Idealized Piecewise Linear Branch Prediction

Daniel A. Jiménez
Department of Computer Science
Rutgers University, Piscataway, NJ 08854

Abstract

Traditional branch predictors exploit correlations between pattern history and branch outcome to predict branches, but there is a stronger and more natural correlation between path history and branch outcome. I exploit this correlation with piecewise linear branch prediction*, an idealized branch predictor that develops a set of linear functions, one for each program path to the branch to be predicted, that separate predicted taken from predicted not taken branches. Taken together, all of these linear functions form a piecewise linear decision surface.*

Disregarding implementation concerns modulo a 64.25 kilobit hardware budget, I present this idealized branch predictor for the first Championship Branch Predictor competition. I describe the idea of the algorithm and as well as tricks used to squeeze it into 64.25 kilobits while maintaining good accuracy.

1 Introduction

This note describes my entry into the 1st JILP Championship Branch Prediction Competition. It is based on *piecewise linear branch prediction*, a generalization of both perceptron and path-based neural branch predictors [2, 1]. I paid no attention whatsoever to issues of implementation such as delay or numbers of gates in random logic; my only concerns were accuracy and keeping to the 64.25 kilobit limit on state. The algorithm uses only branch address and outcome information.

The original perceptron predictor learns the equation of a hyperplane in n dimensional space where n is the history length for the predictor. Dynamic branches whose pattern histories lie above the hyperplane are predicted not taken; pattern histories below the hyperplane are predicted taken. This scheme is highly accurate in practice, but cannot capture the nuanced behavior of certain branches.

Piecewise linear branch prediction learns the equation of several hyperplanes based on the path leading up to the branch to be predicted. The intersection of these hyperplanes forms the decision surface for prediction. Figure 1 shows a piecewise linear decision surface for predicting a branch whose outcome is equal to the exclusive-OR of the outcomes of the last two branches. This branch cannot be predicted with more than 50% accuracy with perceptrons; however, piecewise linear branch prediction classifies it perfectly.

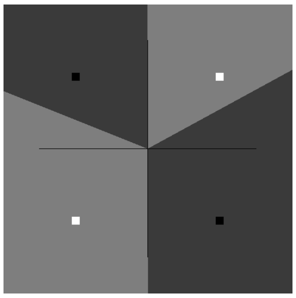


Figure 1: A piecewise linear decision surface for XOR

Section 2 describes the idea of the algorithm. Section 4 gives a list of tricks used to make the algorithm more accurate. Section 5 computes the size of the predictor to show that it stays within the limits imposed by the contest.

2 The Idea of the Algorithm

I present an algorithm in Algol-like pseudocode that captures the idea of the algorithm without going into too much detail.

2.1 Variables

The following variables are used by the algorithm:

W A three-dimensional array of integers. Addition and subtraction on elements of W saturate at +127 and -128. The dimensions of the array are arbitrarily large, i.e., large enough to accommodate any access that might be made during the algorithm.

GHL The global history length. This is a small integer.

GHR The global history register. This vector of bits accumulates the outcomes of branches as they are executed. Branch outcomes are shifted into the first position of the vector.

```
function predict (address: integer): boolean
begin
    (* output is initialized to bias weight *)
    output := W[address, 0, 0]
    (* sum weights (or their negations) chosen using
    the addresses of the last GHL branches *)
    for i in 1..GHL do
        if GHR[i] = true then
            (* if the i^th branch in *)
            output := output + W[address, GA[i], i]
        else
            (* otherwise subtract it *)
            output := output − W[address, GA[i], i]
        end if
    end for
    (* predict the branch taken if the output is at least 0 *)
    predict := output ≥ 0
end
```

Figure 2: Prediction algorithm

```
procedure train (address: integer; taken: boolean)
begin
    if |output| < θ or output ≥ 0 ≠ taken then
        if taken = true then
            W[address, 0, 0] := W[address, 0, 0] + 1
        else
            W[address, 0, 0] := W[address, 0, 0] − 1
        end if
        for i in 1..GHL
            if GHR[i] = taken then
                W[address, GA[i], i] := W[address, GA[i], i] + 1
            else
                W[address, GA[i], i] := W[address, GA[i], i] − 1
            end if
        end for
    end if
    GA[2..GHL] := GA[1..GHL − 1]
    GA[1] := address
    GHR[2..GHL] := GHR[1..GHL − 1]
    GHR[1] := taken
end
```

Figure 3: Training algorithm

GA An array of addresses. As branches are executed, their addresses are shifted into the first position of this array. In the implementation, the elements of the array are simply the lower 8 bits of the branch address.

output An integer. This integer is the dot product of a weights vector chosen dynamically and the global history register.

2.2 Prediction and Update Algorithms

Figure 2 shows the function *predict* that computes the Boolean prediction function. The function accepts the address of the branch to be predicted as its only parameter. The branch is predicted taken if *predict* returns `true`, not taken otherwise. Figure 3 shows the procedure *train* that is used when the branch is executed and it is time to update the predictor. It accepts two parameters: the address of the branch and a Boolean value that is true if and only if the branch was taken. It assumes that all variables retain the values they had at the end of the invocation of *predict* for this branch.

