Gshare, Tournament, Customized Branch Predictor Implementation and Analysis

CSE240A: Principles or Computer Architecture

Jiawen Xu

September 30, 2023

Abstract

This is the final project report of CSE240A: *Principles of Computer Architecture*. In this project, we implement a Gshare, Tournament, and customized branch predictor. We implement our branch predictor with the framework provided by the class in C and evaluate the performance of three branch predictors with the traces provided.

1 Introduction

For this project, each entry of branch history table (BHT) is a bimodal counter with `weak taken`, `strong taken`, `weak not-taken`, and `strong not-taken`. The bimodal counter increments or decrements based on the resolved outcome feedback. The prediction of the branch predictor is based on the entry of branch history table.
Link for source code: https://drive.google.com/drive/folders/1CRS08U5HLgQTK_ACnOc9QmNCQ66GIElk?usp=sharing

1.1 G-share

G-share branch predictor where branch history table (BHT) is index by a combination of global history register and program counter. [2] A global history register is a shift register which records the pattern of the resolved branch result. A global branch predictor would have a BHT indexed by the (size saturated) global history register. However, simply using the global branch predictor results in lack of information from the program counter. Program counter identifies the instruction being executed. By using G-share branch predictor, we performs an `xor` operation on PC and global history register, which generally has higher performance than global branch predictor. [2] The advantage of G-share selector is that it combines both information from program counter and global history register. However, for superscalar or hardware-multithreading processor, the global history register may not reflects the actual execution sequences as multiple instruction may execute simultaneously. To resolve this problem, we might need to replicate global registers for each thread.

1.2 Tournament

Tournament branch predictor combines the local branch predictor and global branch predictor. Local branch predictor contains a history table indexed by saturated program counter whose value becomes the index of the branch history table. The entry of the local history table save the pattern mapped to the saturated PC value. The tournament would take one of the result between local and global(gshare) branch predictor if the current chosen branch predictor mispredicts but the other does not. With both branch predictor implemented, we can achieve a better prediction result from best of the both world, but one of the limit is the resource constraint. More hardware is needed for this implementation. If we have limited resources, we will need to reduce the table size to meet the hardware constraints.

1.3 Custom Predictor

The custom predictor we are implementing is a voting branch predictor from g-share, tournament, and perceptron branch predictor.[1] The custom predictor will select the majority of the prediction from three branch predictor. The G-share and tournament branch predictor are the same as described above. We add a perceptron branch predictor structure in our implementation.[1]

The perceptron branch predictor uses a list of weights and bias to predict the branch. The size-saturated PC and global history register will perform a `xor` operation which index to the perceptron table. We call this index *hashed index*. The perceptron table contains the weights and bias for hashed index. The perceptron branch predictor calculates the dot product of the perceptron and the global history which produces the *score* value. If the score is greater than 0, we will predict `taken`; otherwise we will predict `not-taken`. We update the perceptron table whenever misprediction happens or the previously calculated score is less than a threshold value. The details will be covered in implementation section.

The benefit of using a voting system for three branch predictors is that it can higher the accuracy of branch prediction as the tolerance of one branch predictor making wrong prediction is higher. However, the disadvantage is the hardware resources limitation. It will need more hardware/circuits for this implementation especially the perceptron branch predictor.

2 Implementation

2.1 Bimodal Counter

The first item we implements is the bimodal counter. A bimodal counter is an entry of the branch history table, where it is updated with feedback resolved branch. The prediction depends on the current state of the bimodal counter. A bimodal counter has four states: `strongly taken (ST)`, `weakly taken (WT)`, `strongly not-taken (SN)`, `weakly not-taken (WN)`. The transition of the finate state machine (FSM) in this counter is shown as:

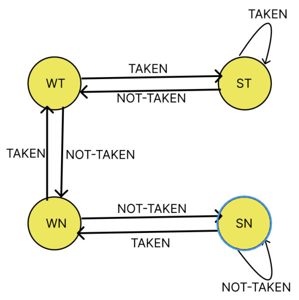

Figure 1: State Diagram of Bimodal Counter

2.2 Shift Register (Global History Register)

Another implementation is the shift register. The shift register will be updated when feedback of the actual outcome is received. The shift register in our code is a 32-bit data type integer where each bit represents `TAKEN` (1) and `NOTTAKEN` (0). To update the shift register, we simply do

```
(global_register << 1) + outcome
```

For xor operation, we only need to do

```
global_register ^ program_counter
```

which performs a bit-wise xor operation in C. Doing this will reduce the computational complexity as well as improving the memory utilization. As both global register and program counter are 32 bit in our framework, it is easier to implement this way as well.

