Dynamic Branch Prediction Using Machine Learning

ECS-201A Fall 2005

Yu Wang & Lei Chen
Department of Computer Science
University of California, Davis

Abstract

Microarchitectural prediction based on machine learning has received increasing attention in recent years. The common problem is that most of the Neural Network Based Branch Predictors (NNBBP) can achieve high accuracy; however, that achievement is not big enough to offset the cost of latency. As a result, the idea of using Neural Networks (NN) to perform branch prediction is not practically used compared to industrial designs. As microachitecture becomes more and more deeply pipelined, branch misprediction latency is the most important component of performance degradation. At present, there are several NNBBP have been proposed, and their algorithms and relative performance will be examined in this project. In addition to that, we have proposed our own dynamic branch predictor using Learning Vector Quantization (LVQ) to suggest another approach within the realm of machine learning.

1. Introduction

Modern computer architectures increasingly rely on speculation to boost instruction-level parallelism (ILP). One of the critical factors to improve the computer

performance is by how much the implementation can take advantage of the parallelism. As the pipelines deepen and instructions issued per cycle increases, the accurate branch prediction mechanisms become more and more important in increasing the processor performance.

There are two categories of branch prediction: static and dynamic. For static branch prediction, we have prediction not taken, prediction taken and predict based on the opcode. Some compiler-based static branch prediction uses profile, for example, IA64 uses sptk (static predict taken) and spnt (static predict not taken). There are many kinds of dynamic branch predictors. They are more widely used than the static branch predictors as well. Usually, the hardware keeps a history in cache or a separate BHT (branch history table). BHT can be indexed by low-order bits of PC on each cycle, or a more cache-like index and tag-check scheme can be used. The most commonly seen dynamic predictors include: one-bit, two-bit (2bc), two-level adaptive prediction (2la), gshare, hybrid[1], local and global , bi-mode[2], YAGS - Eden and Mudge[3].

Machine learning methods have proved to be promising in data mining, pattern recognition, prediction and classification. Using machine learning algorithm in branch prediction is one of the newest and possibly the most accurate branch prediction mechanism. The first dynamic branch prediction based on machine learning methods is proposed by Jimenez and Lin in 2000. They used one of the simplest possible neural networks, the perceptron, and achieved increased accuracy by making use of long branch histories[4].

In this paper, we implemented and tested some of the previous branch prediction mechanisms utilizing machine learning methods. In addition, we explored another machine learning method that could be used to solve the same problem. The paper contains the following parts:

1. Examine the currently proposed branch predictors based on machine learning approaches. Introduce their algorithms explicitly.
2. Implement each of the above machine learning predictors, and simulate the explicitly.
3. Propose our own branch predictor by using LVQ method.
4. Use a small Benchmark program, called Bubble Sort from SPEC Benchmark suite, to compare the performance of these predictors.
5. Based on the performance comparison, conclude the usefulness of branch predictors utilizing machine learning methods.

2. Previous Work

In 1997, B Calder et al. proposed a compiler-based static branch predictor using neural networks[5]. The branch is predicted at compile-time by using program features such as control-flow and opcode information as attributes to train the neural networks. A branch successor path that leads out of a loop or a function is less likely to be followed than a path that remains within the loop or the function. Same year Emer and Gloy used genetic algorithms to design more accurate predictors. They used genetic algorithms to "evolve" branch predictors, but the result was something similar to a highly tuned

traditional predictor.

Most recent research on dynamic branch prediction has concentrated on two-level adaptive techniques. The branch prediction using machine learning is also based on the same idea. The traditional two-level adaptive branch predictors use two levels of branch history information to make the prediction. The first level consists of a History Register that records the outcome of the last k branches. The second level is the Pattern History Table of two-bit saturating counters that record the behavior of a branch during previous occurrences of the first level predictor. The structure of traditional two-level limits the length of the history register to the logarithm of the number of counters. While using machine learning methods, it is possible to use more accurate mechanism that involves much longer history than saturating counters.

Almost all of the previous dynamic branch predictors are focusing on neural networks. Since the latency of branch prediction is a critical issue, the nature of neural networks that can reach an outcome within a small fraction of time makes the prediction easy to accomplish. Daniel A. Jimenez and Calvin Lin first proposed dynamic branch prediction using neural networks. First version was to predict using perceptrons [6]. The main idea was to replace the pattern history table with a table of perceptrons. Years later they proposed an improved predictor called fast path-based neural branch predictor [7]. In G. Steven, R. Anguera, C. Egan, F. Steven and L. Vintan's paper, they did the dynamic branch prediction using Backpropagation Network.

