Neural method for dynamic branch predictor

Xin Du
University of Florida

Abstract

Branch instructions account for a large proportion in the whole instructions which is definitely an essential part of modern microarchitecture. If there is no strategy for branch prediction, there will be many stalls when a branch is encountered, which will fatally decrease the efficiency of programs. The branch prediction will benefit more when pipelines deep and the number of instructions issued per cycle increases. Hence, a good and fast branch prediction plays a significant role in increasing the accuracy and speed of instruction execution.

The most commonly used method in branch prediction is two level prediction with branch history table and 2-bit counter. Although it can be implemented very well in practice, this method of 2-bit counter will restrain the length of branch history table, resulting in its limit to increase the branch prediction accuracy. So, a new method called neural network for dynamic branch prediction has been put forward. The goal of this paper is to explore the possibility of using neural branch predictors instead of 2-bit counters in the second level prediction to predict branch outcomes with an attempt to increase the accuracy of branch prediction. We designed and evaluated hardware implementations of simplified neural branch predictors and the result were obtained running Spec2000 benchmarks on the simple-scalar architecture simulator. This approach is valuable and practical since it improves the design of hardware branch predictors.

Introduction

In order to achieve the high speed in computers, people are dedicated to reducing the pipeline stalls in instruction executions. Out of order instruction processing is used in modern computers. Since instructions are being fetched and executed out-of-order, it is important that the next fetched instruction is exactly the instruction that would be fetched when a branch instruction is encountered. Branch prediction is used to reduce the stalls in the computer for the reason that branch predictor can predict branch target address in IF stage. In this way, the predicted instruction can be fetched in ID stage of the branch instruction no matter the prediction is actually right or not. If the prediction is right, there will be no stalls; if the prediction is wrong, the fetched instruction that should not be fetched will be flushed and the program restarts in the right instruction. According to Schlansker et al., "branch prediction is the process of correctly predicting whether branches will be taken or not before they are actually executed". Therefore, correct prediction can avoid stalls used for waiting for the final result of branch behavior and keep the pipeline as busy as possible by prefetching the instruction that has been predicted. Hence, a good predictor with high accuracy is meaningful in branch prediction since more stalls can be reduced resulting in high speed of computer. In this paper, we come up with a method that can be used as a good dynamic branch predictor: neural branch predictor. First, we review the basic knowledge about two level branch predictor and 2-bit counters. Then, we introduce the neural branch predictor with perceptron-based algorithm and explain why we choose the perceptron as a new branch predictor. Finally, we show the implementation of neural branch predictor in simple-scalar.

Two level branch predictor

Branch predictors that use the behavior of other branches to make a prediction are called two level predictors (correlating predictors). The first level consists of a branch history register (BHR, also called branch history table, BHT) that records the outcomes of last n branched encountered. There are two types of HR. One type is a single global history register (BHRG) which keep track of the outcome of most recent n branches in the instruction stream. Figure 1 shows the two level global history predictor. The other type is several local history registers (BHRL) which records the last n outcomes of each branch. Figure 2 shows the two level local history predictor. The second level is pattern history table (PHT) which records the information about history of branches to predict later actions, namely, taken or not taken, when corresponded branches are taken again. The first level generates an index used in the second level to access PHT. The basic method of indexing is using the information in BHR to find the corresponding 2-bit counters in PHT to predict taken or not taken. However, the outcomes of branch are also correlated with other branches' outcomes. So Yeh and Yale N. Patt come up with a two-level adaptive branch prediction, which is Gshare branch predictor, shown in Figure 3: A popular scheme in branch prediction uses a pattern history table (PHT) which contains two-bit saturating counters indexed by a combination of branch address and global or per-branch history. [1]