2.3 Gshare Branch Predictor

The Gshare branch predictor will be an application of the bimodal counter and the global register introduced in section 2.1 and 2.2, and its architecture is shown as below:

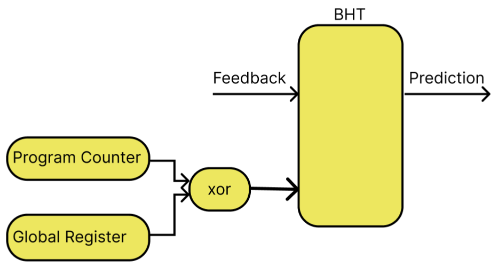

Figure 2: Architecture of Gshare branch predictor

Each entry of the BHT is a bimodal counter. The feedback is used for updating/training the BHT which is previously resolved branch outcome. In the training process, the bimodal counter in the BHT will transition to next state based on the feedback outcome. As well, The global register is updated during the training process. The predicting process is simple as it only needs to read the mapped bimodal counter and return the prediction based on the state. Note that the training (with feedback) and predicting process will use different program counter. The illustration didn't distinguish the two program counters (training and predicting) just for simplicity and readability.

2.4 Local Branch Predictor

For local branch predictor, we add a local history table (LHT) where each entry is a shift register described in 2.2. The output of the LHT becomes the index of the BHT which produCes the prediction. The architecture of the local branch predictor is shown as below:

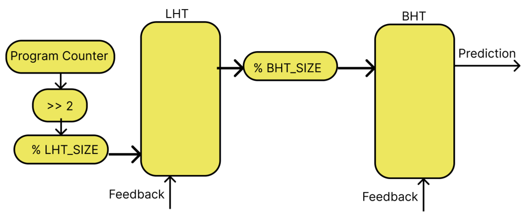

Figure 3: Architecture of Local branch predictor

For the program counter, we do not consider first 2 least significant bits because the program counter generally increments by 4. Then we saturate the index with our local history table size. The output of the local history table will be saturated with the branch history table size and map to the bimodal history table to produce the prediction. The training process is similar. The entry of LHT is a shift register which will be updated by a left shift and addition of the outcome. The entry of the BHT is a bimodal counter which will be transitioned to the next state based on the feedback outcome.

2.5 Tournament Branch Predictor

The tournament branch predictor is a combiantion of the Gshare branch predictor in section 2.3 and Local branch predictor in section 2.4:

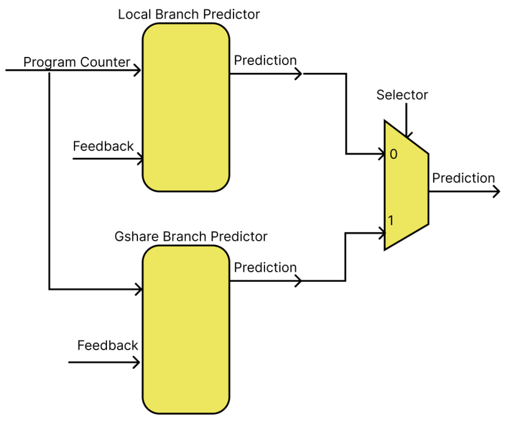

Figure 4: Architecture of Tournament branch predictor

The selector in is updated every time when misprediction occurs. Two branch predictors are working (predicting and training) simultaneously regardless if it is selected.

2.6 Perceptron Branch Predictor

The perceptron branch predictor is used in our customized branch predictor. The architecture is shown as below:

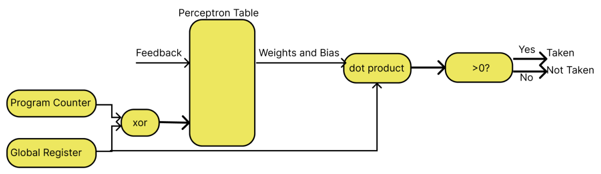

Figure 5: Architecture of Perceptron branch predictor

As shown in the illustration, the perceptron branch predictor uses a perceptron table whose entries are vectors of weights and bias. The logic combination of program counter and global history register will map to the perceptron table and get a vector of weights and bias. Then we will perform dot product with this vector and the global history register. The dot product, or the *score*, will decide the prediction of the perceptron branch predictor. If the score is greater than 0, return `TAKEN`. Otherwise, return `NOTTAKEN`.

