OH-SNAP: Optimized Hybrid Scaled Neural Analog Predictor

Daniel A. Jiménez
Department of Computer Science
The University of Texas at San Antonio

Abstract

Neural-based branch predictors have been among the most accurate in the literature. The recently proposed scaled neural analog predictor, or SNAP, builds on piecewise-linear branch prediction and relies on a mixed analog/digital implementation to mitigate latency as well as power requirements over previous neural predictors. I present an optimized version of the SNAP predictor, hybridized with two simple two-level adaptive predictors. The resulting optimized predictor, OH-SNAP, delivers high accuracy.

1 Introduction

This note describes my entry into the 3rd JILP Championship Branch Prediction Competition. My entry is based on *scaled neural analog prediction* that was presented in MICRO 2008 [1] and IEEE-Micro 2009 [2]. My optimized version of this predictor attempts to strike a balance between implementability and accuracy. An actual implementation would likely differ in design complexity but deliver similar performance. The Optimized Hybrid Scaled Neural Predictor, or OH-SNAP, uses only branch address and outcome information, eschewing the other pipeline information available in the CBP3 infrastructure.

Section 2 describes the idea of the algorithm. Section 3 gives a list of tricks used to make the algorithm more accurate. Section 4 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 pseudo-code 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 two-dimensional array of integers weights. Addition and subtraction on elements of W saturate at +63 and -64.

h The global history length. This is a small integer, 258 in my implementation.

H 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.

A 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 the lower 9 bits of the branch address.

C An array of scaling coefficients. These coefficients are multiplied by the partial sums of weights in a dot product computation to make the prediction. There is a different coefficients for each history position, exploiting the fact that different history positions make a different contribution to the overall prediction. The coefficients are chosen as $C[i] = f(i) = 1/(A + B \times i)$ for values of A and B chosen empirically. This formula reflects the observation that correlation between history and branch outcome decreases with history position, illustrated in Figure 1 taken from the original SNAP paper [1].

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

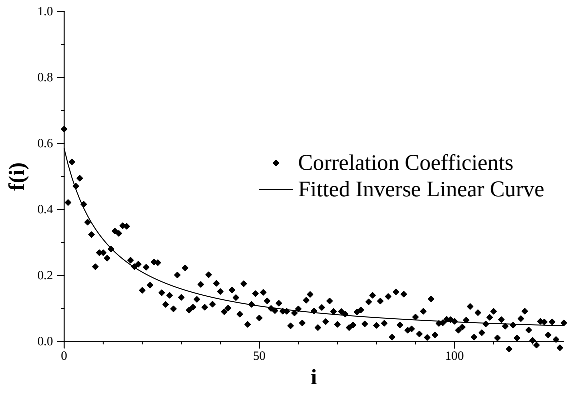

Figure 1: Weight position and branch outcome correlation. This figure is taken from the original SNAP paper [1].

2.2 Prediction Algorithm

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. The weights are organized into blocks of 8 weights each to reduce the number of tables, hence decreasing selection logic overhead. The dot product computation can be expressed as summing of currents through Kirchhoff's law. The multiplication by coefficients can be expressed by appropriately sizing transistors in the digital-to-analog converters described in the original SNAP article [1].

2.2.1 Predictor Update

The predictor update algorithm is not show for space reasons. However, it is basically the same algorithm presented in several previous related works [5, 3, 4]. The weights used to predict the branch are updated according to perceptron learning. If the prediction was incorrect, or if the sum used to make the prediction has a magnitude less than a parameter θ, then each weight is adjusted up if the outcome of the current branch is the same as the outcome of the corresponding branch in the history, or decremented otherwise.

3 Tricks

In this section, I describe a number of tricks used to fit the predictor into 65 kilobytes 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.cc`.

3.1 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. Weights for local perceptrons are kept separately from global weights. 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 [6] and has been referred to as *alloyed* branch prediction in the literature [9]. These parameters were chosen empirically.

3.2 Ragged Array

The W matrix is represented by a ragged array. That is, it is not really a matrix, but a structure in which the size of the row varies with the index of the column. Rows for correlating weights representing more recent history positions are larger since these positions have higher correlation with branch outcome and thus should be allocated more resources. The sizes of the components of the array are given in Table 1 as part of accounting for the size of the predictor.

3.3 Training Coefficients Vectors

The vector of coefficients from the original SNAP was determined statically. My predictor tunes these values dynamically. When the predictor is trained, each history position is examined. If the partial prediction given at this history position is correct, then the corresponding coefficient is increased by a certain factor (`factor` in the code); otherwise is is decreased by that factor. Also, four separate coefficients vectors are kept, indexed by branch address modulo four. Coefficients are part of the state of the predictor, so they are represented as 24-bit fixed point numbers. Now that coefficients vary, they can no longer be represented through fixed-width transistors in the digital to analog converters. However, they can still be implemented efficiently by being represented digitally similarly to the perceptron weights, then multiplied by the partial products

```
function prediction ($pc$: integer) : { taken , not_taken }
begin
    $sum := C[0] \times W[pc \bmod n, 0]$                              Initialize to bias weight
    for $i$ in 1 .. $h$ by 8 in parallel                               For all $h/8$ weight tables
        $k := (\text{hash}(A[i..i+7]) \text{ xor } pc) \bmod n$         Select a row in the table
        for $j$ in 0 .. 7 in parallel                                  For all weights in the row
            $sum := sum + C[i+j] \times W[k, i+j+1] \times H[i+j]$      Add to dot product
        end for
    end for
    if sum $>= 0$ then                                                 Predict based on sum
        prediction := taken
    else
        prediction := not_taken
    endif
end
```

Figure 2: SNP algorithm to predict branch at PC. This figure is taken from the original SNAP paper [1] and slightly modified.

through digital-to-analog conversion and multiplication with op-amps.

