Extended Scaled Neural Predictor for Improved Branch Prediction

Zihao Zhou, Mayank Kejriwal, and Risto Miikkulainen

Abstract— **A perceptron-based scaled neural predictor (SNP) was implemented to emphasize the most recent branch histories via the following three approaches: (1) expanding the size of tables that correspond to recent branch histories, (2) scaling the branch histories to increase the weights for the most recent histories but decrease those for the old histories, and (3) expanding most recent branch histories to the whole history path. Furthermore, hash mechanisms, and saturating value for adjusting threshold were tuned to achieve the best prediction accuracy in each case. The resulting extended SNP was tested on well-known floating point and integer benchmarks. Using the SimpleScalar 3.0 simulator, while different features have different impact depending on whether the test is floating point or integer, overall such a well-tuned predictor achieves an improved prediction rate compared to prior approaches.**

I. Introduction

DYNAMIC branch prediction is a fundamental component in modern computer architecture design to achieve high performance. A branch in machine code is essentially analogous to an if-statement in high level code. When a branch is first encountered, it may not be possible to decide whether or not it should be taken: the code is executed in multiple pipelined stages and the needed information may not be available yet. Instead of stalling, however, a pipelined processor uses branch prediction to predict the target of a branch, and pre-fetches and executes instructions on the path of the predicted decision. The more accurate such branch prediction is, the more likely such speculation is to be useful. Accurate branch prediction is therefore essential to facilitate instruction-level parallelism and better performance [1].

Most research in the 1990's focused on branch predictors based on two-level adaptive scheme [2]. Two-level predictors make predictions from previous branch histories stored in a pattern history table (PHT) of two-bit saturating counters. The table is indexed by a global history shift register that stores the outcomes of previous branches. This scheme led to a series of subsequent works that focused on eliminating aliases [3]-[5]. However, all such improvements were within the framework of the existing prediction mechanism.

On the other hand, machine learning techniques offer possibility of further improvement in the prediction mechanism itself. Jimenez and Lin [6] proposed to use fast perceptrons [7], instead of PHT. Compared to other artificial neural networks that are able to fit high-dimensional non-linear data, perceptrons are easier to understand and implement, faster to train, and computationally economical. In particular, they work well with linearly separable branches, which cover a significant number of branches of practical interests. Furthermore, perceptron-based predictors can take advantage of longer histories than traditional saturating counters. Thus, improvements in perceptron based predictors have become an active area of research. For example, more complicated mechanisms like expanded branch histories, path histories, separate storage of weights, and different training methods were introduced [8]-[12].

In this paper, several different approaches proposed in the literature are combined to improve the prediction rates. In particular, the effect of using different saturating numbers to change threshold [10] and different ways to make use of history of branch addresses to form path history [11] are considered, along with the usage of expanded history [12], and the scaling of branch histories by coefficients [12]. Although the previous works proposed the models and methods of perceptron-based branch predictor in sufficient detail, they did not address the issue of how different values of the parameters or different choices of mechanism may influence the prediction rates. Furthermore, to date this issue has not been investigated empirically. This paper aims at bridging this gap by using a wide range of both integer and floating point benchmarks. The above techniques are implemented into a Scaled Neural Predictor (SNP) [12], and thoroughly evaluated and tested on a well-known open-source simulator, SimpleScalar 3.0 [13]. Conclusions are then drawn on recommended choices of each mechanism for both floating point and integer tasks, in order to optimize the performance of SNP. The paper shows that adopting the recommendations can lead to significant improvements in branch prediction rates.

SNP offers a crucial advantage over other digital neural predictors in that many of the crucial components of the SNP can be implemented using analog circuitry. Since branch prediction is primarily hardware oriented, practical branch predictors would have to be competitive in a hardware implementation. A reasonable implementation and the efficiency gains are discussed in detail in the original SNP paper [12].

The remainder of the paper is organized as follows. Section II briefly introduces the mechanism of perceptron-based branch predictors and techniques for their improvement. Section III presents the design and implementation of predictors based on SimpleScalar [13];

Manuscript received February 25th, 2013.

Zihao Zhou is a graduate student in the Department of Computer Science at the University of Texas at Austin, TX 78712, USA (e-mail: zzhou@cs.utexas.edu).