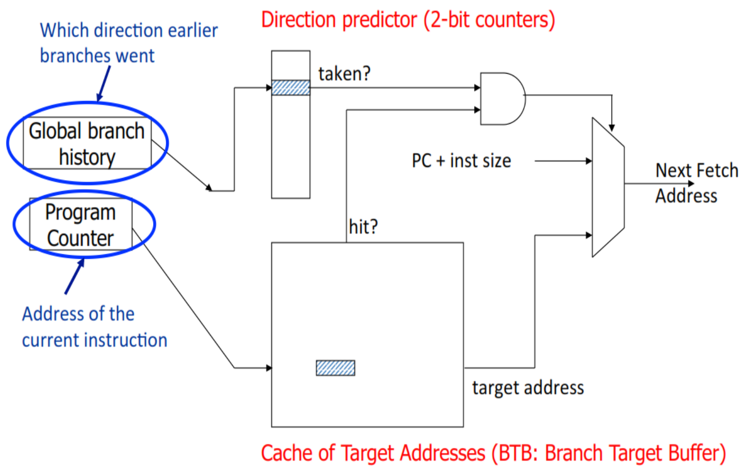
Which direction earlier branches went
Direction predictor (2-bit counters)
taken?
Global branch history
PC + inst size
Next Fetch Address
Program Counter
hit?
Address of the current instruction
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Figure 1. Two-Level Global History Predictor

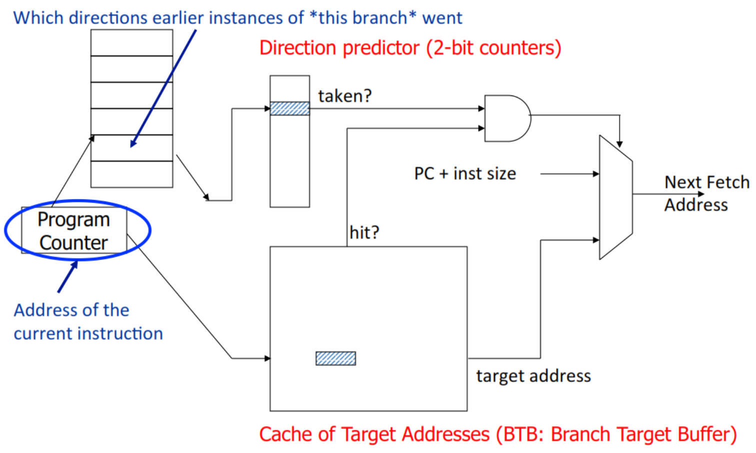
Which directions earlier instances of *this branch* went
Direction predictor (2-bit counters)
taken?
PC + inst size
Next Fetch Address
Program Counter
hit?
Address of the current instruction
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Figure 2. Two-Level Local History Predictor

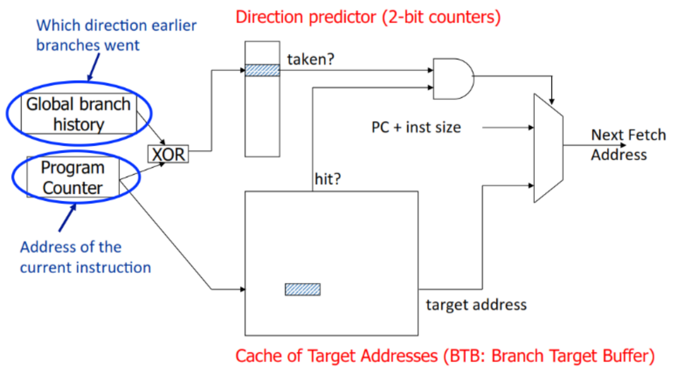
Which direction earlier branches went
Direction predictor (2-bit counters)
taken?
Global branch history
XOR
PC + inst size
Next Fetch Address
Program Counter
hit?
Address of the current instruction
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Figure 3. Two-Level Local Gshare Predictor

As for branch target buffer (BTB), it stores the predicted address for the next instruction after a branch. In IF stage, the PC of the instruction is matched against a set of instruction stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch (for not taken branches the target is simply PC+4). In the second field, predicted PC, contains the prediction for the next PC after branch. The predicted PC can be known before the current instruction is decoded and therefore enables fetching to begin after IF-stage. The third field, which is optional, may be used for extra prediction state bits. [2]