3. Potential Machine Learning Models

3.1 Perceptrons

The study of artificial neural networks (ANN) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons.

Perceptron is one basic type of ANN system. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs x_1 through x_n, the output $o(x_1, x_2, x_3 \ldots x_n)$ computed by the perceptron is:

$$o(x_1, x_2, x_3 \ldots x_n) = 1, \quad \text{if } w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n > 0$$

$$-1, \quad \text{otherwise}$$

Where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.

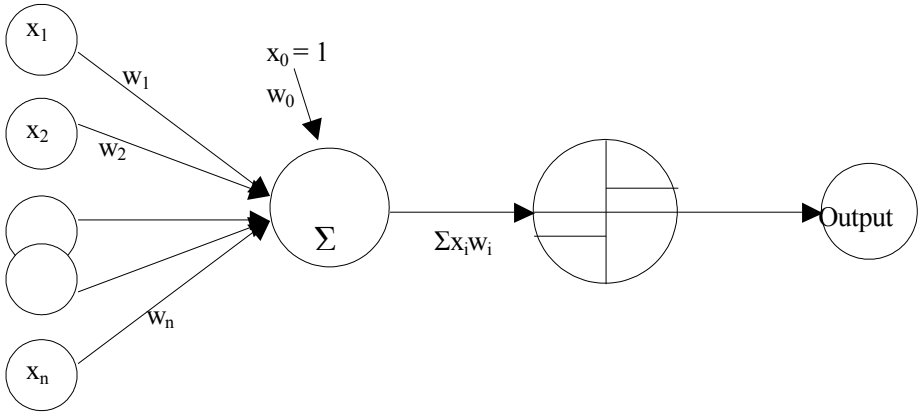


Figure-1 The Perceptron Model

The way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example. Usually, this process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input x_i according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where $\Delta w_i = \eta (t-o) x_i$

Here η is the learning rate that usually $0< \eta<1$.

As it is mentioned before, branch prediction with perceptrons is a good choice because they are simple and could be efficiently implemented in hardware.

3.2 Learning Vector Quantization

LVQ is a supervised competitive ANN. This algorithm was proposed by Kohonen in 1986 as an improvement over Labelled Vector Quantization. LVQ is associated with the neural network class of learning algorithms, though works significantly differently compared to conventional feed-forward networks like Back Propagation. LVQ intended for statistical classification. Goal of the algorithm is to approximate the distribution of a class using a reduced number of codebook vectors where the algorithm seeks to minimize classification errors. Codebook vectors become exemplars for a particular class - attempting to represent class boundaries. Each output unit represents a particular class. After training, an LVQ net classifies an input vector by assigning it to the same class as

the output unit that has its weight vector (reference vector) closest to the input vector.

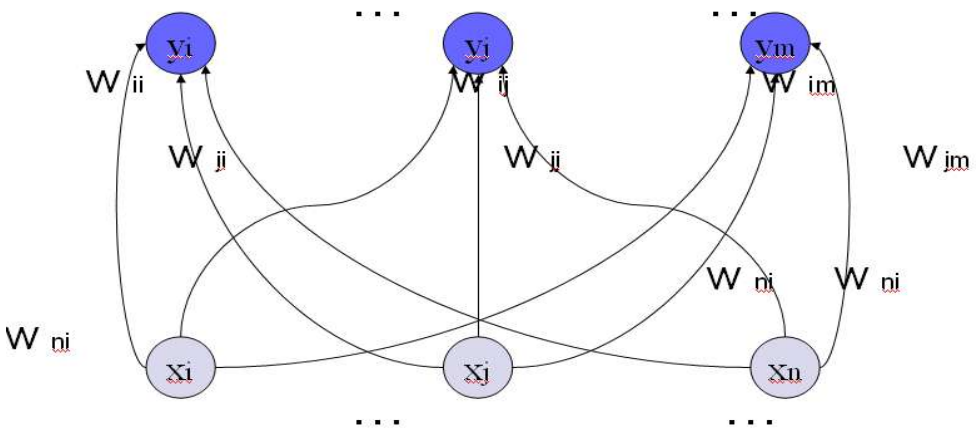


Figure-2 The LVQ model

More precisely, given input vector x and the codebook keeps the class vector (w_1, w_2 …w_n), the output class j is computed by the LVQ: $\| x\text{-}w_j \|$=min.

if the target output is equal to the prediction:

$$T = Cj,$$

then reinforce the weight vector by

$$w_j(new) = w_j(old) + a[x - w_j(old)],$$

otherwise

$$w_j(new) = w_j(old) + a[x - w_j(old)].$$

Here "a" is the learning rate that usually 0<a<1.