Mayank Kejriwal is a graduate student in the Department of Computer Science at the University of Texas at Austin, TX 78712, USA (e-mail: mayankkejriwal@utexas.edu).

Risto Miikkulainen is Professor in the Department of Computer Science at the University of Texas at Austin, TX 78712, USA (e-mail: risto@cs.utexas.edu).

Section IV presents comparison between difference choices of parameters as well as an analysis of the results on integer and floating point benchmarks.

II. BACKGROUND ON NEURAL BRANCH PREDICTORS

A perceptron is a vector of $h + 1$ small integer weights, where h is the history length of the predictor. A branch history is a list of 1 (taken) and -1 (not taken) of length h, in reference to the most recent h branches. The first h weights, called the correlation weights, are in one-to-one correspondence to the h branches, understood as the influence of the i^{th} branch to the next prediction, and the last weight is the bias. A table of n rows is used to store the weights of n perceptrons, with each row containing the $h + 1$ weights of one perceptron. Given the branch program counter (PC), the address is mapped to one row of the table by a hash function, for example, modulo n, such that the weights of the perceptron are dot-multiplied to the branch history. The resulting value plus the bias is the predicted value: if the value is no less than zero, the branch will be taken, and not taken otherwise. This process is illustrated by Fig. 1.

At any time when a misprediction is made, an update procedure is triggered, changing the weights of the perceptron. The bias will be decremented if the branch was taken and incremented if it was not. Each correlation weight is either increased or decreased, depending on whether the corresponding value in the branch history is 1 or -1. As an example, if the result of the branch prediction in Fig. 1 (taken) is incorrect, then the weight update once the true result is known is triggered. As shown in Fig. 2, the same branch will now be predicted as not taken, after this single update.

Recent works focus on improving the basic perceptron predictors for better accuracy. Jimenez [11] suggested using path and the history of branch addresses, and proposed a path-based predictor, where weights are accessed as a function of the PC and the path. The benefit is that the predictor can correlate not only with the pattern history, but also with the path history. Seznec [9] suggested that breaking the weights into a number of independently accessible tables rather than keeping them in a single table with h columns correlates the branch history with perceptron weights better. Renee, Jimenez and Burger [12] proposed using coefficients to scale weights, the idea of which was motivated by a practical observation that the more recent branching behavior should have more influence on the prediction in near future. They also suggested using expanded history, which contains a history of 128 bits repeatedly selected from a branch history of 40 bits. For training perceptrons, Seznec [10] proposed an adaptive threshold training algorithm in which weight update is triggered not only when the prediction is incorrect, but also when the perceptron output is less than a threshold.

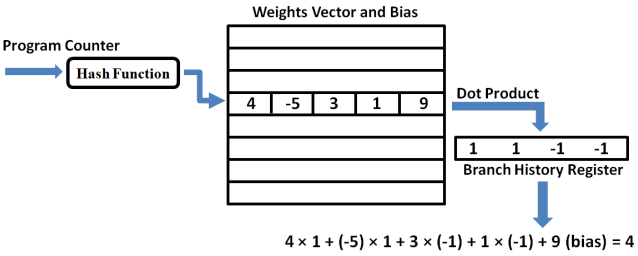

Fig. 1 Dot product of weight vector of a perceptron and branch history register. In this instance, the perceptron output is no less than 0, so the branch is predicted taken.

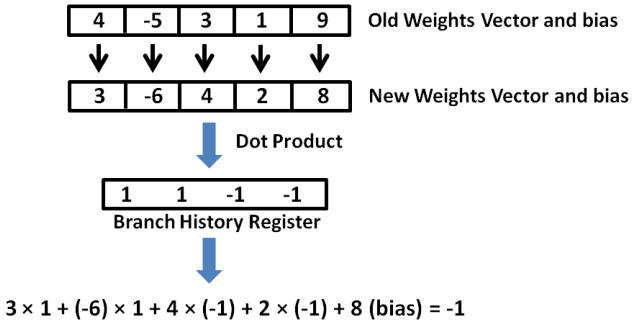

Fig. 2 Adjusting weights by a misprediction. The components of the weight vector that had positive influence are incremented while those that had negative influence are decremented. As a result, the dot product now evaluates to a negative value, so the branch is correctly predicted as not taken now.