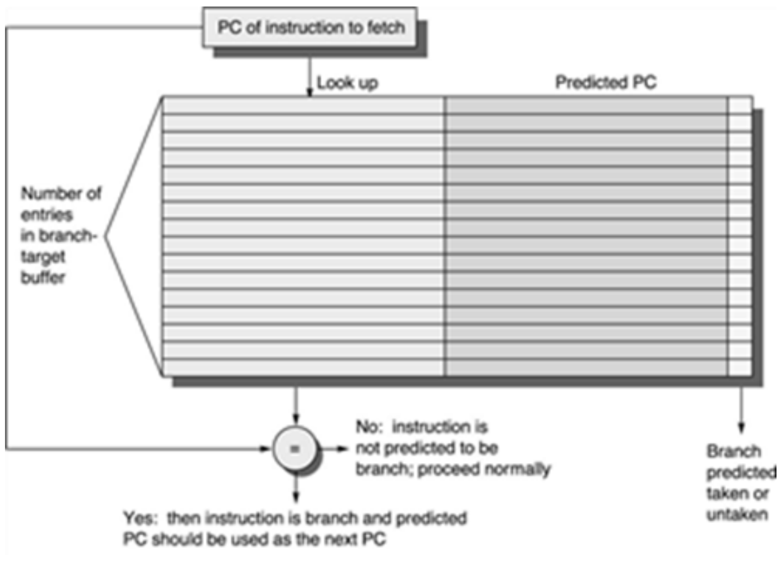
PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Figure 4. Branch Target Buffer (BTB)

In the conventional two level branch predictors, PHT is implemented by 2-bit counters. In this paper, we introduce a new method: neural branch predictor. We replace the 2-bit counters with neural branch predictor to explore its performances and compare them with the 2-bit counter.

1. 2-bit counters

2-bit counters is one conventional method to achieve PHT, which keeps track of the prediction history. The PHT consists of an array of two bit saturating counters indexed by the GHR to obtain the prediction. A prediction must miss twice before it is changed in this method. With a k-bit BHRG, 2k entries are required if a global PHT is provided. If it is a number of BHRL with k-bit, (the number of BHRL)* (2k) entries are required to point to PHTs provided for each branch. Obviously, the number of 2-bit counters grow exponentially with the branch history table length which in return limits the length of BTB. Figure 4 shows how 2-bit counters is achieved in a state diagram.

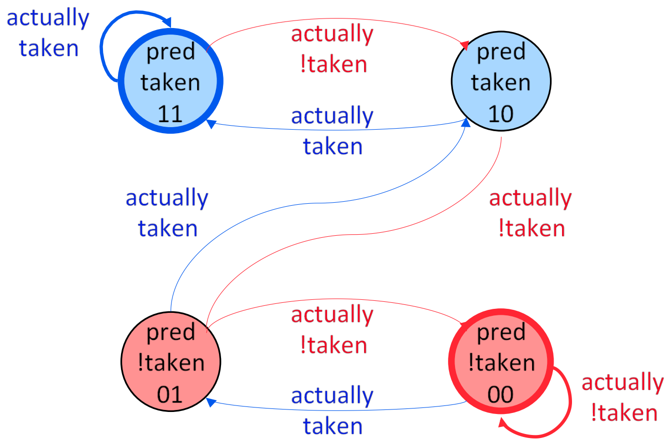

Figure 5. Two-Bit Prediction Method States

2. *neural branch predictor*

Algorithm of perceptron-based branch predictor

The method of neural network is a part of machine learning. The basic concept is that data are imported into the input unit neurons and propagate through the whole network and are exported in the output neurons. [3] We use the result from the output to make the decision in some problems. In the basic algorithm of perceptron-based branch predictor, there are two inputs: 1 and vector X. The branch history containing vector X uses 1 to represent taken and −1 to represent not taken. Simultaneously, there is a weight related to each input, which is vector W. Each weight represents the degree of correlation between the behavior of a past branch and the behavior of the branch being predicted. [3] Weights are bipolar. Positive weights represent positive correlation, and negative weights represent negative correlation. The predicting direction depends on the resulting sum Y which is calculated by the following formula:

$$Y=W0+X1W1+X2W2+X3W3+\ldots+Wm$$

We set the threshold of Y to be 0. If the resulting sum Y is positive, we predict taken; otherwise we predict not taken. Fig.1 shows the basic algorithm of perceptron-based branch predictor.

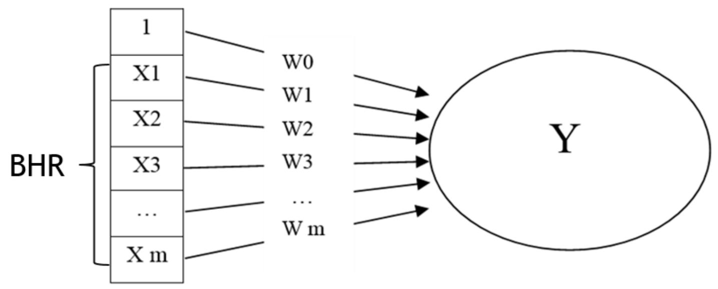

Figure 6. Basic Perceptron-Based Algorithm

The weights for each branch are put into a table called perceptron table which is indexed by the address of the current branch by using the hash function. After selecting a set of weights in this table, these weights are multiplied by each bit in BTB to export Y so as to make the predictions.