There are some advantages of LVQ. For example, the model is trained significantly faster than other neural network techniques like Back Propagation. It is able to summarise or reduce large datasets to a smaller number of codebook vectors suitable for classification or visualisation. It can approximate just about any classification problem as long as the attributes can be compared using a meaningful distance measure.

Normalisation of input data is not required and the generated model can be updated incrementally.

4. Implementations of Various Predictors:

***Prediction with perceptrons*:** the perceptron branch predictor proposed by Daniel Jimenez keeps a global history shift register that records the outcomes of branches as they are executed. The width of the register is determined by the history length for the predictor, let's refer to it as *h*. In addition to the global history register, the predictor also keeps an *n*(h+1)* matrix *W[0..n-1, 0..h]* of integer weights, n is a design parameter. Each row of the matrix is a *(h+1)*-length **weight vector**, which stores the weights of one perceptron when training it. In one particular weight vector, *w*, the first element *w[0]* is the learning bias. The global history shift register is represented as *G[1..h]={1..h} * {taken, not_taken}*. We simply used Boolean constants *taken* (1) and *not_taken* (0) to refer to the branch prediction. The perceptron is trained by taking the branch point at address *pc* as input to the *prediction* algorithm, which will return a Boolean value. When the training starts, one of the weight vectors is retrieved from *W* by *pc* mod *n*. According to the global history register, if the *j*th branch was a taken, then the corresponding weight at *w[j]* is added to the bias weight; otherwise, it is subtracted from the bias weight. Thus, there is a accumulative sum of the weights called *y_out.* Our prediction is simply *sign (y_out)*. If that is a positive number, the perceptron predicts it as a *taken*. When the actual branch outcome becomes available, the weight-tuning rule will adjust the weight vector in the following two steps: Step 1, if the predicted Boolean output is not equal to the actual branch outcome or the absolute value of the sum of all the weights (*y_out*) falls

below a threshold value *theta*, then the training bias of the weight vector will be incremented or decremented by 1 respectively according to the actual branch outcome. Step 2, the other weights in the weight vector will also be updated according to the global history register *G*. See figure 3 for the perceptron prediction and weight update algorithm.

function *prediction* (*pc*:**integer**) :*{taken, not_taken};*

begin

i:= *pc* mod *n*

y_out := *W[i, 0]* + $\sum_{j=1}^{h}$ *W[i, j]* if *G[j] = taken*

y_out := *W[i, 0]* - $\sum_{j=1}^{h}$ *W[i, j]* if *G[j] = not_taken*

if $y_{out} \geqslant 0$ **then**

prediction := *taken*

else

prediction := *not_taken*

end if

end

procedure *train* (*i, y_out*: **integer;** *prediction, outcome* : *{taken, not_taken}*);

if *prediction* != *outcome* **or** $|y_{out}| \geqslant 0$ **then**

W[i, 0] := *W[i, 0]* + 1 if *outcome = taken*

W[i, 0] := *W[i, 0]* - 1 if *outcome = not_taken*

for *j* **in** 1..*h* **in parallel do**

W[i, j] := *W[i, j]* + 1 if *outcome = taken*

W[i, j] := *W[i, j]* - 1 if *outcome* != *not_taken*

end for

end if

G := *(G*<<1*)* **or** *outcome*

end

Figure-3. Perceptron prediction and update algorithm

Path-based neural predictor: due to the high latency generated from adding up all the weights of a weight vector to predict one branch, Daniel Jimenez proposed another perceptron predictor, which chooses its weights vector according to the path leading up to a branch, rather than according to the branch address alone. Similar to the previous perceptron predictor, it keeps a weight matrix *W,* however, this time only the 0^{th} weight is used to help predict the current branch. Its value is added to a running total that has been kept for the last *h* branches, with each summand added during the processing of a

