Appears in Proceedings of the 2011 IEEE International Conference on Computer Design (ICCD), Amherst, Massachusetts, October 2011.

An Optimized Scaled Neural Branch Predictor

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

***Abstract*—Conditional branch prediction remains one of the most important enabling technologies for high-performance microprocessors. A small improvement in accuracy can result in a large improvement in performance as well as a significant reduction in energy wasted on wrong-path instructions. 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. We present an optimized version of the SNAP predictor, hybridized with two simple two-level adaptive predictors. The resulting optimized predictor, OH-SNAP, delivers very high accuracy compared with other state-of-the-art predictors.**

I. Introduction

High-performance microprocessors rely on accurate branch predictors to continuously supply the core with right-path instructions. A small improvement in conditional branch predictor accuracy can result in a significant improvement in performance as well as a reduction in energy consumption as fewer wasted wrong-path instructions are executed. For instance, a recent study finds that for the Intel Xeon E5440, a perfect branch predictor would yield a performance improvement of 26.0%, while halving the average number of mispredictions per 1000 instructions (MPKI) from 6.50 to 3.25 would improve performance by 13.0% [13]. For future generation processors with deeper pipelines and/or larger instruction windows, the effect on performance would be even more pronounced.

Figure 1 illustrates the steady improvement in branch predictor accuracy over the years on a set of SPEC CPU 2000 and 2006 integer benchmarks. The bimodal [20], two-level adaptive training GAs [22], *gshare* [12], hybrid [4], BiMode [11], perceptron [8], *2Bc-gskew* [18], piecewise linear neural [7], L-TAGE [16], and SNP/SNAP [1] predictors are illustrated, showing the continued progress of branch prediction research over the past two decades.

This paper introduces a set of optimizations to the Scaled Neural Analog Predictor [1], [2], a branch predictor based on neural learning and assuming a mixed analog/digital implementation to provide low latency and low power. Previous work describes in detail the implementation of this predictor; this paper shows that aggressive optimization of SNAP results in significantly improved accuracy.

The SNP/SNAP predictor is based on neural branch prediction, a technique introduced by Vintan [21] and refined by Jiménez *et al.* [8], [6], [7]. The basic idea is to associate each branch (or branch path, dynamically) with a *perceptron* [3], [14], i.e., a vector of small integer weights learned by on-line training. The perceptron is a simplified model of a neuron, hence the term "neural prediction." When the branch is predicted, the dot-product of these weights with a binary vector of recent branch outcomes is computed and the branch is predicted to be taken if the dot-product value exceeds 0, or not taken otherwise. The mechanism seems complex, but through tricks derived from high-speed computer arithmetic as well as pipelining, perceptron-based predictors can achieve latencies competitive with table-based predictors [6]. The SNAP predictor uses a mixed analog/digital implementation to achieve even lower latency and improved accuracy [1].

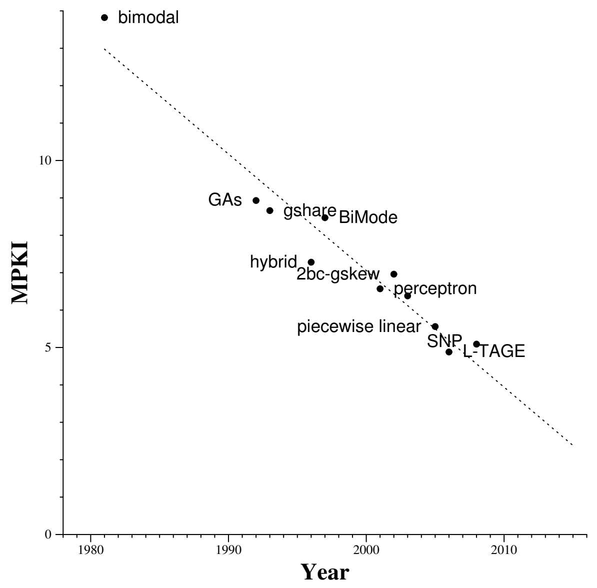

Fig. 1. Conditional branch predictor accuracy over the years. MPKI is mispredictions per 1000 instructions; lower is better.

