SNIP: Scaled Neural Indirect Predictor

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

Abstract

This paper proposes an indirect branch predictor based on neural learning. Neural-based conditional branch predictors have been among the most accurate in the literature, so it makes sense to adapt them to the indirect branch prediction problem. However, it is not clear how to use a predictor optimized to produce a true/false output for a problem requiring the prediction of a branch target. My technique, Scaled Neural Indirect Predictor (SNIP) adapts the recently proposed SNAP predictor to predict several bits of the target address, then chooses a known target of the indirect branch with the most matching bits.

1 Introduction

This note describes my indirect branch predictor 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]. The Scaled Neural Indirect Predictor uses the idea of the SNAP predictor to predict several bits in the target address, then uses these bits to make a target prediction by choosing the most closely matching target from a list of known targets for the indirect branch being predicted. This indirect branch predictor 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

The SNAP predictor is a conditional branch predictor. For the SNIP predictor I modify the SNAP predictor to predict several target bits. I present the algorithm for predicting a single target bit 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:

h The global history length. This is a small integer, 42 in my implementation.

W An $h + 1$-column matrix of integers weights. Addition and subtraction on elements of W saturate at +15 and -16. The first column of this array, i.e. column 0, are *bias weights*, i.e., they track the bias of this bit of the branch target to be 0 or 1 regardless of branch history. The remaining columns $1..h$ are *correlating weights*, i.e. they track the tendency of this target bit to be correlated to the outcome of the corresponding branch in the history.

H The global history register. This vector accumulates the outcomes, taken or not taken, of branches as they are executed. For convenience of notation, in the algorithm these outcomes are recorded as bipolar values, i.e., -1 for not taken and 1 for taken. However, in the implementation the representation is binary. Branch outcomes are shifted into the first position of the vector. This array represents the *pattern history* of the branches leading to this indirect branch.

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 11 bits of the branch address. This array represents the *path history* of the branches leading to this indirect branch.

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 hypothesis that the correlation between conditional branch history and branch target decreases with history position.

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

2.2 Prediction Algorithm

Figure 1 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 function is invoked repeatedly for several target bits, predicting a single bit per invocation. 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 Predicting a Target

A subset of the bits of the target address is predicted by repeated invocations of the prediction algorithm using different hash functions to select weights columns. A tagless set-associative memory similar to a branch target buffer (BTB) keeps previously visited target addresses. A set of this memory is selected by taking the branch target modulo the number of sets, and the set is searched for a target with as few differences as possible in the predicted bits and target bits. That is, the target with minimal Hamming distance between the predicted and target bits is chosen as the prediction. The BTB is filled with new targets with the least-recently-used target being replaced.

Parameter	**Value(s)**
# of BTB sets	98
# of BTB ways	26
# of bias weights	19,008
# of weights vectors	1,664
History length	42
Initial θ	240
Predicted target bits	1,3,4,5,6,7,8,9,10, 11,12,14,16,17,&19
Bits per weight	5
Coefficient factor	1.0000045500
A	0.059
B	0.006
min. init. coefficient	4.3

Table 1: Empirically tuned parameters for the predictor.

2.2.2 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 of a particular bit 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 mispredicted target bit of the current branch has the same value 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. Figure 1 gives their values. Some of the parameters would be different in a real implementation. For example, the number

```
function prediction (pc: integer) : { 1, 0 }
begin
    sum := C[0] × W[pc mod n, 0]                    Initialize to bias weight
    for i in 1 .. h in parallel                     For all h weight columns
        sum := sum + C[i + j] × W[k, i + 1] × H[i]  Add to dot product
    end for
    if sum >= 0 then                                Predict based on sum
        prediction := 1
    else
        prediction := 0
    endif
end
```

Figure 1: SNIP algorithm to predict one bit of target for branch at PC. This figure is taken from the original SNAP paper [1] and modified.

of BTB sets and weights vectors would be powers of two to simplify selection logic. However, with the constraint of 65KB of state, I choose non-power-of-two table sizes to fit within the budget.

3.1 Separating Bias Weights

I divided the weights into bias weights correlating weights. Bias weights and correlating weights have different properties, e.g. the bias weight is usually much more correlated with target bit 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.

3.2 Skipping Bits

Rather than try to predict all 32 bits of the target address, the predictor only predicts certain target bits chosen through empirical tuning. Thus, the predictor skips the other bits. The full 32 bits of the target are predicted as the known target most closely matching in the predicted bits. By skipping less salient bits, the predictor avoids costly aliasing.

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. Coefficients are part of the state of the predictor, so they are represented as 64-bit floating 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 through digital-to-analog conversion and multiplication with op-amps.

3.4 Adaptively Training θ

The adaptive training algorithm used for O-GEHL [6] 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. Adaptive training seeks to strike a balance between the number of times the weights are adjusted due to an incorrect prediction versus a correct but low-confidence prediction.

3.5 Other Minor Optimizations

A minimum coefficient value was tuned empirically; coefficients are prevented from going below this value when initialized.

Source of bits	Quantity of bits	Remarks
bias weights	$19,008 \times 5 = 95,040$	19,008 5-bit bias weights
other weights	$1,664 \times 5 \times 42 = 349,440$	1,664 vectors, 5 bits/weight, 42 columns
tagless BTB	$98 \times 26 \times 32 = 81,536$	98 rows, 28 ways, 32 bits/target
pattern history	$42 + 129 = 171$	enough for all in-flight branches
path history	$(42 + 129) \times 11 = 1881$	enough for all in-flight branches
θ	12	one θ
branch queue	$(42 + 129) \times 9$	9-bit indices into history buffers
coefficients vector	$(42 + 1) \times 64$	1 bias and 42 correlating 64-bit coefficients
total	$532,371$	66,546 bytes = 64.987KB
surplus	$532,480 - 532,371 = 109$	enough for miscellaneous variables

Table 2: Computing the total number of bits used.

I initially tried predicting the first several bits of the target address. I later experimented with a number of random combinations of bits to predict. The result, represented in the code as the mask `0xb5ffa`, shows that predicting a non-consecutive subset of the target bits yields superior performance.

4 The Size of the Predictor

Figure 2 shows how I compute the size of the state used for the predictor. The total number of bits used by my predictor is 532,371, 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] 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.