The training algorithm

Just like programs need to be debugged after being written, the weights in neural network also needs to be tuned after being set in the first time. This process is called training. Neural networks learn to compute a function using example inputs and outputs. Each training process will revise weights once. The perceptrons are trained by an algorithm that increments a weight when the branch outcome agrees with the weight's correlation and decrements the weight otherwise. That is if its corresponding branch is taken, we add the weight; otherwise we subtract the weight. [3] The correlation between inputs and outputs are adjusted gradually. So the more examples there are, the better solution will be and the higher accuracy will be achieved. In this sense, the neural network seems suitable for branch prediction since computer handles millions of instructions per second which provides multiple examples to be used to tune weights.

The main advantage of the neural predictor is the perceptron tables grow linearly with the length of BTB, so the neural branch predictor just occupy smaller size of memory compared with 2-bit counter. Higher accuracy prediction can be achieved by increasing the length of BTB. However, there are some disadvantages of this method: the high latency, since during this process, multiple calculations need to be done compared to the 2-bit counters.

Simulation using simple-scalar

1. *Simulation outlines*

Simple-Scalar has a modular layout which allows for great versatility. Components can be added or modified easily. Therefore, we can add some codes to achieve the function of neural branch predictor. Figure 7 shows the software structure for Simple-Scalar.

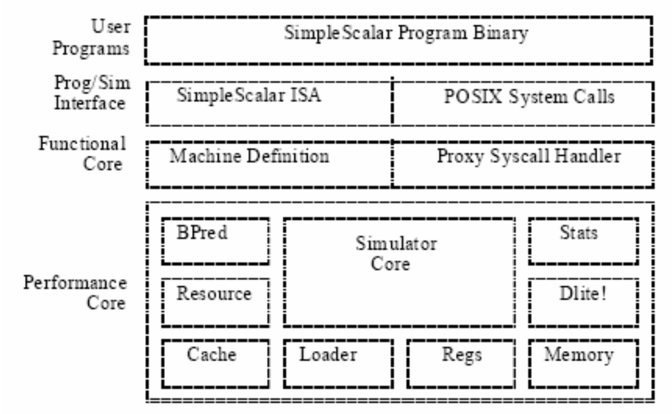

Figure 7. Simple-Scalar Software Architecture

2. *Implementation of perceptron prediction*

There are four steps to achieve the goal of neural branch predictor:

1. Develop and design the perceptron-based algorithm and training algorithm.
2. Modify the simple-scalar simulator to achieve neural branch predictor.
3. Train the neural branch predictor.
4. Evaluate the neural branch predictor.

The neural branch predictor is achieved by perceptron-based algorithm and training algorithm. For the selection of branch is achieved by switch statement. Hence, we only need to add a case of perceptron-based branch predictor so that to achieve our goal of changing predictor. The function that needs to be achieved in simple-scalar is showed as Figure 8.