Our optimized version of the SNAP predictor attempts to strike a balance between implementability and accuracy. An actual implementation would likely differ in design complexity but deliver similar performance; this paper explores the limits of the scaled neural predictor idea with predictor state as the only practical constraint. The Optimized Hybrid Scaled Neural Analog Predictor, or OH-SNAP, uses only branch address and outcome information. Section II describes the idea of the algorithm. Section III gives a list of optimizations used to make the algorithm more accurate. Section IV describes the experimental methodology. Section V gives results comparing the accuracy of OH-SNAP with previous work.

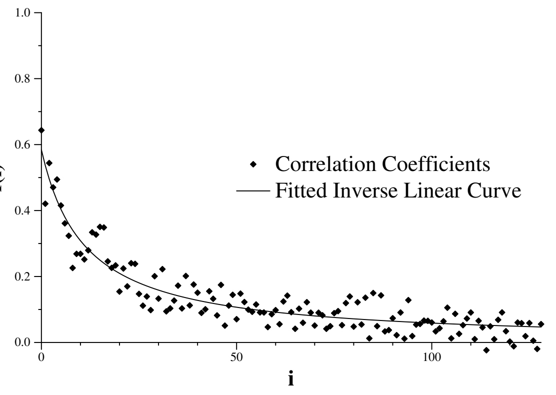

Fig. 2. Weight position and branch outcome correlation. This figure is taken from the original SNAP paper [1].

II. The Idea of the Algorithm

This section briefly reviews the Scaled Neural Predictor algorithm.

A. Variables

The following variables are used by the algorithm:

a) W: A two-dimensional array of integers weights. Addition and subtraction on elements of W saturate at +63 and -64. The first weight in each row, denoted $W[i, 0]$ for the i^{th} row, is a *bias weight*; the rest of the weights are *correlating weights*.

b) h: The global history length. This is a small integer, 258 in our implementation.

c) 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.

d) 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.

e) 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 coefficient 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 2 taken from the original SNAP paper [1].

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

B. Prediction Algorithm

Figure 3 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 paper [1]. As discussed in the original paper, the mixed analog/digital implementation allows performing the complex dot-product computation to be done with very low latency and power with negligible impact on accuracy.

1) Predictor Update: There are three kinds of updates to the predictor, each of which can proceed in parallel:

1) **Training** As branches are committed, the predictor training algorithm is invoked. 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. Correlating weights are incremented if the outcome of the current branch is the same as the outcome of the corresponding branch in the history, or decremented otherwise. The bias weight is incremented if the branch was taken, decremented otherwise. All weights saturate at 63 and -64. Since the updates to each weight occur independently of one another, this algorithm is highly parallel and is dominated by the latency of performing an increment/decrement on a 7-bit number. This algorithm is basically the same algorithm presented in several previous related works [8], [6], [7].
2) **History Update** Branch outcomes are shifted into H as they become available. Two versions of H are kept: a speculative version that is updated as soon as a prediction is made, and a non-speculative version that is updated on commit. When a mispredicted branch is discovered, the speculative version is recovered with the non-speculative version in parallel with other branch misprediction recovery actions.
3) **Adaptive Threshold Training** The threshold θ itself is adaptively trained according to an algorithm due to Seznec [15] who found that predictor accuracy seemed to be best when the number of training rounds due to mispredictions was roughly equal to the number of training rounds due to low-confidence predictions where the prediction was correct but the magnitude of the neural output (i.e. the dot-product) failed to exceed θ. This algorithm increases θ on a misprediction and decreases it on a low-confidence but correct prediction.

III. Optimizations

In this section, we describe a number of optimizations used to improve the accuracy of the predictor.

A. Using Global and Per-Branch History

To boost accuracy, we use 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

```
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
```

Fig. 3. SNP algorithm to predict branch at PC. This figure is taken from the original SNAP paper [1] and slightly modified.

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 [9] and has been referred to as *alloyed* branch prediction in the literature [19]. The number of branch histories, history length, and number of correlating weights vectors for local histories were chosen empirically.

B. 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 vary according to the same formula for deciding the coefficient values, i.e., the number of correlating weights per row is proportional to $f(i) = 1/(A + B \times i)$ for values of A and B chosen empirically. It would be possible to make a predictor with the same accuracy with rows all of the same size, but the resulting predictor would consume approximately twice the storage budget as our predictor.