Even though the prediction is correct, this procedure aims to achieve the balance of adjusting weights so that the computed dot product, when making a prediction, is equal to at least the threshold value. The threshold is adaptive in that it is increased after a certain number of incorrect predictions and decreased after a certain number of correct predictions. Good accuracy was achieved when training algorithm was thus invoked both after correct and incorrect predictions [10]. More details about the implementation of the adaptive threshold algorithm are provided in the next section.

III. DESIGN AND IMPLEMENTATION

In this paper, the original scaled perceptron-based branch predictor SNP [12] is implemented in SimpleScalar 3.0 [13]. The techniques discussed in Section II and incorporated in the original SNP were implemented, including the update procedure invoked on mispredictions, the adaptive threshold, coefficient scaling of weights to give greater preference to recent branches and usage of both the path and history of branch addresses. The relative effects of each of these techniques were then investigated to determine which of these yield the most significant improvements, and under what parameter settings.

SimpleScalar 3.0 is a system software infrastructure that is widely deployed commercially and in academic research for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification [13].

Each perceptron in the implementation has 128 correlation weights, stored in 16 tables, each having eight columns containing eight weights ranging from -64 to 63. The first table has 512 rows, because the most recent weights are the most important, and all the others have 256 rows. Bias weights are stored in a vector of length 2048. All correlation weights in perceptron tables are initialized to be zeroes. Seznec [10] showed that good performance was achieved with bias weights initialized to be 2.14 × (*expanded history* + 1) + 20.58. Here, *expanded history* refers to the number of the most recent branch addresses that are considered by the predictor. The bias weights in this paper's implementation were initialized in accordance with the above expression, with *expanded history* set to 128, for all experiments that were conducted.

To index each table, branch PC is hashed to 11 bits. Eight of the 11 bits are XORed with a fraction of the array A, resulting in an eight or nine bit index for one of the 256 rows (or 512 rows for the first table). The whole 11 bits are also used to index one of 2048 bias weights. The indexed correlation weight vector is dot multiplied with the expanded history, and subsequently added to the indexed bias weight; this final value then leads to the prediction.

As described in Section II, an adaptive threshold training algorithm is used. The weights are updated not just on mispredictions, but also if the calculated dot product is below a threshold that changes dynamically as the program executes. The change in threshold is controlled by a saturating counter, which records the number of consecutive correct or consecutive incorrect predictions. This counter increments every time when a correct prediction is made and decrements otherwise. Unlike correlation weight update, the threshold is only changed once the magnitude of this counter reaches a pre-set maximum value. The algorithm to trigger modification of threshold is illustrated in Fig. 3. The pre-set maximum value shown in this figure is set to 32. Experiments are conducted in which different pre-set maximum values are used to find the best frequency to change the threshold and the results are discussed in Section IV-A.

In the implementation discussed in this paper, expanded path history is used. The addresses of recent branches are hashed into 128 bits, forming the so-called path history, stored in an array A[128]. The algorithm uses an eight bit sliding window to obtain the vector A[0…7], A[8… 15],…, A[120-127] by moving the sliding window 16 times on recent branch histories. Only one bit of each branch address is selected. The moving distance of the sliding window can be one bit, two bits, four bits, or eight bits, thus allowing a choice of different numbers of branch addresses to generate the path array. The algorithm for generating eight bit long A_k for k^{th} table is illustrated in Fig. 4. The coefficient before k controls the moving distance of the sliding window. If the moving distance is one bit each time, then only the most recent branch addresses are used.

```
if(pred_taken != taken){
     thread_adapt_counter++;
     if(thread_adapt_counter >= 32){
          theta++;          //theta is the thread hold
          thread_adapt_counter = 0;
     }
}
if(pred_taken == taken && perceptron_output < theta){
     thread_adapt_counter--;
     if(thread_adapt_counter <= 32){
          theta--;
          thread_adapt_counter = 0;
     }
}
```

Fig. 3 Changing threshold by saturating number

```
//moving distance is 4 bits in this case
int i = 4*k;
// A_k is the hash path for the k^th table
int A_k = 0;
for(int j = 0; j < 8; j++){
     A_k <<= 1;
     //addresses is the past branches addresses
     //hash the 3rd lowest bit of the addresses
     A_k |= !!(4 & addresses[i++];
}
//table 0 has 512 entries. We need one more bit
if(k==0){
     i = 0;
     A_k <<= 1;
     A_k |= !!(8 & addresses[i];
}
return A_k;
```

Fig. 4 Path selecting and hashing algorithm

However, if the moving distance is eight bits, then no repeated history addresses are used and total of 128 different branch addresses are taken into account. The influence of different moving distances on prediction accuracy is discussed in Section IV-B.