previous branch. The difference between the perceptron predictor and the path-based neural predictor is that the weights used to predict the branch is different. In the perceptron predictor, each of the weights in the weight vector associated with branch b_j is used to predict branch b_t. In the path-based neural predictor, the *i*th weight in the weight vector associated with branch b_j is used to predict branch b_{j+i}. The new method introduces a new vector of integers *SR[0..h]*, which contains the running total computing the perceptron output that will be used to predict the *j*th branch after the current one. The prediction algorithm is similar to the perceptron predictor. When the actual branch outcome becomes available, the weigh vector is also tuned in the following two steps: Step 1, the bias weight is updated the same as the perceptron predictor. Step 2, if the actual outcome is equal to *H[j],* which is the *j*th branch outcome in the global history register, then the *j*th weight in the weight vector is strengthened by 1; otherwise, it is weakened by 1. See figure 4 for the path-based neural prediction and weight update algorithm.

```
function prediction (pc:integer) :{taken, not_taken};
begin
        i:= pc mod n
        y:= SR[h] + W[i,0]
        if   y ⩾ 0   then
                prediction := taken
        else
                prediction := not_taken
        end if
        for j in 1..h in parallel do
                k = h -j
                if prediction = taken then
                        SR'[k + 1] := SR[k] + W[i, j]
                else
                        SR'[k + 1] := SR[k] - W[i, j]
                end if
        end for
        SR:=SR'
        SR[0] := 0
        SG := (SG << 1) or prediction
end

procedure train (i, y_out: integer; prediction, outcome : {taken, not_taken},
        v: array[1..h] of integer; H: array[1..h] of {taken, not_taken} );
begin
```

```
if prediction != outcome or |y_out| ≥ 0 then
        W[i, 0] := W[i, 0] + 1                          if outcome = taken
        W[i, 0] := W[i, 0] - 1                          if outcome = not_taken
        for j in 1..h in parallel do
                k := v[j]
                W[k, j] := W[k, j] + Σ_{j=1}^{h} 1      if outcome = H[j]
                W[k, j] := W[k, j] - Σ_{j=1}^{h} 1      if outcome !=H[j]
        end for
end if
G:= (G<< 1) or outcome
if prediction != outcome then
        SG := G
        SR := R
end if
end
```

Figure-4: Path-based neural prediction and update algorithm

Learning Vector Quantization(LVQ) predictor: as we have discussed previously, LVQ is a method to do binary classification. When it comes to utilize LVQ to predict branches, our goal is to determine whether respectable success rates could be delivered by a simple LVQ network that is dynamically trained after each branch prediction. We have two classes labels, 0 indicates *not_taken,* 1 indicates *taken.* In addition to that, we keep a global branch history *G[h]*, which stores the last *h* predictions, here we choose *h = 4*. The two initial weight vectors are chosen as *w0 = [0,0,0,0,0,0], w1 = [1,1,1,1,1,1].* The input vector for the LVQ network is constructed by concatenating the two least significant bits of the branch point *pc* with the bits stored in the *G[h]*. The prediction is performed by calculating the Euclidean Distances between the input vector and the initial weight vectors (the reference vectors). If the input vector is closer to either reference vector, it is predicted as a branch taken or not taken, respectively. Then the reference vectors are updated according to our prediction. The η is a user-specified learning rate, typical value could be 0.1. See figure 5 for the LVQ training and update algorithm.

Step0: Initialize reference vectors and learning rate η

Step1: **while** stopping condition = false **do** 2-6

Step2: **for** each training input vector X, **do** 3-4

Step3: **find** *j* so that || *X-Wj* || = *min*

Step4: Update *Wj* as follows:

if *outcome = class j,* **then**

$$Wj\ (new) = Wj\ (old) + \eta\ (X - Wj\ (old));$$

if *outcome != class j,* **then**

$$Wj\ (new) = Wj\ (old) - \eta\ (X - Wj\ (old));$$

Step5: Reduce learning rate

Step6: Test stopping condition

Figure-5 The LVQ training and update algorithm

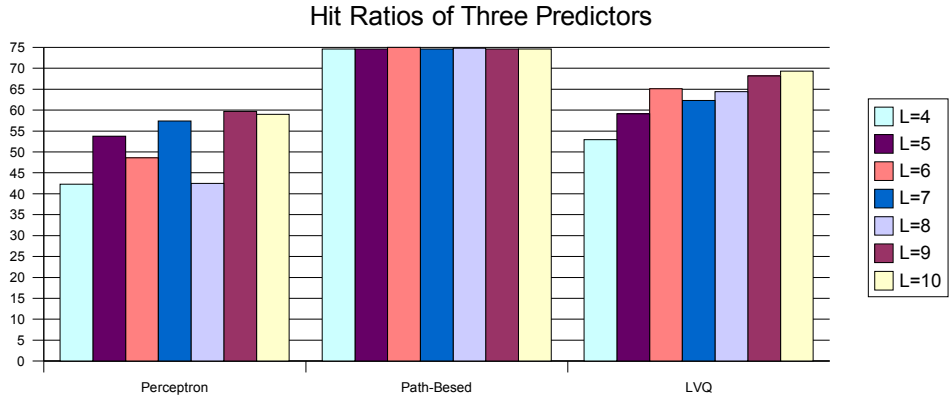
5. Predictor Simulations

We have simulated the three branch predictors using machine learning method by executing a Bubble Sort program from SPEC benchmark. The program has 576 branches to be predicted. We ran the program on each of the predictors for branch history length ranging from 4 to 10. The perceptron predictor achieved as high as 59.7% accuracy. But since it's accuracy oscillating between 42.3% to 59.7%, we consider it as an unstable approach. In fact, it might be one of the reasons why perceptron predictor is not used in today's architecture design. The path-based perceptron predictor achieved a very stable 74.6% accuracy not matter which history length we chose. But, we consider our own predictor is among the best because simulation reveals that the accuracy of LVQ predictor tends to increase as the branch history length increases, ranging from 52.9% to 69.3%. The following table shows the simulation result.