For the training process, the perceptron is updated when a mispredict occurs, and the weight will be updated based on the global history register:

```
if (mispredict || step < threshold):
  if (outcome == TAKEN):
    bias++
  else:
    bias--
  for weight[i] in weights:
    if (outcome == global_register):
        weight[i]++
    else:
        weight[i]--
```

Above is the pseudocode for training. In addition, we saturate all the weights and bias with maximum value 127 and minimum value -128.

2.7 Voting (Custom) Branch Predictor

Our voting branch predictor combines the three branch predictors - gshare, tournament, and perceptron.

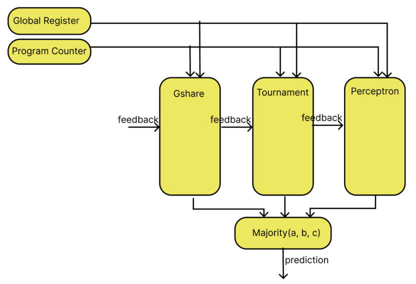

Figure 6: Architecture of Voting (Custom) branch predictor

This voting perceptron will take the majority of the three branch predictors as the final prediction. Doing this can effectively reduce the prediction, but at the same time it increases the hardware requirements which can result in higher power or area for the processor.

3 Observation

Misprediction Rate (%)	fp_1.bz2	fp_2.bz2	int_1.bz2	int_2.bz2	mm_1.bz2	mm_2.bz2
static	12.128	42.350	44.136	5.508	50.353	50.353
gshare	0.842	1.500	13.900	0.426	6.523	6.523
tournament	1.264	3.710	16.798	0.587	6.073	6.073
custom	0.991	3.703	11.789	0.462	5.532	5.532

Table 1: Misprediction Rate result four 4 Branch Predictor Implementation and 6 traces

Table 1 shows the performance of our branch predictors implementation. Static branch predictor always return `TAKEN` for its prediction, so it has the worst performance. We treat it as our baseline.

Gshare branch predictor is run with branch history table size of $2^{13} = 8192$ entries. We use a `uint8_t` data type for the bimodal counter. Totally it takes $8192 * 8 = 65536$ bits = 64KB.

Tournament branch predictor is run with local history table size of $2^{10} = 1024$ entries and branch history table size of $2^9 = 512$ entries. We use a `uint8_t` data type for the bimodal counter and a `uint32_t` data type for local history table entry. Totally it takes $1024 * 32 + 512 * 8 = 36864$ bits = 36KB.

Custom branch predictor is run with local history table size of $2^9 = 512$ entries and branch history table size of $2^9 = 512$ entries. Additionally, the perceptron table size is $512*10 = 5120 entries$. We use a `uint8_t` data type for the bimodal counter and a `uint32_t` data type for local history table entry. Each perceptron entry is `uint8_t` data type. Totally it takes $512*32+512*8+5120*8 = 61440$ bits = 60KB.

In summary, here is the memory resources used for tables in each implementaion:

Implementation	Table total sizes
gshare	64KB
Tournament	36 KB
Custom	60 KB

Table 2: Memory Usage for Each Branch Predictor Implementation

4 Results and Conclusion

From the section 3: observation, we can see our three branch predictors perform much better than the baseline branch predictor. Our gshare and tournament have similar performance; however we distribute less memory for the tournament branch predictor. In other word, if we distribute more memory to the tournament branch predictor or enlarge the table size in the tournament branch predictor, it may surpass the performance of the gshare branch predictor. We actually try making the tournament same size as the gshare branch predictor and it shows the tournament branch predictor performs much better than the gshare branch predictor.

For our custom branch predictor, it beats the tournament branch predictor for all 6 traces, and it beats 3 of the gshare branch predictor with the limited memory resources available. Although the previous research shows that the perceptron branch predictor actually does not have performance as good as gshare or local [1], it is still a good direction of exploring new method for branch predicor especially nowadays when artificial interlligence and deep learning is developed rapidly. For furture development, developing a fully connected neural network with each layer maintaining the perceptron could be a good option.

References

[1] D.A. Jimenez and C. Lin. "Dynamic branch prediction with Perceptrons". In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture* (). DOI: 10.1109/hpca.2001.903263.

[2] Scott McFarling. *Combining branch predictors - HP labs.* URL: https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf.