# Evaluation of Dynamic Branch Prediction Technics

Χiaobei Yu(xyu5@gwu.edu)

**Abstract**

Branch prediction is an interesting area due to its impact on the efficiency of pipelined and superscalar processor. There are various methods proposed to predict the path of an instruction stream after a branch. This paper evaluates the accuracy of predicting by five dynamic branch predictors. The result shows that the neural predictor has the highest prediction accuracy.

**1. Introduction**

In a highly parallel computer system, branch instructions can break the flow of instruction fetching, decoding and execution. This results in delay, because the instruction issuing must often wait until the actual branch outcome is known. To make things worse, the deeper the pipelining is, the greater performance loss is. (software-based and hardware-based Branch Prediction Evaluation Strategies and Performing Evaluation) Brach prediction overcomes the fetching limitation by control hazards in order to expose instruction level parallelism. If branch prediction rates are high enough, the processor will likely to have a better overall performance. Without branch prediction, such a processor must stall whenever there are unresolved branch instructions. Since on average 20 percent of instructions are branches, this imposes a substantial penalty on the performance of such processors. (Dynamic Branch Prediction) The work uses selected benchmarks from the Stanford to compare several well known branch prediction schemes.

**2. Static Branch Prediction Strategies**

Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing. Instead it predicts the outcome of a branch based solely on the branch instruction.

The early implementations of SPARC and MIPS (two of the first commercial RISC architectures) used single direction static branch prediction: they always predicted that a conditional jump would not be taken, so they always fetched the next sequential instruction. Only when the branch or jump was evaluated and found to be taken did the instruction pointer get set to a non-sequential address.

Both CPUs evaluated branches in the decode stage and had a single cycle instruction fetch. As a result, the branch target recurrence was two cycles long, and the machine would always fetch the instruction immediately after any taken branch. Both architectures defined branch delay slots in order to utilize these fetched instructions.

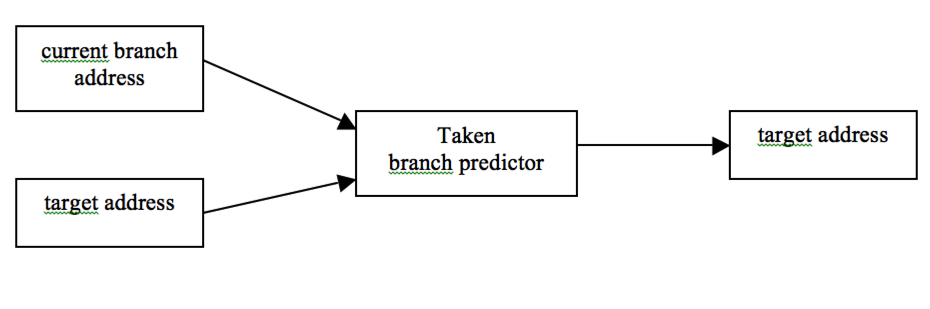


figure 1 always taken branch predictor

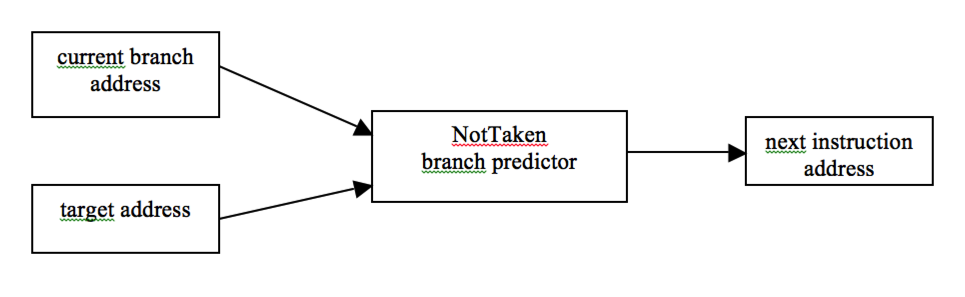


figure 2 always not taken branch predictor

A more complex form of static prediction assumes that backward branches will be taken, and forward-pointing branches will not be taken. A backward branch is one that has a target address that is lower than its own address. This technique can help with prediction accuracy of loops, which are usually backward-pointing branches, and are taken more often than not taken.

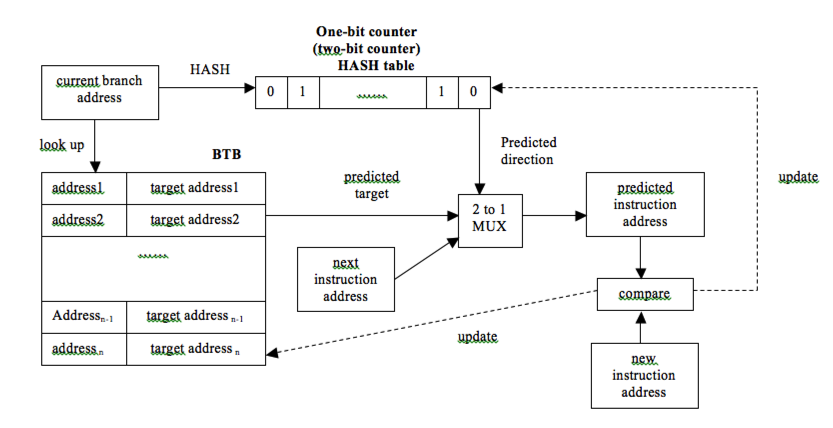


figure 3 branch prediction counter

Some processors allow branch prediction hints to be inserted into the code to tell whether the static prediction should be taken or not taken. The Intel Pentium 4 accepts branch prediction hints while this feature is abandoned in later processors.

Static prediction is used as a fall-back technique in some processors with dynamic branch prediction when there isn't any information for dynamic predictors to use. Both the Motorola MPC7450 (G4e) and the Intel Pentium 4 use this technique as a fall-back.

**3. Dynamic Branch Prediction Strategies**

**3.1 one-bit counter**

One-bit counter is the simplest dynamic branch prediction schema. The branch prediction buffer is a small memory indexed b the lower portion of the address of the branch instruction. It contains a bit that says whether the branch was recently taken or not, which is the hint for the next branch instruction. If the hint turns out to be wrong, the prediction is inverted and store back.

The schema has performing shorting. Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken.(software-based and hardware-based branch prediction strategies and performance evaluation)

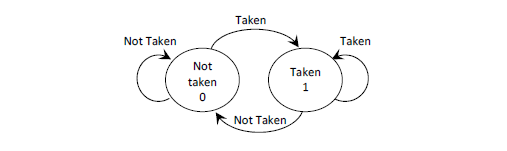


figure 1

**3.2 two-bit counter**

Two-bit schema is a specialization of the n-bit schema and is the most commonly used counter. When a branch is evaluated, the corresponding state machine is updated. Branches evaluated as not taken decrement the state toward strongly not taken, and branches evaluated as taken increment the state toward strongly taken. The advantage of the two-bit counter over a one-bit scheme is that a conditional jump has to deviate twice from what it has done most in the past before the prediction changes. For example, a loop-closing conditional jump is mispredicted once rather than twice. The schema is shown in figure 2. *Software –Based and Hardware-Based Branch Prediction Strategies and Performance Evaluation)*

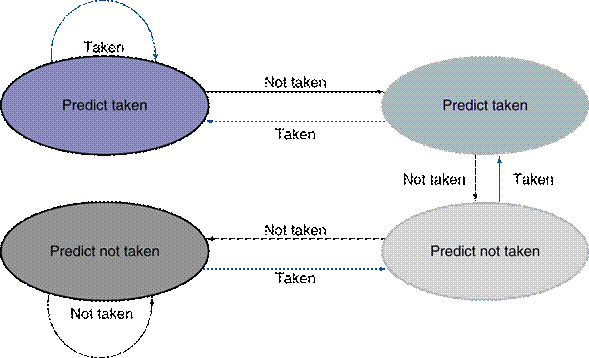


figure 2 The states in a two-bit branch prediction counter

**3.3 Two level branch prediction**

The two level branch prediction also referred to as correlation-based branch prediction, use a two-dimensional table of counters, also called “ pattern history table”. The outcome of the branch depends not only on the branch address, but also on the outcome of other recent branches( inter branch correlation) and a longer history of the same branch itself( intra branch correlation). There are three information sources to predict the branch outcome. (*Software –Based and Hardware-Based Branch Prediction Strategies and Performance Evaluation)*

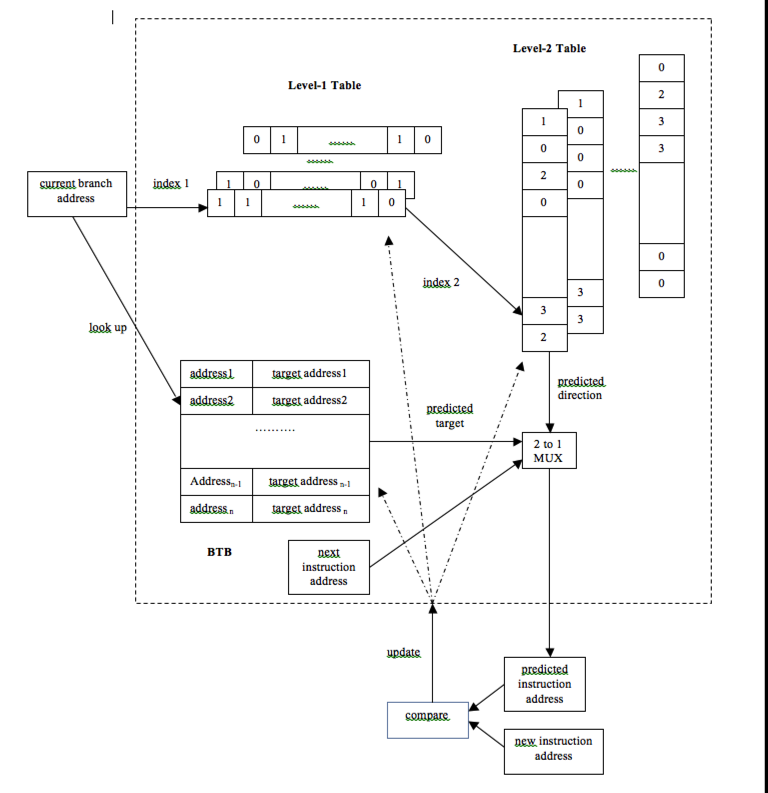


figure 3

**3.3.1 Gag**

In the GAg schema, the global history of the most recent m branches is recorded in an m-bit shift register, named global branch history register, where each bit records whether the branch was taken or not taken. We have another global branch history pattern table, which has 2m entries, each corresponding to the 2-bit counter for a global branch history. The prediction of a branch is based on the history pattern of the most recent m branches. That is, the value of the global branch history register is used to query the global branch history pattern table to find the corresponding 2-bit counter. When the counter is greater than or equal to one half of its maximum value 2, the branch is predicted as taken; otherwise, it is predicted not-taken. After the conditional branch is resolved, the outcome is shifted left into the global branch history register, and the 2-bit counter is incremented on a taken branch and decremented on a not-taken branch. (*Software –Based and Hardware-Based Branch Prediction Strategies and Performance Evaluation)*

**3.3.2 Gshare**

Gshare scheme is similar to bimodal predictor or branch history table. However, like correlation-based prediction, this method records history of branches into the shift register. Then, it uses the address of the branch instruction and branch history (shift register) to XOR them together. The result is used to index the table of prediction bits. Implementation can use either tagged or tagless tables. In the tagged table, we compare the indexed entry with a tag and, if they don’t match, we just predict as not taken. If they match, we use the corresponding prediction bits to predict the branch outcome. In the tagless implementation, the table is just directly indexed and the corresponding prediction bits are used to predict the branch outcome. We have tried both these methods, and found that tagless table performs better. We just index an entry and use its contents (prediction bits) to predict the branch outcome. ( *Software –Based and Hardware-Based Branch Prediction Strategies and Performance Evaluation)*

**3.3.3 Pag**

Instead of using one global branch history register for all the branches as in the GAg, we can have a branch history register for each branch. That is, we have a branch history register table. The address of a conditional branch is used for hashing into this table. Each entry in this table is a branch history register with m bits, recording whether the most recent m branches corresponding to this entry are taken or not. We have another global branch history pattern table which is the same with PAg. The prediction of a branch is based on the history pattern of the last k outcomes of executing the branch. Whenever a conditional branch is encountered, we find the corresponding entry in the branch history register table and get the branch history register. The branch history register is used to address the global branch history pattern table to make the prediction. After the conditional branch is resolved, the outcome is shifted left into the branch history register in the least significant bit position and is also used to update the 2-bit counter in the global branch history pattern table entry. ( *Software –Based and Hardware-Based Branch Prediction Strategies and Performance Evaluation)*

**3.3.4 Pap**

Instead of having one global branch history pattern table for all the branches, we can have a branch history pattern table for each branch. That is, for each branch, there is a branch history register and a branch history pattern table. Whenever a conditional branch is encountered, we find the corresponding entry in the branch history register table, which is used to index the branch history pattern table for that branch to find the 2-bit counter and make the prediction. After the conditional branch is resolved, the outcome is shifted left into the branch history register in the least significant bit position and is also used to update the 2-bit counter in the branch history pattern table entry. ( *Software –Based and Hardware-Based Branch Prediction Strategies and Performance Evaluation)*

**3.4 Neural Predictor**

Neural networks have been used to perform static branch prediction, where the likely direction of a branch is predicted at compile-time by supplying program features, such as control-flow and opcode information, as input to a trained neural network. This approach achieves a 20% misprediction rate compared to a 25% misprediction rate for static heuristics. Static branch prediction performs worse than existing dynamic techniques, but can be useful for performing static compiler optimizations and providing extra information to dynamic branch predictors such as the agree predictor. Learning vector quantization (LVQ), another neural method, has been suggested for dynamic branch prediction by Vintan and Iridon. LVQ prediction is about as accurate as a table-based branch predictor. Unfortunately, LVQ does not lend itself well to high-speed implementation because it performs complex computations involving floating point numbers. By contrast, our predictor has accuracy superior to any table-based method and can be implemented efficiently.（Neural Method for Dynamic Branch Prediction）

**3.4.1 Perceptron**

Perceptrons are a natural choice for branch prediction because they can be efficiently implemented in hardware. Other forms of neural networks, such as those trained by back propagation, and other forms of machine learning, such as decision trees, are less attractive because of excessive implementation costs. One benefit of perceptrons is that by examining their weights, i.e., the correlations that they learn, it is easy to understand the decisions that they make. By contrast, a criticism of many neural networks is that it is difficult or impossible to determine exactly how the neural network is making its decision. Techniques have been proposed to extract rules from neural networks, but these rules are not always accurate. Perceptrons do not suffer from this opaqueness; the perceptron’s decision-making process is easy to understand as the result of a simple mathematical formula.（Dynamic Branch Prediction with Perceptron）

**4. Implementation based on ABPS**

## 4.1 ABPS Simulator

ABPS- Advanced Branch Prediction Simulator：

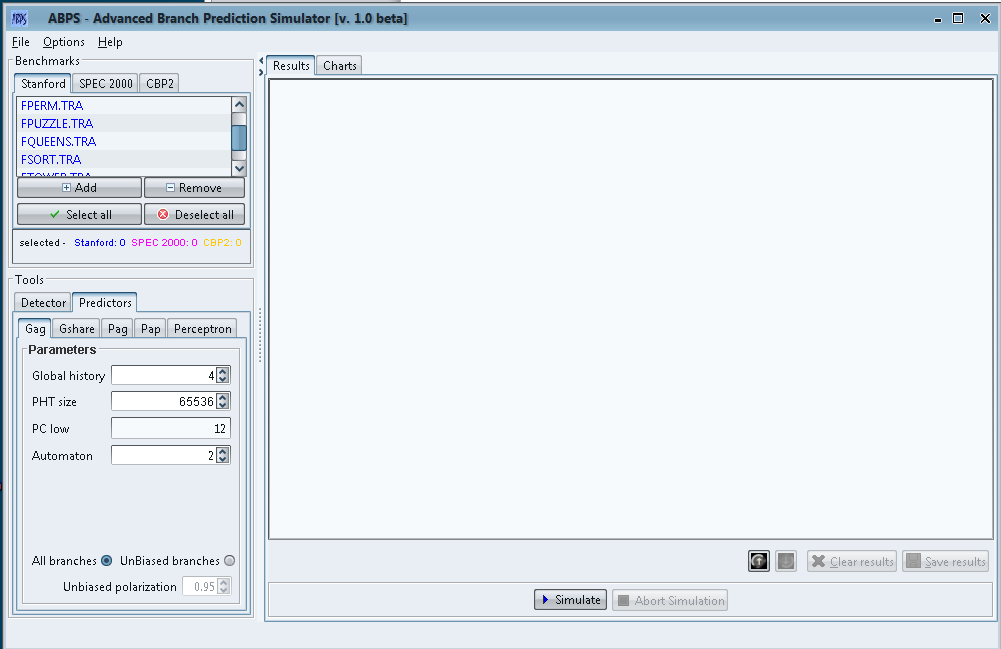
* ABPS currently uses two kind of benchmarks for simulation purpose:
* Stanford benchmarks(very helpful for didactic purpose-students);
* SPEC 2000 benchmarks(more accurate, useful for a fine grained branch prediction study)

ABPS is a Detector:

* Detecting difficult to predict branches can be easily found with ABPS, in a highly configurable manner;
* ABPS includes several detection schemes, based on:
  + Local history
  + Global history
  + HrL + HrG;
  + HrG + Path;
  + HrL + HrG + Path;

ABPS is a Predictor

* ABPS integrates two level predictors( Gag, GShare, PAg, Pap) and (State of art) neural predictors( simple perceptron, fast-path based peceptron)

****

screen shoot of ABPS main page

## 4.2 Work Based on ABPS Simulator:

The experiment is to evaluate the prediction accuracy of different branch predictor. There are five files selected from Stanford Benchmarks and tested on five different predictors. The research question thus reads as follows: ‘ Is there a prediction accuracy difference between difference predictors? ’

To answer the research question, the following research hypothesis is examined.

Hypothesis 0: the accuracy of prediction is the same for all predictors.

Hypothesis 1: The accuracy of prediction is different from all predictors.

# 5. Simulation and Performance Evaluation

## 5.1 Methology and Simulation Model

One of ABPS’s major functionalities is that it allows you to study a very important aspect of branch prediction: difficult to predict branches. ABPS can detect the branches that are difficult to predict in different contexts, using the appropriate detection schemes.

The detector has three parameters:

Detector HrL(uses only local history)

Detector HrG(uses only global history)

Path( uses path information)

The detection process (for difficult to predict branches) is very flexible, you can:

Choose the type of detector used.

Establish how much information can be used(how many bits)

Specify the threshold for a branch to be considered difficult to predict.

Next, we will being a detection process for Stanford benchmark Puzzle. We use only global history( on 4 bits) – detector HrG and that the unbiased threshold in 95%(0.95)

Next, we will perform a detection over all Stanford benchmarks using local(4 bits) and global history(4 bits) ( detector HrL + HrG)

Another major functionality of ABPS is that it allows you to simulate branch prediction using several predictors: GAg, GShare, PAg, PAp, Perceptron( simple and fast-path based)

For each predictor, you can vary its parameters.

Also, please observe that prediction can be done:

On all branches; only on unbiased branches.

Simulation over unbiased branches only, implies that the unbiased branches are first detected and afterwards predicted. Thus, prediction over unbiased branches takes more time.

We will perform next a prediction over Stanford benchmark FPUZZLE, using a simple perceptron, with default values for parameters.

The same simulation will be done again, but only over unbiased branches.

Please observe that Accuracy of prediction for unbiased branches only(Apub) is very low. This indicates how important is the study regarding difficult to predict branches.

ABPS allows you to generate simulation charts. Using charts, you can interpret simulation results much easier, you can find out the best configuration for a certain predictor, that can generate the best(prediction) performance.

Each predictor has a set of metrics that can be graphically measured.(y)

For each type of predictor you can vary each of its parameters.(x)

Our first chart will test how Ap – Accuracy of prediction varies along with the global history of a fast-path based perceptron predictor.

A chart can be represented in more ways. Choose the method that suits your simulation best.

You can set the precision of the result in option.

Generate chart for all files in Stanford directory.

To examine the hypothesis, we simulate the branch prediction process on five different predictors with the default parameter. The detail is shown in the table 1.

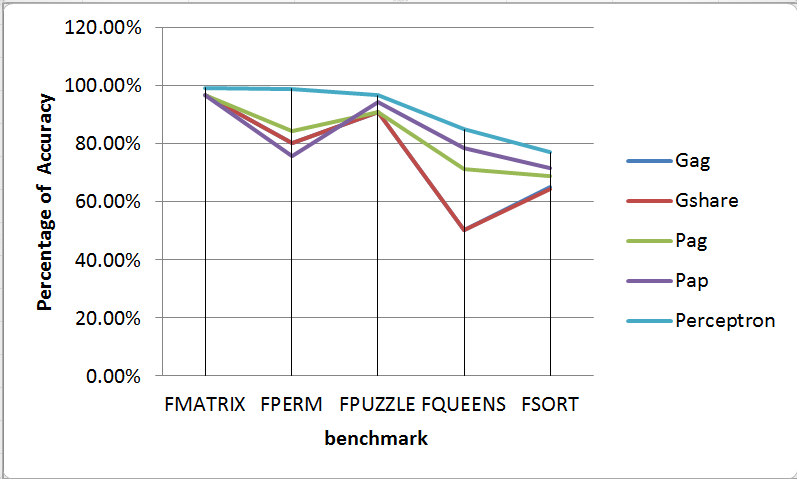
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Parameter1 | Parameter 2 | Parameter3 | Parameter4 |
| Predictor 1 |  |  |  |  |
| Predictor 2 |  |  |  |  |
| Predictor 3 |  |  |  |  |
| Predictor4 |  |  |  |  |
| Predictor5 |  |  |  |  |

## 5.2 Experiement Result

We ran the five Stanford benchmarks on the ABPS simulator. Table 1 lists the percentages of correct branch predictions from different predictors.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **FMATRIX** | **FPERM** | **FPUZZLE** | **FQUEENS** | **FSORT** |
| **Gag** | **96.705%** | **80.185%** | **90.917%** | **50.129%** | **64.971%** |
| **Gshare** | **96.705%** | **80.185%** | **90.918%** | **50.129%** | **64.288%** |
| **Pag** | **96.63%** | **84.324%** | **90.898%** | **71.252%** | **68.986%** |
| **Pap** | **96.621%** | **75.689%** | **94.294%** | **78.485%** | **71.597%** |
| **Perceptron** | **99.301%** | **98.828%** | **96.624%** | **84.93%** | **77.065%** |

**Table1: percentages of correct branch predictions from different predictors**

****

**5.3 Performance Evaluation**

The performance of the five hardware-based dynamic branch predictors is shown in figure 9. On average, all the five different hardware-based dynamic branch strategies achieve prediction accuracy better than 90%. Among all of them, the PAp-2 branch predictor and the Two-bit branch predictor are the best two. This can be explained in the following way. Since the One-bit branch predictor has an intrinsic shortcoming as described in section 2.1, the Two-bit branch predictor always works better than the One-bit branch predictor. For the PAp branch predictor, there is a two-bit counter for each global branch history pattern of each branch instruction, with our configuration each history pattern could get enough training to achieve the highest prediction accuracy. For the Two-bit branch predictor, there is a two-bit counter for each instruction. The number of the two-bit counters is large enough to ensure the high prediction accuracy.

**5.3.1 Two Level Branch Predictor**

As regards to the 2-level adaptive branch predictors, at the same space cost, PAp achieves the highest prediction accuracy with the shorter register; its prediction accuracy decreased when we made the shift register wide. We can explain this as follows. At the same space cost, with a longer shift register, since there are too many branch history patterns compared to the number of instructions executed, each branch history pattern can’t get enough training to reach high performance. Therefore, PAp with a shorter shift register achieves higher prediction accuracy. While PAp with a longer shift register achieves even less prediction accuracy than GAg does.