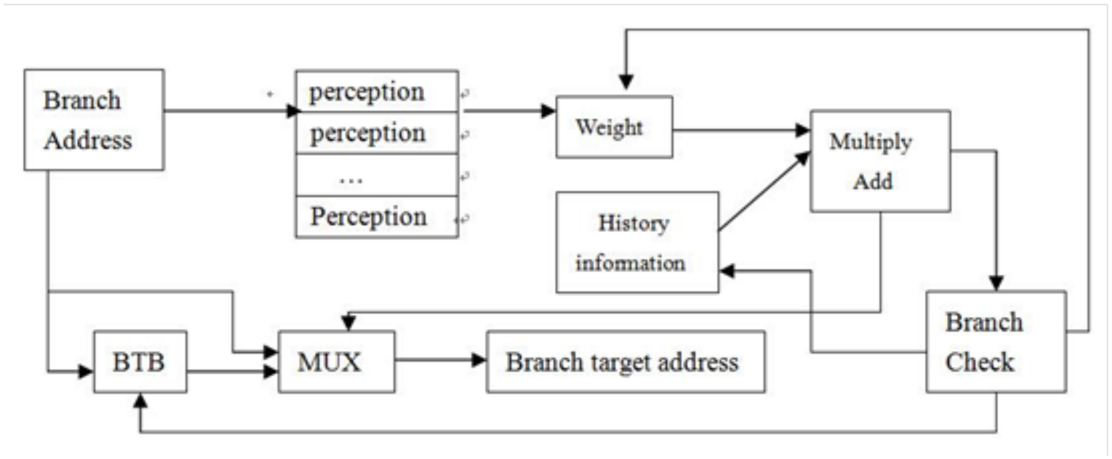

Figure 8. Achievement of Perceptron-Based Algorithm in Simple-scalar

The related files containing the branch prediction in the simple-scalar are sim-bpred.c, bpred.c, bpred.h. Sim-bpred.c is a main function that shows the five steps in pipeline with the branch prediction. Bpred.c contains some functions being called by main function in Sim-bpred.c. Bpred.h contains some necessary definitions of functions in bpred.c.

In the main function of sim-bpred.c, there are five standard steps to execute instructions: IF, ID, EXE, MEM, WB. The function of prediction is inserted in EXE stage to figure out whether the prediction is right or wrong. There are three parts in branch prediction: 1. Create BTB, BHR, perceptron table. 2. Prediction of branch instructions and decide whether the branch is taken or not. 3. Updating the contents in BTB, BHT and PHT.

The first part is contained in bpred_create function. We create a new branch history table, a perceptron table and a bias weight in this function. We do not create BTB since we use the previous BTB in the simple-scalar to achieve its function. We create a single array of changeable length for BHR and 1024 lines of arrays with the same length as BHR.

The second part is contained in the bpred_lookup function. There are 6 types of strategies of branch prediction: BPredComb, BPred2Level, BPred2bit, BPredTaken, BPredNotTaken, BPred_NUM. We can choose any type of these six to achieve branch prediction. What we need to do is replacing 2-bit counters with our perceptron table and evaluate its performance. To be specifically, we use the branch address to index to the perceptron table to select a specific line for weights and use all bits of weights to be multiplied by their corresponding bits in BHT then add them together. The result of sum needs to compare with the threshold, such as 0 in our experiment, to make the final decision of taken or not taken. If the sum is above 0, the prediction is taken; if the sum is below 0, the prediction is untaken. After deciding the result of prediction, this function will return the target address for main function if the prediction is taken; return 0 for main function if the prediction is untaken.

The third part is contained in bpred_update function. There is some information that has to be changed after the actual execution of instructions, such as BTB, BHR, perceptron table and so on. For neural branch predictor just changed the second level, the BTB in simple-scalar remains same in this new method. Thus, in this stage, we only need to revise BHR, perceptron table after the actual execution of instructions every time. First, revise the perceptron table. When the prediction is correct, we do nothing. Only when the prediction is wrong, I revise the value. If branch outcome is correct, add some values to the weights which correspond to 1 in BHR and minus some values to weights which correspond to -1 in BHR. If branch outcome is wrong, minus some values to the weights which correspond to 1 in BHR and add some values to weights which correspond to -1 in BHR. Then, revise BHR. BHR needs to be shift left a bit each time and add 1 or -1 in the last bit of the BHR according to the actual outcome of the branch behavior.

Evaluate the neural branch predictor

I evaluated the performance of the neural branch predictor in 2 different ways.

First, I compared different types of branch predictor on six different benchmarks of Spec 2000, as shown in figure 9. As we can see that in equake, crafty, vortex and lucas, neural branch predictor outperform than other types of predictors.