3.4 Training and tweaking θ

The adaptive training algorithm used for O-GEHL [8] is used to dynamically determine the value of the threshold θ, the minimum magnitude of perceptron outputs below which perceptron learning is triggered on a correct prediction. I extend this algorithm to include multiple values of θ, chosen by branch address modulo number of θs. Also, the value trained adaptively is multiplied by a small empirically tuned factor to determine whether to trigger perceptron learning.

3.5 Branch Cache

The predictor keeps a cache for conditional branches with entries consisting of partially tagged branch addresses, the bias weight for the given branch, and flags recording whether the branch has ever been taken or ever been not taken. The cache is large enough to achieve more than 99% hit rate on most of the traces. This way, there is no aliasing between bias weights. Also, branches that have only displayed one behavior during the run of the program can be predicted with that behavior and prevented from training and thus possibly aliasing the weights. The number of entries, size of partial tags, and associativity of the branch cache are empirically determined. The cache is filled with new conditional branches as they are predicted, with old branches being evicted according to a least-recently-used replacement policy.

3.6 Hybrid Predictor

Two other predictors are used alongside the SNAP predictor. A *gshare*-style predictor [7] indexed by a hash of branch address and branch history is consulted if the magnitude of the perceptron output falls below a certain tuned threshold. If the *gshare* prediction has low confidence (i.e. if the two-bit saturating counter from the gshare does not have the maximum or minimum value) then a PAg-style local-history predictor [10] consisting of a table of single bits is consulted. The history lengths of the *gshare* and PAg predictors is tuned empirically. The threshold below which the table-based predictors take over is expressed as a fraction of θ.

3.7 Other Minor Optimizations

A minimum coefficient value was tuned empirically; coefficients are prevented from going below this value when initialized. The output of local perceptrons is multiplied by a tuned coefficient before being summed with the bias weight and partial sum from correlating weights. The block size was changed from eight in the original SNAP to three in this implementation.

Source of bits	Quantity of bits	Remarks
branch queue	$129 \times (9+1+17) = 3,483$	9-bit index, 1 bit prediction, 17 bit local history
local history	$384 \times 17 = 6,528$	384 local histories, 17 bits each
local weights	$7 \times 96 \times 17 = 11,424$	96 local perceptrons, each 17 7-bit weights
weights blocks 0..6	$7 \times (3 \times 7 \times 512) = 75,264$	1st 7 columns have 512 blocks of 3 weights
weights blocks 7..12	$6 \times (3 \times 7 \times 256) = 32,256$	next 6 columns have 256 blocks of 3 weights
weights blocks 13..85	$73 \times (3 \times 7 \times 128) = 196,224$	last 73 columns have 128 blocks of 3 weights
branch cache	$64 \times 140 \times (10+7+2) = 170,240$	64 sets, 140 ways 10-bit tag, 7-bit bias, 2 bit T/NT
other predictor	$2,048 \times (2+1) = 6,144$	2K 2-bit gshare + 2k 1-bit local counters
coefficients vectors	$4 \times 24 \times (258+1) = 24,864$	4 24-bit 259-entry fixed-point vectors
pattern history	$258 + 129 = 387$	enough circular buffer for all in-flight branches
path history	$9 \times (258+129) = 3,483$	enough circular buffer for all in-flight branches
θ values	$12 \times 26 = 312$	26 12-bit threshold values
total	$530,609$	total number of bytes is 66,326 = 64.77KB
surplus bits	$65 \times 1024 \times 8 - 530,609 = 1,871$	enough for miscellaneous variables

Table 1: Computing the total number of bits used. Total number of bytes is 66,326.

4 The Size of the Predictor

Figure 1 shows how I compute the size of the state used for the predictor. The total number of bits used by my predictor is 530,609, which is less than the 65KB = 532,480 bits allowed for the contest.

5 Acknowledgement

This research is supported by NSF grants CRI-0751138 and CCF-0931874.

References

[1] Renée St. Amant, Daniel A. Jiménez, and Doug Burger. Low-power, high-performance analog neural branch prediction. In *Proceedings of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*. IEEE Computer Society, November 2008.

[2] Renée St. Amant, Daniel A. Jiménez, and Doug Burger. Mixed-signal approximate computation: A neural predictor case study. *IEEE Micro – Top Picks from Computer Architecture Conferences*, 29(1):104–115, 2009.

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

[4] Daniel A. Jiménez. Piecewise linear branch prediction. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, June 2005.

[5] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pages 197–206, January 2001.

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

[7] Scott McFarling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[8] André Seznec. Analysis of the o-geometric history length branch predictor. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*, pages 394–405, June 2005.

[9] 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.

[10] T.-Y. Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, November 1991.