For the PAg branch predictor, the branch histories of all the branch instructions share the same global two-bit counter tables, which leads to confusion. Therefore, it can’t achieve as high performance as the GAg branch predictor.

**6 Neural Predictor**

**6.1 How Perceptrons Work**

The perceptron is a way to study brain function. The simplest of many types of perceptrons is a single-layer perceptron consisting of one artificial neuron connecting several input units by weighted edges to one output unit. A perceptron learns a target Boolean function t(x1,…..xn) of n inputs. In our case, the xi are the bits of a global branch history shift register, and the target function predicts whether a particular branch will be taken. Intuitively, a perceptron keeps track of positive and negative correlations between branch outcomes in the global history and the branch being predicted.

Figure 1 shows a graphical model of a perceptron. A perceptron is represented by a vector whose elements are the weights. For our purposes, the weights are signed integers. The output is the dot product of the weights vector, w0…Wn, and the input vector, x1…Xn (x0 is always set to 1, providing a “bias” input). The output y of a perceptron is computed as Y = W0 +

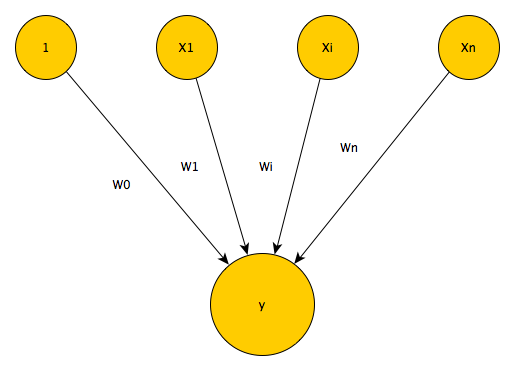


figure 1 Perceptron Model

The inputs to our perceptrons are bipolar, i.e., each xi is either -1, meaning not taken or 1, meaning taken. A negative output is interpreted as predict not taken.

A non-negative output is interpreted as predict taken.

The input values x1….xn, are propagated through the weighted connections by taking their respective products with the weights w1….. wn. These products are summed, along with the bias weight w0, to produce the output value y.

**6.2 Training Perceptrons**

Once the perceptron output y has been computed, the following algorithm is used to train the perceptron. Let t be -1 if the branch was not taken, or 1 if it was taken, and let be the threshold, a parameter to the training algorithm used to decide when enough training has been done. Since t and xi are always either -1 or 1, this algorithm increments the i th weight when the branch outcome agrees with xi , and decrements the weight when it disagrees. Intuitively, when there is mostly agreement, i.e., positive correlation, the weight becomes large. When there is mostly disagreement, i.e., negative correlation, the weight becomes negative with large magnitude. In both cases, the weight has a large influence on the prediction. When there is weak correlation, the weight remains close to 0 and contributes little to the output of the perceptron.

**6.3 Linear Separability**

A limitation of perceptrons is that they are only capable of learning linearly separable functions [8]. Imagine the set of all possible inputs to a perceptron as an n-dimensional space. The solution to the equation w0 +Xn i=1 xiwi = 0 is a hyperplane (e.g. a line, if n = 2) dividing the space into the set of inputs for which the perceptron will respond false and the set for which the perceptron will respond true.

A Boolean function over variables x1…n is linearly separable if and only if there exist values for w0…n such that all of the true instances can be separated from all of the false instances by that hyperplane. Since the output of a perceptron is decided by the above equation, only linearly separable functions can be learned perfectly by perceptrons. For instance, a perceptron can learn the logical AND of two inputs, but not the exclusive-OR, since there is no line separating true instances of the exclusive-OR function from false ones on the Boolean plane.