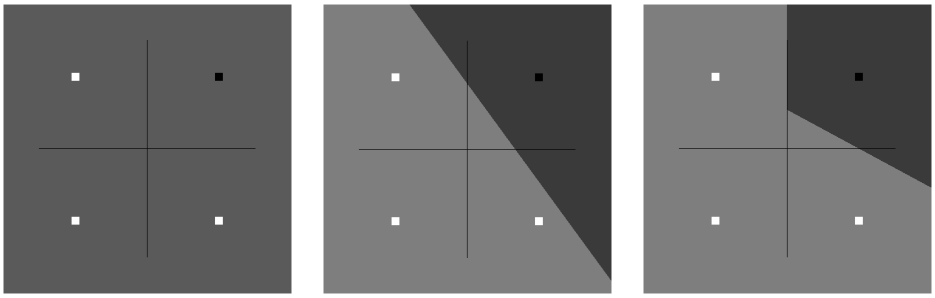
3 Examples

Perceptrons learn the equation of a hyperplane that forms a decision surface in the feature space. For branch prediction, the feature space is the outcomes of previous branches. Consider a global history length of 2. The last branch executed has outcome x which is positive if the branch was taken, negative otherwise. The second-to-last branch has an outcome of y. The perceptron predictor learns the coefficients m_1, m_2, and b for the equation of a plane $z = m_1 x + m_2 y + b$. The coefficients m_1 and m_2 are correlating weights and b is the bias weight. If x and y fall below the decision surface, i.e., $z > 0$, then the current branch is predicted taken, otherwise it is predicted not taken.

The piecewise linear branch predictor learns the equations of several hyperplanes. For any given prediction, the coefficients for that prediction are identified using the path leading to the current branch. Taken together, each of the hyperplanes used for successive predictions forms a piecewise linear decision surface in the feature space.

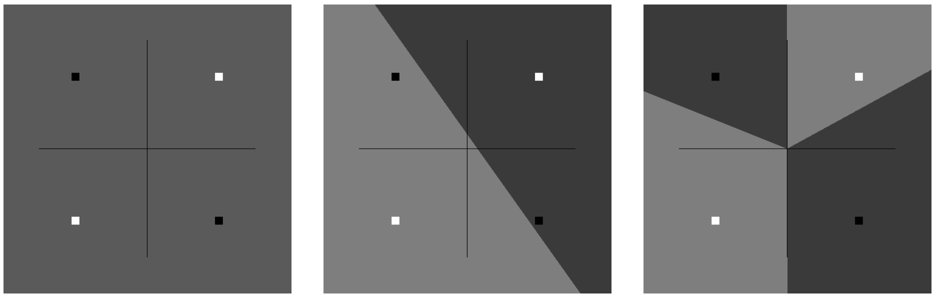
Figure 4 (a) shows the AND function represented in two-dimensional space. A white dot means false, i.e. not taken, and a black dot means true, i.e. taken. In terms of branch prediction, this figure represents a branch that is taken if any only if both of the previous branches in the global branch history were taken. Figure 4 (b) shows a 2-dimensional representation of the intersection of the $z = 0$ plane and a decision surface learned by the perceptron predictor for the AND function. The darker shaded region indicates points below the decision surface, i.e. x, y coordinates for which the predictor predicts *taken*. The lighter shaded region indicates points above the decision surface for which *not taken* would be predicted. Figure 4 (c) shows a decision surface learned by piecewise linear branch prediction. Both algorithms separate the AND function perfectly.

Figure 5 (a) shows the XOR function, i.e., a branch that is taken if and only if the previous branches in the global history had behaviors opposite from one another. Figure 5 (b) shows a decision surface learned by the perceptron predictor. This surface classifies instances correctly only 50% of the time. Clearly, a single plane cannot separate the taken and not taken instances of the XOR function; it is *linearly inseparable* [3]. Nevertheless, Figure 5 (c) shows a decision surface learned by piecewise linear branch prediction that perfectly separates taken from not taken instances. Using path information from the program that contains the branches in question, a piecewise linear decision surface is learned that classifies the XOR function correctly.



(a) (b) (c)

Figure 4: The AND function (a), a perceptron decision surface (b), and a piecewise linear decision surface (c)



(a) (b) (c)

Figure 5: The XOR function (a), a perceptron decision surface (b), and a piecewise-linear decision surface (c)

4 Tricks

In this section, I describe a number of tricks used to fit the predictor into 64.25 kilobits as well as achieve good accuracy. A number of parameters to the algorithm were chosen empirically; unfortunately, limited space does not allow me to show their values in this note, but they are described in my `predictor.h`.