Another way to investigate the influence of path is to consider different hashing schemes. A hash function aims to select the most representative bits from the branch addresses to predict a future branch, but some bits may be more representative than others. For example, the third lowest bit is much more representative than the second and first lowest bit because the lowest two bits may never change if the instructions are stored by word. Different hashing schemes are evaluated: hashing the third and fourth lowest bits, hashing the third lowest bit, or hashing the second lowest bit from branch addresses into path, and their influences. Results are shown and discussed in section IV-C.

Renee, Jimenez and Burger [12] proposed that the branch history be stored in a global branch history register H using 40 bits, and then expanded into 128 bits by repeatedly selecting bits from this register. They reported that expanded history, hashing from the most recent 40 branches to 128 bits, leads to better prediction rates. The claim was evaluated in this paper by comparing a 128 bit expanded history with a 128 bit ordinary history. The results are discussed in section IV-D.

The more recent histories have a larger correlation with current prediction. To emphasize the recent branch histories, the recent histories were scaled to a larger value while reducing the effect of old histories. To this end, each bit in

the expanded history has a coefficient to stress stronger influence of more recent branches on future predictions. In particular, Renee, Jimenez and Burger [12] obtained the relationship between the coefficient and i^{th} bit in expanded history from experiments, which is $1/(0.1111+0.037i)$. In their implementation, they placed an upper bound of one for the coefficients, i.e. the coefficient is either $1 / (0.1111 + 0.037i)$, when the value is smaller than one, or one otherwise. In this paper, experiments are conducted to investigate the effect of scaling coefficients by assigning coefficients without bound, assigning coefficients with an upper bound of one, and not assigning any coefficients. The results are discussed in Section IV-E.

It should be noted that in these experiments, the traditional training-prediction paradigm of machine learning is not used, in which the perceptron is first trained off-line by a set of training samples until weights converge, and then used for static prediction. Instead, an on-line training and prediction approach is used, wherein the weight update procedure is invoked every time the calculated dot product is below the adaptive threshold, as detailed in Section II. This is in contrast to a two-stage system, where the first stage only involves training, and the second, testing or evaluation. The branch predictor is therefore not static and can change continuously during the program flow. Also, during the training and prediction, an out-of-order simulator is employed, which means that the CPU is allowed to pipeline ahead of instructions. Hundred million instructions are used for each round of testing.

IV. EXPERIMENTAL RESULTS

In this section, comparisons of prediction rates under different configurations of the predictors are presented and the influence of those parameters on prediction accuracy is studied. All figures in this section have prediction accuracy on the *y*-axis, scaled to a range between zero and one. Prediction accuracy is defined as the ratio of the number of successful predictions to the total number of predictions made by the predictor.

A. Saturating Counter for Changing Threshold

Based on the algorithm in Fig. 3, saturating numbers for the saturation counter are chosen to be 0 (basic training method), 1, 16, 32, or 64. The third lowest bit of all 128 branch addresses is used to form path array A. Scaling coefficients are also applied to branch histories. Both integer benchmarks and floating point benchmarks are tested, as shown in Fig. 5.

The results show that using a saturating number larger than zero generally leads to better results for integer benchmarks and for most floating point benchmarks. For integer benchmarks, there is not a significant difference as long as the saturating number takes a value larger than zero.