C. Training Coefficients Vectors

The vector of coefficients from the original SNAP was determined statically. Our 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; 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 through digital-to-analog conversion and multiplication with op-amps.

D. Training θ

The adaptive training algorithm used for O-GEHL [15] 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. We 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.

E. 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 all benchmarks tested. This way, there is no aliasing between bias weights. Moreover, 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.

F. Hybrid Predictor

Figure 4 shows the probability that a branch is taken given the value of the neural output. In most cases, the neural output is a very good indicator of whether or not a branch will be taken. However, for neural output values near zero, the probability that a branch is taken is very close to 50%. That

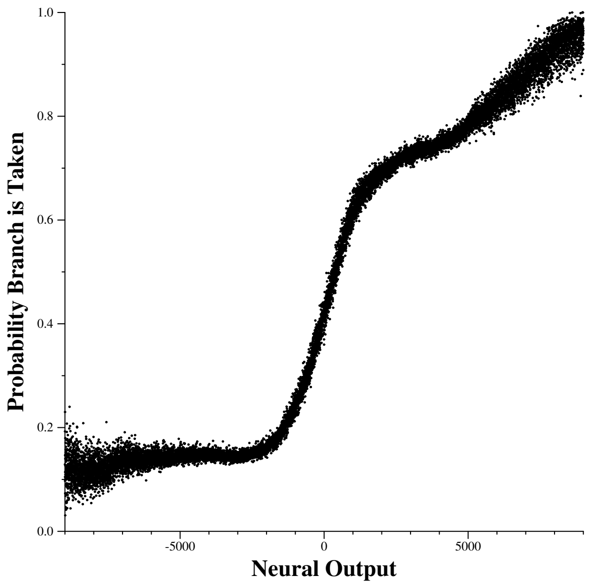

Fig. 4. Perceptron output tracks probability branch is taken

is, the neural output becomes useless as a predictor when it is near zero. Thus, when the neural output is close to zero we use an alternate method of making a prediction.

Two other predictors are used alongside the SNAP predictor. A *gshare*-style predictor [12] 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 [22] consisting of a table of single bits is consulted. The history lengths of the *gshare* and `PAg` predictors is tuned empirically. The empirically-tuned threshold below which the table-based predictors take over is expressed as a fraction of θ.

G. 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.

IV. Methodology

A. Trace-Based Simulator

We use a trace-based simulator written in C++ that apply the various branch prediction algorithms tested to a set of 40 benchmarks from the JWAC-2 competition [10]. The 40 benchmark traces are provided by Intel and are divided into five categories: client (CLIENT), integer (INT), multimedia (MM), server (SERVER), and workstation (WS). Each trace contains conditional branch address and outcome representing a run of 50 million microinstructions. We report branch predictor accuracy as number of mispredictions per 1000 instructions (MPKI). This statistic has been used in most recent branch prediction work and is preferred to the misprediction rate (i.e., number of mispredictions divided by total number of predictions) because it reflects the impact of mispredictions on performance proportionately to the normalized number of branches executed.

B. Predictors

We compare OH-SNAP with the following predictors from industry and the academic literature:

1) **L-TAGE** The L-TAGE branch predictor is currently the most accurate branch predictor in the academic literature [16], [17]. It is based on prediction by partial matching. Several tables are indexed using different history lengths based on a geometric progression; the longest matching history is used to make a prediction. We implement a version of L-TAGE that uses a 65KB storage budget based on the original author's code available from the CBP-2 website.
2) **SNP** The SNP predictor is the infeasible digital version of the analog SNAP predictor. We implement a version of SNP that uses a 65KB storage budget based on one of the original author's code available from his website.
3) **Intel Core 2** The Intel Core 2 branch predictor represents the state-of-the-art in accuracy for implemented branch predictors. The actual design of the predictor is a closely guarded trade secret, but we use a reverse-engineered model of this predictor that has been validated to achieve very similar accuracy on a set of benchmarks [13].
4) **Opteron** The AMD Opteron branch predictor is another highly accurate industrial predictor. We simulate it using a reverse-engineered model [5] that we have independently validated as having very similar on a set of benchmarks.
5) **OH-SNAP** We simulate the OH-SNAP predictor as described in this paper, starting with the design given in the original paper [1] as well as code provided by one of the authors on his website.