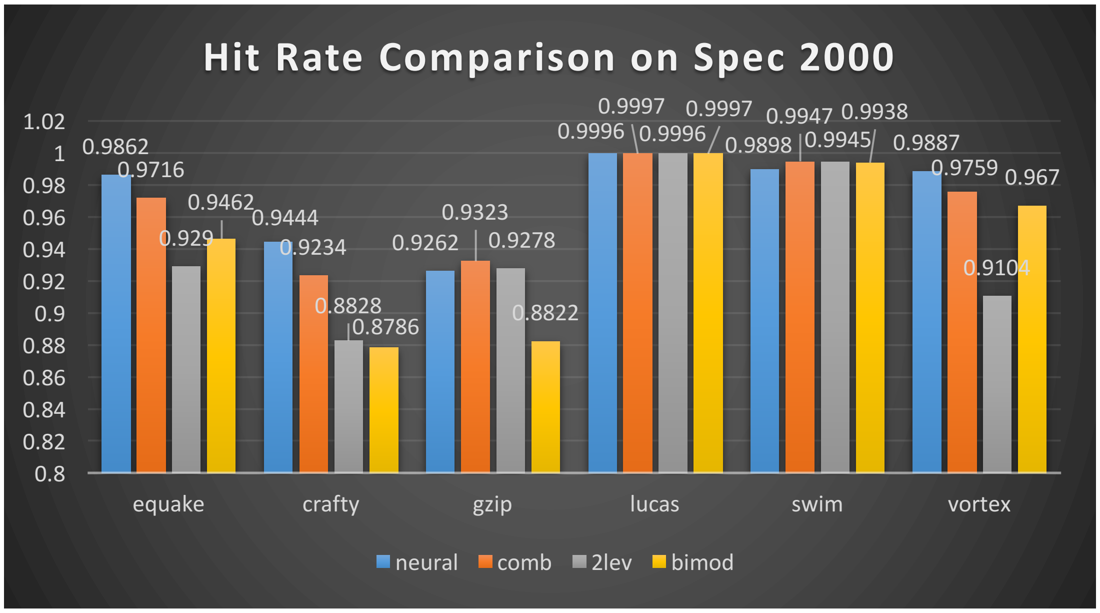

Figure 9. Hit rate comparison on Spec 2000 in simple-scalar

Second, I compared performance of neural branch predictor on Spec2000 with different history register length, which is shown in figure 10. As we can see that longer the history length it is, the better performance it will achieve. However, it is more obvious when history length is below 70.

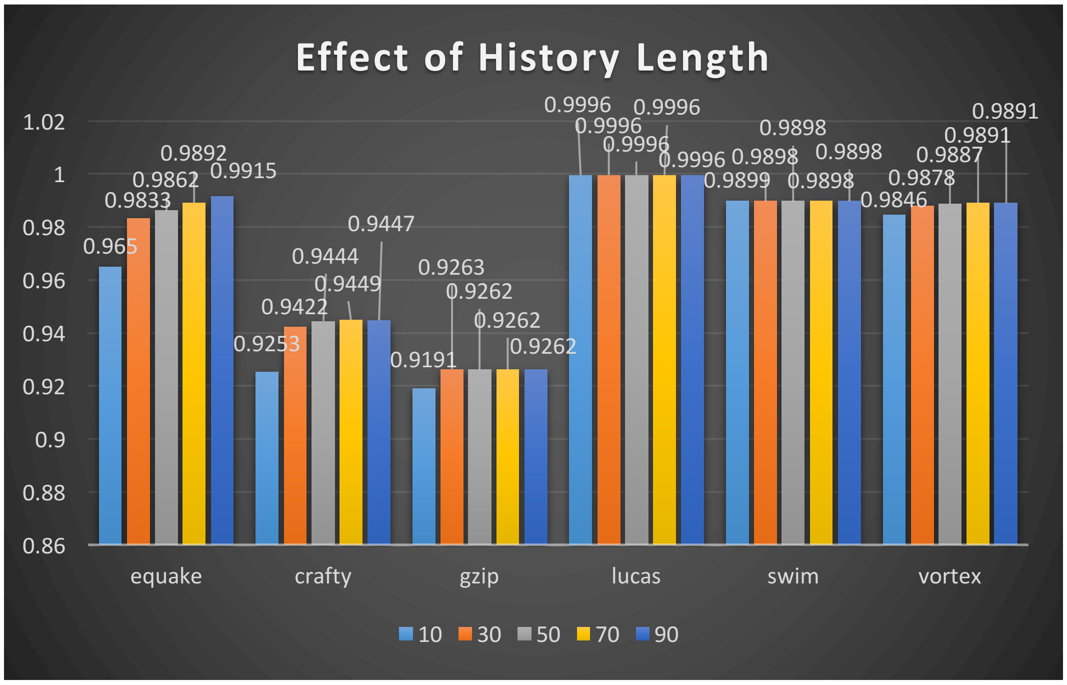

Figure 10. Performance comparison on Spec 2000 with different history length

As you can see from the chart, our neural branch predictor is as good as we predicted.