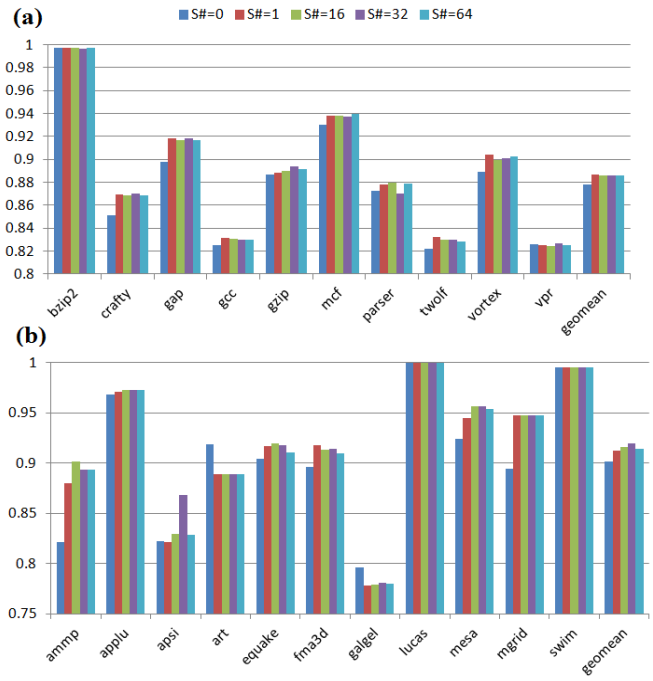

Fig. 5 Integer benchmark results (a) and floating point benchmark results (b) results for different saturating numbers. In this figure, and figures 6-9, the benchmarks are on the *x*-axis and prediction accuracy, scaled to [0,1] is on the *y*-axis. In this figure, the bars for each benchmark represent the accuracies for five different saturation values of the saturating counter. The results show that a saturation value of 32 would be best overall for maximizing branch prediction accuracy.

For floating point benchmarks, saturating number of 32 generally yields better results, according to the *geomean* benchmark, indicating that for floating point benchmarks, the threshold should be adapted at every 32 consecutive correct or consecutive incorrect predictions. For general prediction design therefore, setting the saturation value to 32 is recommended since this value would maximize prediction accuracy on both integer and floating benchmarks, according to these benchmark results.

B. Expanding Path History

As proposed in the previous section, the path array A is employed to establish better correlation between branch address history patterns and future predictions. It is important to study the effects of different hashing schemes on the branch addresses to generate path array A. In this section, different moving distances are used, e.g. one bit, two bits, four bits or eight bits, of sliding window to generate path array A (this algorithm is shown in Fig. 4). A smaller moving distance corresponds to repeatedly selecting more recent branch addresses, while a larger moving distance also takes branch addresses further in the past into account. To complete the comparison, the prediction rate is determined if the path array A is not used. The results are illustrated in Fig. 6.

This result empirically confirms the proposal of Jimenez [11] that taking address path patterns into account, along with branch history patterns, leads to better prediction rates. Using paths is always much better than not using paths.

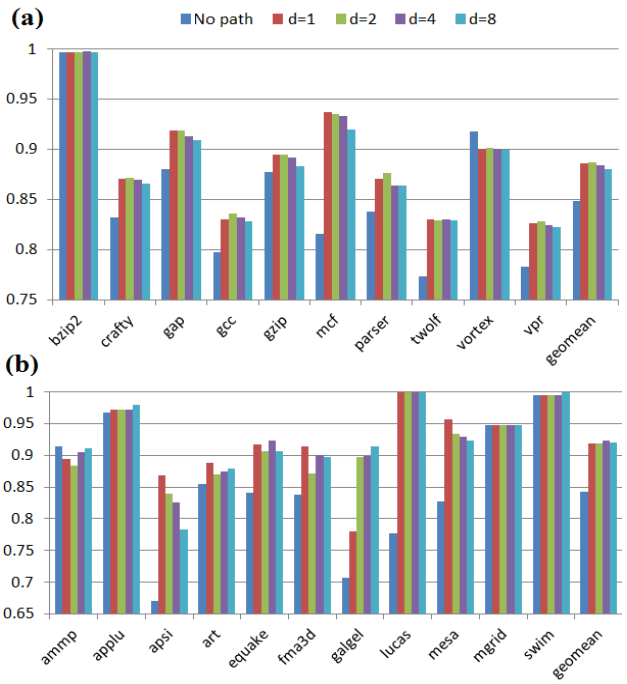

Fig. 6 Integer benchmark results (a) and floating point benchmark results (b) for generating array A at different moving distance. The bars represent different values for *d,* the moving distance. For integer benchmarks, maximum accuracy (on average) is achieved when *d* equals two bits, and for floating point benchmarks, maximum accuracy (on average) is achieved when *d* equals four bits.