C. The Size of the Predictors

Figure I shows the computation of the size of the state used for the predictor. The total number of bits used by the predictor is 530,609, or approximately 65KB. We feel that 65KB is a reasonable size for a future-generation branch predictor; it is the size given for contestants in the JWAC-2 competition [10], and the Alpha EV8 predictor was designed in 2002 to consume approximately 48KB [18].

V. Results

This section gives results of experiments on all 40 benchmarks for OH-SNAP as well as other predictors. Accuracy is discussed, as well as the contribution of individual optimizations to accuracy.

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

TABLE I
COMPUTING THE TOTAL NUMBER OF BITS USED FOR OH-SNAP

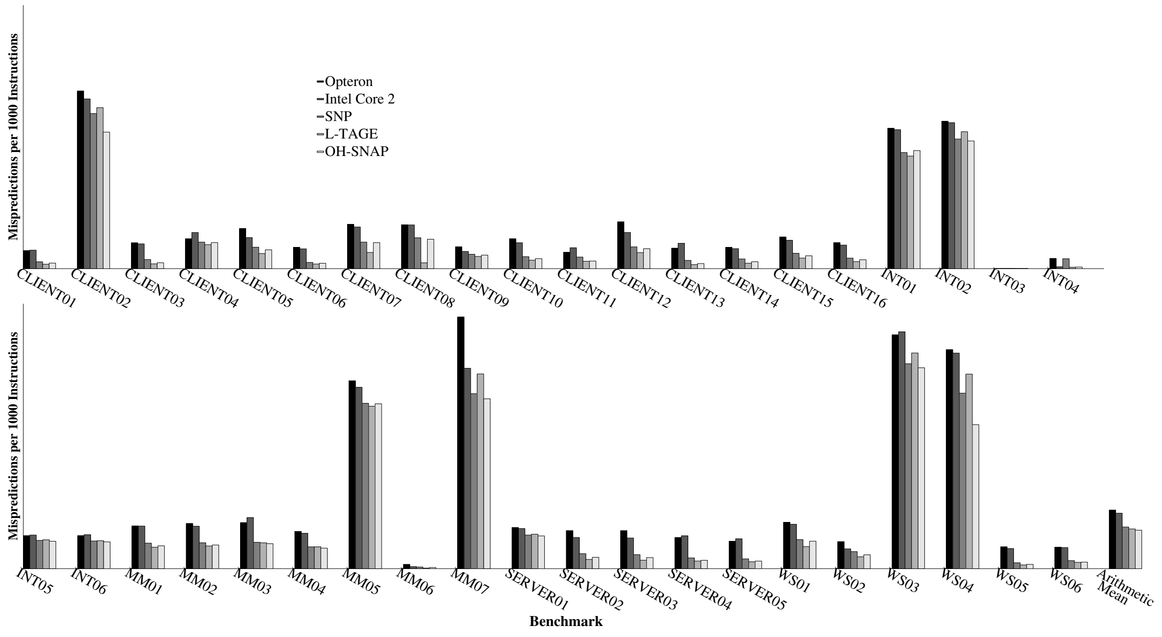

Fig. 5. Mispredictions per 1000 instructions for various predictors.

A. Accuracy

Figure 5 shows that OH-SNAP has high accuracy. On average, OH-SNAP delivers an accuracy of 3.78 MPKI. L-TAGE yields 3.90 MPKI, so OH-SNAP gives an accuracy 3.1% better than L-TAGE. SNP, the predictor on which OH-SNAP is based, gives an accuracy of 4.09 MPKI. Thus, OH-SNAP improves by 7.6% over its predecessor. The Intel Core 2 predictor yields 5.45 MPKI and the AMD Opteron predictor gives 5.76 MPKI. OH-SNAP outperforms all of the predictors except for L-TAGE on all of the benchmarks. OH-SNAP frequently outperforms L-TAGE and has fewer mispredictions on average.