4.1 Hashing

An arbitrary-sized three-dimensional array has the potential to exceed the 64.25 kilobit limit for the contest. So I use hashing to map indices of the arbitrary-sized array into locations of finite-sized table. Some triples of indices will collide with one another in the table, possibly causing destructive interference. I settled on the following hash function that seems to reduce interference over other functions I tried. Let N be the number of weights in the finite-sized table. Let H_1, H_2, and H_3 be prime numbers chosen empirically. Then the hash function is:

function *hash* (i,j,k : integer): integer
begin
  $hi := i \times H_1$
  $hj := j \times H_2$
  $hk := k \times H_3$
  $hash := (hi \text{ xor } hj \text{ xor } hk) \bmod N$
end

The values chosen for N, H_1, H_2, and H_3 are chosen empirically and appear in `predictor.h`.

4.2 Separating Bias Weights from Other Weights

I divided the weights into two pools: a pool of bias weights and a pool of general weights. Bias weights and general weights have different properties, e.g. the bias weight is usually much more correlated with branch outcome than any particular history weight, and the same bias weight is always used for a given static branch. Separating the weights into these two pools allows the sizes of these pools to be determined empirically. It also enables another optimization described below, dynamic adjusting the history length.

4.3 Using Global and Per-Branch History

To boost accuracy, I used a combination of global and per-branch history rather than just global history as outlined in the algorithms above. A table of per-branch histories is kept and indexed by branch address modulo number of histories. These histories are incorporated into the computations for the prediction and training in the same way as the global histories. This technique was used in the perceptron predictor [3] and has been referred to as *alloyed* branch prediction in the literature [5]. These parameters were chosen empirically.

4.4 Adjusting the Threshold for Taken Branches

The algorithm predicts a branch to be taken if the value of *output* is at least 0. It turns out that most of branches in the distributed traces are biased to be not taken, so changing this threshold from 0 to 3 gives slightly better accuracy.

4.5 Dynamically Adjusting the History Length

With a large number of static branches, destructive interference can be a big problem. One solution is to use a shorter global and local history length so that fewer weights are involved in any particular prediction. After 300,000 branches have passed, my predictor estimates the number of static branches by counting all of the bias weights whose magnitudes exceed 2. If this number exceeds 300, then the predictor switches to lower global and local history lengths; otherwise, it switches to higher global and local history lengths. These history lengths as well as the figures 300,000 and 300 were determined empirically. An idea of changing history length dynamically is described in [4].

4.6 Extra Weights

The bias weight and the first several global weights are repeated. That is, the algorithm uses other sources for these weights as well as the original source. Thus, a branch has more than one bias weight: one from the pool of bias weights and several from the pool of general weights. A branch also has more than one of each of the first several global weights. This improves accuracy by reducing the effect of destructive interference as well as emphasizing the relative predictive power of these weights in computing the output of the predictor. The number of extra weights is determined empirically and dynamically adjusted as described above.

4.7 Inverted Bias Weights

A bias weight is normally incremented when a branch is taken and decremented otherwise. I found that it is helpful to have extra bias weights that are decremented when a branch is taken and incremented otherwise, and subtracted from the output rather than added. The number of these inverted weights to use is determined empirically and dynamically adjusted as above.

5 The Size of the Predictor

To simplify accounting for the sizes of these variables, all variables representing predictor state are declared as fields of the class `PREDICTOR`. I only count the bits in each variable that are actually used by the algorithm, e.g. the signed variable `theta_upper` only accounts for 9 bits since its maximum magnitude never exceeds 255, even though it is represented by a 16-bit `short int`. Each of the weights is 7 bits since their maximum and minimum values are 63 and -64, respectively, even though they are represented by 8-bit `signed chars`. Figure 1 shows how I compute the size of the state used for the predictor.

Quantity of bits	Source of bits
7 * 8590	8590 7-bit general weights
+ 7 * 599	599 7-bit bias weights
+ 8 * 48	48 8-bit global addresses
+ 48	48 bits for global history register
+ 16 * 55	55 16-bit local history registers
+ 32	`output` of predictor is 32-bit int
+ 16	`i` is a 16-bit int used as loop index
+ 8	`global_history_length` is 8-bit int
+ 8	`local_history_length` is 8-bit int
+ 8	`extra_bias_length` is 8-bit int
+ 8	`extra_history_length` is 8-bit int
+ 8	`inverted_bias_length` is 8-bit int
+ 9	`theta_upper` is 9-bit signed int
+ 9	`theta_lower` is 9-bit signed int
+ 16	`lh` is a 16-bit local history
+ 32	`ntimes` is a 32-bit int
65789	**total number of bits**

Table 1: Computing the total number of bits used

The total number of bits used by my predictor is 65,789, which is less than the 64K + 256 = 65,792 bits allowed for the contest.

6 Acknowledgement

This research is supported by NSF Grant CCR-0311091.

References

[1] Daniel A. Jiménez. Fast path-based neural branch prediction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–252. IEEE Computer Society, December 2003.

[2] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th Int'l Symposium on High Performance Computer Architecture*, pages 197–206, January 2001.

[3] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.

[4] Toni Juan, Sanji Sanjeevan, and Juan J. Navarro. Dynamic history-length fitting: a third level of adaptivity for branch prediction. *SIGARCH Comput. Archit. News*, 26(3):155–166, 1998.

[5] Kevin Skadron, Margaret Martonosi, and Douglas W. Clark. A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–206, October 2000.