Conclusion

This paper introduces a new method to achieve the dynamic branch prediction and aims to implement the neural branch predictor on simple-scalar and compare the performance between different dynamic branch predictor. The key advantage of this type of branch predictor is its ability to use long history lengths with linear resources. After the comparison, we can see that neural branch predictor can achieve the higher performance than other types of branch predictor, which is meaningful and practical for the design of branch predictors.

Challenge and solutions

I did not familiar with neural network and simple-scalar when I selected this topic as my project so I spent much time reading lots of materials on them in order to have a good understanding. Reading codes in simple-scalar is also a hard work for me. To do the work as fast as possible, I focused on the related codes that I needed. Writing codes is also skillful. There were some bugs and I tried my best to handle them by searching materials and asking students for some help. Besides, the result of evaluation is not very good at beginning, which is only about 0.6. After checking codes, I found the implementation of training algorithm is wrong. After revising the codes, the result became good enough.

Acknowledgment

I would like to thank Prof. Tao Li for his inspiring class and his guidance during this project. I also thank those students who share some experiences and who join the discussion with me on this topic.

Reference

[1] T.-Y. Yeh and Yale N. Patt. Two-level adaptive branch prediction. In Proceedings of the 24th ACM/IEEE Int'l Symposium on Microarchitecture, November 1991.

[2] John L. Hennessy, David L. Patterson computer architecture: a quantitative approach 5th, 2012.

[3] DANIEL A. JIME´ NEZ and CALVIN LIN. Neural Methods for Dynamic Branch Prediction, 2002.

[4] P.B. Osofisan, Ph.D. and O.A. Afunlehin, M.Sc. (Eng.) Application of Neural Network to Improve Dynamic Branch Prediction of Superscalar Microprocessors. The Pacific Journal of Science and Technology. Number 1. May 2007.

[5] Gordon Steven, Rubén Anguera, Colin Egan, Fleur Steven and Lucian Vintan. Dynamic Branch Prediction using Neural Networks.

Appendix

In the sim-bred.c

```
1. else if (!mystricmp(pred_type, "neural"))
   {
     if (bimod_nelt != 1)
           fatal("bad bimod predictor config (<table_size>)");
     if (btb_nelt != 2)
           fatal("bad btb config (<num_sets> <associativity>)");

2.else if (!mystricmp(pred_type, "neural"))
   {
     if (bimod_nelt != 1)
           fatal("bad bimod predictor config (<table_size>)");
     if (btb_nelt != 2)
           fatal("bad btb config (<num_sets> <associativity>)");


     pred = bpred_create(BPredneural,
                               /* bimod table size */bimod_config[0],
                               /* 2lev l1 size */0,
                               /* 2lev l2 size */0,
                               /* meta table size */0,
                               /* history reg size */0,
                               /* history xor address */0,
                               /* btb sets */btb_config[0],
                               /* btb assoc */btb_config[1],
                               /* ret-addr stack size */ras_size);
   }
```