Improvements in MPKI translate into improvements in performance and reductions in energy consumption. Thus, the OH-SNAP predictor is capable of improving performance and reducing energy over state-of-the-art academic predictors.

B. Contribution of Optimizations

Figure 6 illustrates the contributions of individual optimizations to accuracy. Each bar represents the presence or absence of certain optimizations. For instance, “none” means there are no optimizations over the baseline SNP predictor, “only hybrid predictor” means that the only optimization over the baseline is to use a hybrid predictor, and “no training theta” means that all optimizations are used except for the method of adaptively training the parameter θ.

The greatest contribution to accuracy is from training the coefficients. The “only training coefficients” bar shows the accuracy achieved by training the coefficients but not applying any other optimization. This optimization alone improves accuracy by 7% over the baseline SNP, more than any of the other optimizations. The addition of a hybrid predictor proves to be the least valuable optimization. If it is the only optimization, the hybrid predictor improves MPKI by 3.3%. If it is omitted but all the other optimizations are kept, MPKI is reduced by only 0.32% compared with keeping all optimizations. Interestingly, using ragged arrays to distribute predictor storage proportionately with history correlation improves performance more than using local histories.

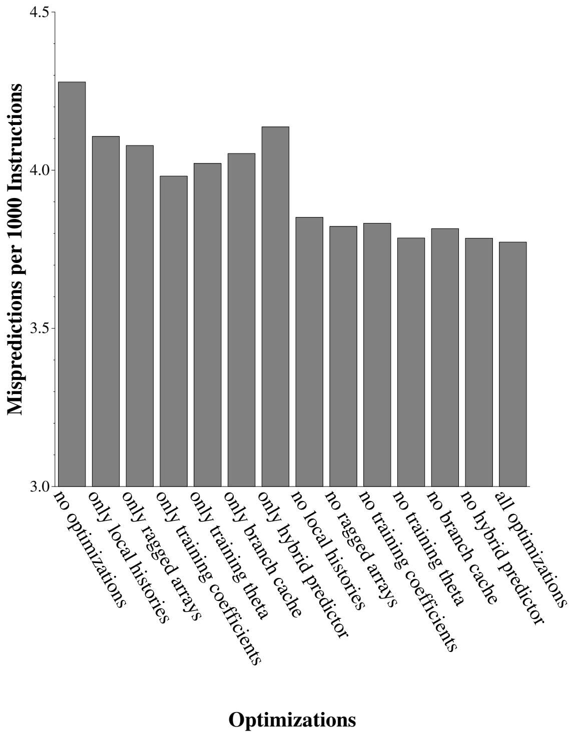

Fig. 6. Contributions of individual optimizations to overall accuracy. "No" means this optimization is the only one omitted; "only" means this optimization is the only one applied.

VI. Conclusion

The SNP/SNAP predictor provided evidence that highly accurate neural branch prediction could be done efficiently with a mixed analog/digital implementation. This paper engineers that design into a practical hybrid predictor with significantly improved accuracy.

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] H. D. Block. The perceptron: A model for brain functioning. *Reviews of Modern Physics*, 34:123–135, 1962.

[4] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[5] Agner Fog. The Microarchitecture of Intel, AMD, and VIA CPUs, 2011.

[6] 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.

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

[8] 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.

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

[10] The Journal of Instruction-Level Parallelism. *The Journal of Instruction-Level Parallelism 2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction, http://www.jilp.org/jwac-2/*, June 2011.

[11] C.-C. Lee, C.C. Chen, and T.N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 4–13, November 1997.

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

[13] Shah Mohammad Faizur Rahman, Zhe Wang, and Daniel A. Jiménez. Studying microarchitectural structures with object code reordering. In *Proceedings of the 2009 Workshop on Binary Instrumentation and Applications (WBIA)*, December 2009.

[14] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, 1962.

[15] 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.

[16] André Seznec. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 9, May 2007.

[17] André Seznec. Storage free confidence estimation for the tage branch predictor. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 443–454, feb. 2011.

[18] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakakis Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[19] 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.

[20] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, May 1981.

[21] Lucian N. Vintan and M. Iridon. Towards a high performance neural branch predictor. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 868–873, July 1999.

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