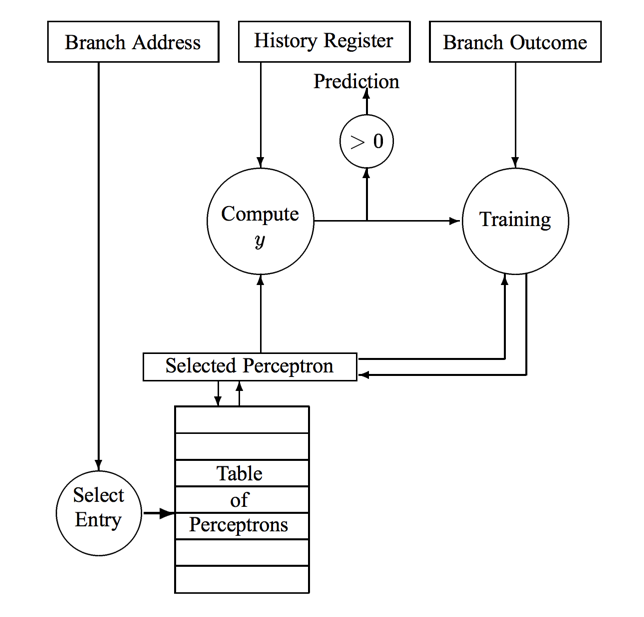
As we will show later, many of the functions describing the behavior of branches in programs are linearly separable. Also, since we allow the perceptron to learn over time, it can adapt to the non-linearity introduced by phase transitions in program behavior. A perceptron can still give good predictions when learning a linearly inseparable function, but it will not achieve 100% accuracy. By contrast, two-level PHT schemes like gshare can learn any Boolean function if given enough training time.

**3.5 Putting it All Together**

We can use a perceptron to learn correlations between particular branch outcomes in the global history and the behavior of the current branch. These correlations are represented by the weights. The larger the weight, the stronger the correlation, and the more that particular branch in the global history contributes to the prediction of the current branch. The input to the bias weight is always 1, so instead of learning a correlation with a previous branch outcome, the bias weight, w0, learns the bias of the branch, independent of the history. Figure 2 shows a block diagram for the perceptron predictor. The processor keeps a table of N perceptrons in fast SRAM, similar to the table of two-bit counters in other branch prediction schemes. The number of perceptrons, N, is dictated by the hardware budget and number of weights, which itself is determined by the amount of branch history we keep. Special circuitry computes the value of y and performs the training. We discuss this circuitry in Section 6. When the processor encounters a branch in the fetch stage, the following steps are conceptually taken:

1. The branch address is hashed to produce an index i 2 0::N 1 into the table of perceptrons.
2. The i th perceptron is fetched from the table into a vector register, P0::n, of weights.
3. The value of y is computed as the dot product of P and the global history register.
4. The branch is predicted not taken when y is negative, or taken otherwise.
5. Once the actual outcome of the branch becomes known, the training algorithm uses this outcome and the value of y to update the weights in P .
6. P is written back to the i th entry in the table.

It may appear that prediction is slow because many computations and SRAM transactions take place in steps 1 through 5. However, Section 6 shows that a number of arithmetic and microarchitectural tricks enable a prediction in a single cycle, even for long history lengths.



<Picture here>

The branch address is hashed to select a perceptron that is read from the table. Together with the global history register, the output of the perceptron is computed, giving the prediction. The perceptron is updated with the training algorithm, then written back to the table

**6. Conclusion and Future Work**

Branch prediction can highly increase the efficiency of a program, however a wrong branch prediction may lead to more delay because the wrongly fetched instructions occupy the useful functional units. So we need some specific strategies to achieve high prediction accuracy. In this paper, we evaluate some dynamic based prediction strategies using simulation tool ABPS. According to the test result, the neural predictor has a better performance than two level predictors.

Use of AI algorithm produced a higher prediction accuracy. In the future work, some research will be done regarding the neural predictor and research about how does AI can improve the branch prediction performance.

**7. Reference**

1. ABPS- advanced branch prediction simulator

<http://www.slideshare.net/adrianflorea/abps-advanced-branch-prediction-simulator>

1. Raj Parihar, *Branch Prediction Techniques and Optimizations*

[3] Daniel A. Jimenez, Calvin Lin*, Dynamic Branch Prediction with Perceptrons*

[4] Daniel A. Jimenez, Calvin Lin, *Neural Methods for Dynamic Branch Prediction*

[5] Gang Luo, Hongfei Guo, *Software –Based and Hardware-Based Branch Prediction Strategies and Performance Evaluation*