In the bpred.c

```
1.case BPredneural:
    pred->dirpred.bimod =
    bpred_dir_create(BPred2bit, bimod_size, 0, 0, 0);
    break;
2.case BPredneural:
   {
    pred->neuralstruct.biasinputweight=50;
    int i;

    /* allocate BTB */
    if (!btb_sets || (btb_sets & (btb_sets-1)) != 0)
        fatal("number of BTB sets must be non-zero and a power of two");
    if (!btb_assoc || (btb_assoc & (btb_assoc-1)) != 0)
        fatal("BTB associativity must be non-zero and a power of two");

    if (!(pred->btb.btb_data = calloc(btb_sets * btb_assoc,
                                      sizeof(struct bpred_btb_ent_t))))
        fatal("cannot allocate BTB");

    pred->btb.sets = btb_sets;
    pred->btb.assoc = btb_assoc;

    if (pred->btb.assoc > 1)
        for (i=0; i < (pred->btb.assoc*pred->btb.sets); i++)
         {
          if (i % pred->btb.assoc != pred->btb.assoc - 1)
            pred->btb.btb_data[i].next = &pred->btb.btb_data[i+1];
          else
            pred->btb.btb_data[i].next = NULL;

          if (i % pred->btb.assoc != pred->btb.assoc - 1)
            pred->btb.btb_data[i+1].prev = &pred->btb.btb_data[i];
         }

    /* allocate retstack */
    if ((retstack_size & (retstack_size-1)) != 0)
        fatal("Return-address-stack size must be zero or a power of two");

    pred->retstack.size = retstack_size;
    if (retstack_size)
        if (!(pred->retstack.stack = calloc(retstack_size,
                                            sizeof(struct bpred_btb_ent_t))))
          fatal("cannot allocate return-address-stack");
    pred->retstack.tos = retstack_size - 1;
```

```
     break;
   }
3.case BPredneural:
   fprintf(stream, "pred_dir: %s: predict taken\n", name);
   break;
4.case BPredneural:
   bpred_dir_config (pred->dirpred.bimod, "neural", stream);
   fprintf(stream, "btb: %d sets x %d associativity",
             pred->btb.sets, pred->btb.assoc);
   fprintf(stream, "ret_stack: %d entries", pred->retstack.size);
   break;
5. case BPredneural:
     {
      index = (baddr >> MD_BR_SHIFT) & (pred->btb.sets - 1);
      int ii=0;
       while(ii<=hislen-1)
         {
sumsum=sumsum+pred->neuralstruct.history[ii]*pred->neuralstruct.weight[index][ii];
          ii++;
            }

      sumsum=sumsum+pred->neuralstruct.biasinputweight;
      break;
     }
6.if(pred->class==BPredneural)
{
   if (pbtb == NULL)
   {
     /* BTB miss -- just return a predicted direction */
     return ((sumsum>= 0)
             ? /* taken */ 1
             : /* not taken */ 0);
   }
  else
   {
     /* BTB hit, so return target if it's a predicted-taken branch */
     return ((sumsum >= 0)
             ? /* taken */ pbtb->target
             : /* not taken */ 0);
7. if(pred->class == BPredneural)
{
      /*Modify weight table by parameter correct*/
      if(correct==1)
        {
            ;
            }
       else{

  //modify the weight table
            int i=0;
            while(i<=hislen-1)
            {
if(taken)
{
            //if taken, then for every +1 in history table,we add 1 to to the corresponding weight
in the weight table,and biasinputweight+1.
            if(pred->neuralstruct.history[i]==1)
            pred->neuralstruct.weight[(baddr >> MD_BR_SHIFT) & (pred->btb.sets -
1)][i]=pred->neuralstruct.weight[(baddr >> MD_BR_SHIFT) & (pred->btb.sets - 1)][i]+1;
            if(pred->neuralstruct.history[i]==-1)
pred->neuralstruct.weight[(baddr >> MD_BR_SHIFT) & (pred->btb.sets -
1)][i]=pred->neuralstruct.weight[(baddr >> MD_BR_SHIFT) & (pred->btb.sets - 1)][i]-1;
            pred->neuralstruct.biasinputweight+=1;

 }
else{       //if not taken, then for every +1 in history table,we minus 1 to to the corresponding
weight in the weight table,and biasinputweight-1.

 if(pred->neuralstruct.history[i]==1)
            pred->neuralstruct.weight[(baddr >> MD_BR_SHIFT) & (pred->btb.sets -
1)][i]=pred->neuralstruct.weight[(baddr >> MD_BR_SHIFT) & (pred->btb.sets - 1)][i]-1;
            if(pred->neuralstruct.history[i]==-1)
pred->neuralstruct.weight[(baddr >> MD_BR_SHIFT) & (pred->btb.sets -
1)][i]=pred->neuralstruct.weight[(baddr >> MD_BR_SHIFT) & (pred->btb.sets - 1)][i]+1;
pred->neuralstruct.biasinputweight+=-1;
}
            i++;
            }
           }
       /*Modify the history table*/
       if(taken==1)
          { //shift
           //put +1
          int j=0;
           while(j<=hislen-2)
            {  pred->neuralstruct.history[j]=pred->neuralstruct.history[j+1];
j++;
              }
            pred->neuralstruct.history[j]=1;
             }
        else
         {    // put -1;
            int j=0;
            while(j<=hislen-2)
               {  pred->neuralstruct.history[j]=pred->neuralstruct.history[j+1];
j++;
                  }
            pred->neuralstruct.history[j]=-1;
              }

}
```

In bpred.h

```
1.struct{
            int biasinputweight;
            int history[hislen];
            int weight[1024][hislen];
 }neuralstruct;
```