However, the results also show that it is not necessarily the case that the longer the actual branch history addresses that are taken into account, the better the prediction rates will be. For integer benchmarks, a moving distance of two bits yields the best results, and for floating point, a moving distance of four bits proves to be superior to the suggestion of Renee, Jimenez and Burger [12], who use a moving distance of eight bits along with 128 different branch addresses. This result may be explained as follows: the most recent branches are just enough for future predictions, and taking into account recent branches from further into the past is not only useless, but also may increase mispredictions in the future. In conclusion, the recent branches have more positive influence and should be emphasized more when making future predictions.

C. *Different Hashing Schemes*

As mentioned earlier, when generating the path array A, only one bit of each branch address is chosen, so it is crucial to choose the most representative bit from each branch address. Different low-order bits or their combinations from branch addresses are used to form a path. In particular, the third lowest bit, combination of the fourth and third lowest bits, and second lowest bit, are used, as shown in Fig. 7.

Testing results clearly indicate that some bits or bit combinations are more representative than others: hashing instructions are stored by word. Hashing the third lowest bit is roughly similar to combining the third and fourth lowest bits for integer benchmarks, but leads to a 2% improvement for floating point benchmarks. Therefore, it is not necessarily the case that the combination provides more information than the single bit and the architectural cost of such combinations could be non-trivial. For general design therefore, hashing the third lowest bit is recommended.

D. *Expanded Branch History vs. Ordinary History*

The fourth experiment focused on using an expanded branch history of 128 bits that repeatedly hash from a history of the most recent 40 branches (i. e. the history of taken or not taken of each branch), and compared it with an ordinary history containing the results of most recent 128 branches. The results are illustrated in Fig. 8.

The results show that the expanded history achieves roughly the same accuracy as ordinary history for floating point benchmarks, and slightly worse accuracy for integer benchmarks. The expanded history, i.e. 40 bits hashing to 128 bits, forms a good approximation to a real history of 128 bits. In general design with sufficient budget, an ordinary history of 128 bits is recommended, but in the case of limited budget, expanded history can be a good approximation.

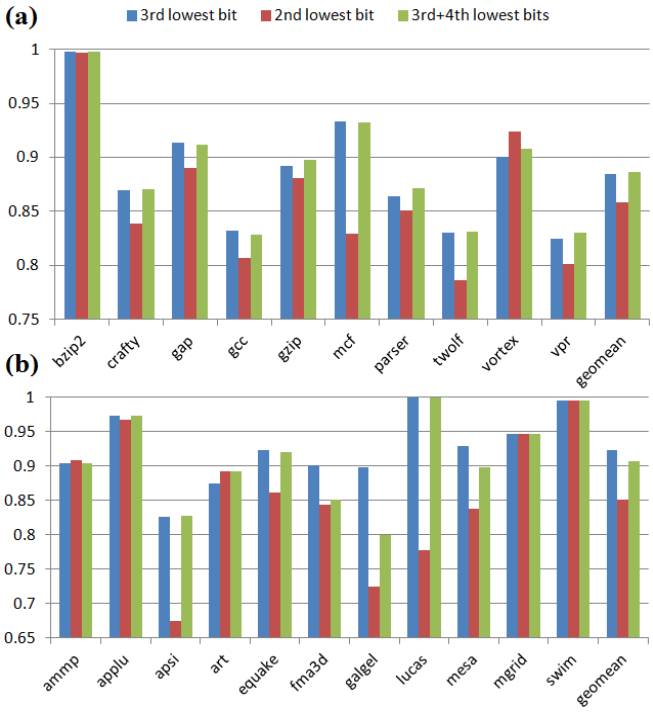

Fig. 7 Integer benchmark results (a) and floating point benchmark results (b) for different hashing schemes. The results show that hashing the second lowest bit of each address is the worst option, while hashing the third lowest bit offers nearly the same advantages as the combination of the third and fourth lowest bits.

