saturating counter, because it takes longer to "change its mind" about a prediction.) The best result was very similar to the two-level 8 PHT predictor, with two main differences: the open predictor has 128 PHTs, and the state machine was modified.

The open predictor was limited to 128Kbits in size. Having 128 tables of 2-bit saturating counters with a GHR of 9 bits allowed the predictor to take full advantage of the available space (128 tables * 2 bits * 512 entries/table = 128Kbits). Having more tables means that there will be less aliasing, as different PCs will have less change of "hashing" to the same table, and colliding with an existing entry. To select one of the 128 tables, 7 bits of the PC were used (bits 9:3). As in the two-level 8-PHT predictor, bits 2:0 of the PC were not used, as instructions were assumed to be 8-byte aligned. The GHR was used to select the line within the table.

The state machine was changed from what we learned in class so that if you were in a weakly not taken state and you see a taken branch, you would transition to a weakly taken state, rather than a strongly taken state. Likewise with weakly taken: if you see a not taken, you would transition to a weakly not taken state. The rest of the state machine remains the same. A diagram illustrating the branch predictor's state machine is as follows (source: Wikipedia)



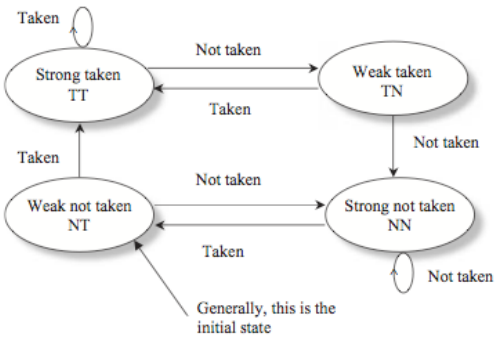
Methodology: How statistics were collected with the simulator

When a *F_COND* operation is seen, the *sim_num_br* variable is incremented. The variable keeps track of how many conditional branches are encountered. If the next PC (*regs.regs_NPC*) value is equal to the current PC value (*regs.regs_PC*) + 8 (where 8 is the size of the instruction in memory), it can be concluded that the branch is not taken. Otherwise, it can be concluded that the branch is taken.

For the **static predict-taken predictor**, every time a "not taken" is encountered, the variable *sim_num_mispred_static* is incremented.

With the **random predictor**, the expression *rand()%2* is used to see if the program should predict taken (1) or not taken (0). If it predicted taken but the branch is actually not taken, then *sim_num_mispred_random* would be incremented. Likewise if the expression predicts not taken but the branch is taken, the variable is also incremented.

Using the **2-bit saturating counter predictor**, the following state machine was implemented (taken from textbook, Figure 4.3):



Every time a branch is predicted incorrectly, *sim_num_mispred_2bitsat* is incremented. The state then changes, as the above diagram shows. In the implementation, state "weakly not taken" is set to 0b00, "strongly taken" is 0b01, "weakly taken" is 0b10 and "strongly not taken" is 0b11. The actual implementation uses integers to store these numbers for ease of programming, but only the last two bits are used.
There is an 8192-element array of integers that uses 12 bits of the PC (bits 14:3 only—bits 2:0 are not used, as instructions are 8-byte aligned) to index into it. There is a risk of aliasing if two instructions have the same 14:3 bits in the PC. What this means is that the branch predictor will not be able to tell these two addresses apart, and it treat the two branches as if it was the same branch. Another limitation is that when a branch is taken (or not taken) a small but predictable amount of the time (ie. if(i%3==0) in a for loop incrementing i) the 2-bit saturating counter will always say the branch is not taken (or taken). It will not take into account the pattern of taken and not taken that it sees.

The **two-level predictor** gets around this issue by using a global history register, as well as bits in the PC. In the implementation, there are 8 tables, with 512 entries each. The global history register (GHR) keeps track of the nine previous branches, if they were taken or not taken. The GHR indexes into the 512 entries, and the PHT is chosen by the three bits in the PC (bits 5:3). The 8 tables are all 2-bit saturating counters, implemented similar to the above (except with 512 rows, instead of 8192). The variable *sim_num_mispred_2level* is incremented when a misprediction occurs.

The **open predictor** is explained in the "Design" section.

Results: Branch misprediction statistics for each predictor

The following table summarizes the statistics for each algorithm with each benchmark, along as the average percentage misprediction for each algorithm.

	gcc.eio	go.eio	compress.eio	Average %
sim_num_br	43013065	62098422	8979180	
sim_num_mispred_static	21310422	28258389	4080979	
sim_br_static_ratio	49.54%	45.51%	45.45%	46.83%
sim_num_mispred_random	21508411	31045534	4489618	
sim_br_random_ratio	50.00%	49.99%	50.00%	50.00%
sim_num_mispred_2bitsat	5782333	15690458	1432348	
sim_br_2bitsat_ratio	13.44%	25.27%	15.95%	18.22%
sim_num_mispred_2level	5845461	16989330	884945	
sim_br_2level_ratio	13.59%	27.36%	9.86%	16.93%
sim_num_mispred_openend	3054565	9939510	855962	
sim_br_openend_ratio	7.10%	16.01%	9.53%	10.88%

It can be seen in the above table that the open-ended algorithm performs the best, mispredicting only 10.88% of the time across the three given benchmarks.

Conclusions: What we may conclude from the statistics

The algorithm for the open-ended solution is very similar to the algorithm to the two-level solution. The main difference was that the open-ended solution used much more memory, and it had a different state machine.

In experiments where the state machine was kept the same, but the memory was increased to 128Kbits, an 11.23% misprediction rate was encountered. A 16-fold increase in memory usage resulted in the misprediction rate dropping by 5.7%—we can see that this is not a linear drop. Depending on the architecture and how many clock cycles a misprediction wastes, there still may be a dramatic improvement between 16.93% and 11.23%.

With a different state machine but the same amount of memory, it was seen that the average misprediction rate was 16.48%. We can see that increasing the memory had more of an improvement than changing the state machine.

With both changes of the memory and the state machine, the misprediction rate dropped even more, from 16.93% down to 10.88%. We can see that the improvements are not added linearly, as shown below

memory_and_state_improvements < memory_improvements + state_improvements
(16.93-10.88) < (16.93-11.23) + (16.93-16.48)
6.05 < 6.15

It is helpful to keep a global history register, and not solely rely on PC values, as seen by the improvement in predictions between the two-bit saturated predictor and the two-level predictor. This allows for the predictor to detect "patterns" in some preceding branches of the program, rather than just predicting based on patterns seen in one specific branch.

The random predictor predicts correctly only half the time, regardless of the actual branches taken or not taken. This can be proven mathematically, where t=percentage of branches taken and (1-t) is branches not taken.

%mispredict = %guess_taken * %not_taken + %guess_not_taken + %taken
%mispredict = 0.5*(1-t) + 0.5(t) = 0.5 – 0.5t + 0.5t = 0.5

The t variable cancels out, and is therefore irrelevant in the calculation.

It can be seen that across the three benchmarks, there are more "taken" branches than not taken. "go.eio" also seems like the most difficult benchmark to predict if a branch is taken or not; it is consistently 2-3x harder to predict than "gcc.eio" or "compress.eio."