predictors	***benchmark***	***Hit Ratios With Different History Length (%)***						
		L = 4	L = 5	L = 6	L = 7	L = 8	L = 9	L = 10
Perceptron	Bubble Sort	42.3	53.8	48.6	57.4	42.5	59.7	59

predictors	*benchmark*	*Hit Ratios With Different History Length (%)*						
		L = 4	L = 5	L = 6	L = 7	L = 8	L = 9	L = 10
Path-Based	Bubble Sort	74.6	74.6	75	74.6	74.8	74.6	74.6
LVQ	Bubble Sort	52.9	59.1	65.1	62.3	64.4	68.2	69.3

Figure-6 Simulation hit ratios with different branch history length



6. Conclusion

In this study, we have examined several dynamic branch prediction mechanisms using machine learning methods. Three types of predictors are simulated: perceptron, fast path-based perceptron and LVQ predictor. The results show that the fast path-based perceptron predictor achieved the best result and the perceptron predictor was the worst. However, by estimating the hardware usage, our LVQ predictor only needs the smallest amount of hardware support. Furthermore, the potential weakness of using perceptron-

based learning methods is that it takes longer time with the increasing computational complexity. We also found that the hit ratio tends to increase as the branch history buffer increases, thus, this provides us with the extensibility in designing more accurate predictors in the future. Given a good predictor and a good algorithm design, the latency could be reduced to the level that is acceptable. The high accuracy makes it appealing to use machine learning methods to do dynamic branch prediction.

Moreover, this paper proposes another way to solve branch prediction uses the idea of LVQ. In the future, there could be more better approaches using LVQ, such as change the structure of the input vector by using pc address and global branch history, or pc address and both the global branch history and local branch history etc. There could be more research on the prediction performance in accordance with the change of branch history length, and/or the length of the pc address. In addition, more complex and sophisticated simulation can be carried out in the future.

7. Acknowledgments

We thank Dr. Matthew Farrens for encouraging us to pursue this study and for his sage advice on the first draft of this paper.

8 .References

[1] S. McFarling. Combining branch predictors. *Tech. Note TN- 36, DEC WRL, June 1993.*

[2] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. "The bi-mode branch predictor" *In Proceedings of the 30th Annual International Symposium on Microarchitecture, pages 4–*

13, Dec. 1997.

[3] A. N. Eden and T. Mudge, "The YAGS Branch Prediction Scheme", *Proceedings of the 31st Annual International Symposium on Microarchitecture*, *Dallas, Texas (November 30-December 2, 1998), pp. 69-77*

[4] Daniel A. Jimenez, Calvin Lin. "Dynamic Branch Prediction with Perceptrons," *hpca, p. 0197, Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), 2001.*

[5] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. “Evidence-based static branch prediction using machine learning.” *ACM Transactions on Programming Languages and Systems*, *19(1), 1997.*

[6]Daniel A. Jimenez. "Fast Path-Based Neural Branch Prediction," *micro, p. 243, 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36), 2003.*

[7]Ayose Falcon, Jared Stark, Alex Ramirez, Konrad Lai, Mateo Valero. "Prophet/Critic Hybrid Branch Prediction," *isca*, *p. 250, 31st Annual International Symposium on Computer Architecture (ISCA'04), 2004.*

[8]Suthikshn Kumar, Kevin Forward, M. Palaniswami. "Performance Evaluation of a RISC Neuro-Processor for Neural Networks," *hipc*, *p. 351, Third International Conference on High-Performance Computing (HiPC '96), 1996.*

[9]Ayose Falcon, Jared Stark, Alex Ramirez, Konrad Lai, Mateo Valero. "Better Branch Prediction Through Prophet/Critic Hybrids," *IEEE Micro*, *vol. 25, no. 1, pp. 80-89, January/February, 2005.*

[10]O. Hammami. "Performance Impacts of Superscalar Microarchitecture on SOM Execution," *ss*, *p. 202, The 31st Annual Simulation Symposium, 1998.*