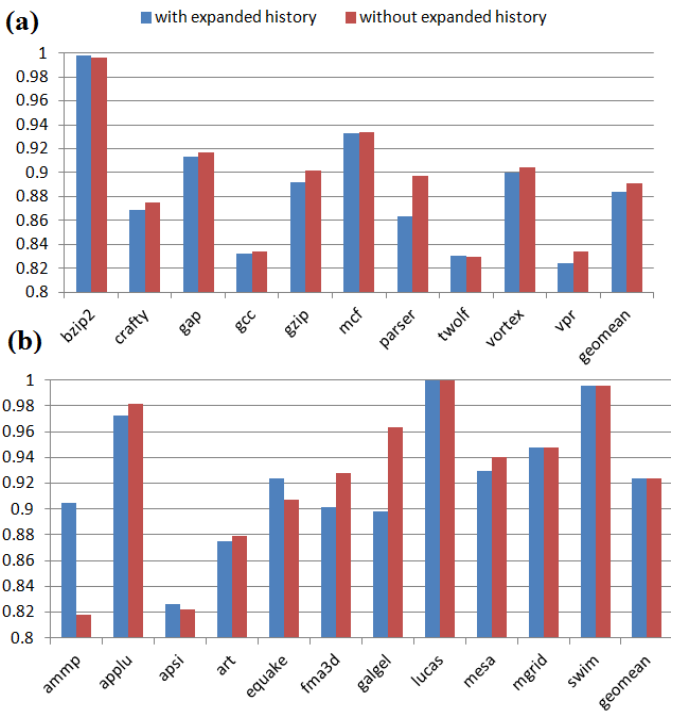

Fig. 8 Integer benchmark results (a) and floating point benchmark results (b) for expanded history and ordinary history. The results show that expanded history outperforms ordinary history for nearly all cases.

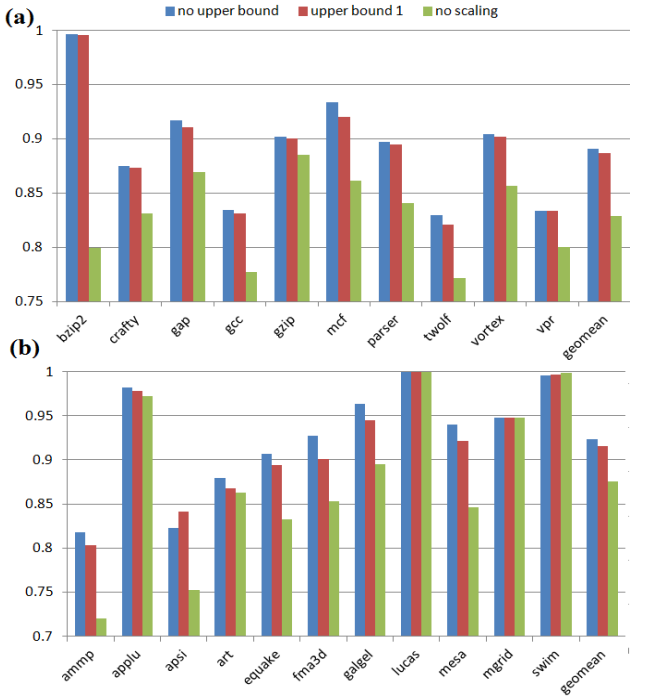

Fig. 9 Integer benchmark results (a) and floating point benchmark results (b) for different scaling coefficient schemes. The results show that scaling outperforms non-scaling considerably. Not using an upper bound for the scaling is better than using it, although such an upper bound is one of the extensions proposed to the original SNP.

E. Scaling Coefficients

Three ways to assign scaling coefficients are considered: (1) $1/(0.1111+0.037i)$, (2) $1/(0.1111+0.037i)$ with an upper bound of one, and (3) no scaling where all coefficients are assigned to one. The testing results are shown in Fig. 9. The results confirm the suggestion of Renee, Jimenez and Burger [12] that scaling places more emphasis on recent branches, and thus, generally leading to better prediction rates. Non-scaling is far worse than two other scaling methods.

However, placing an upper bound behaves slightly worse than no upper bound: this result is intuitive because an upper bound limits the expressiveness of "recent influence" of branches. Therefore, for general design, placing no upper bound on the coefficients is recommended.

V. Conclusion

The SNP had already been shown in prior work to achieve state-of-the-art performance compared to other competitive schemes. In this paper, its original design was implemented in SimpleScalar 3.0, a powerful system software infrastructure that is widely deployed for program performance analysis and microarchitectural detailing. In addition, several modifications to SNP were implemented based on proposals in prior work. Extensive tests were performed on a wide range of integer and floating point benchmarks both with and without these modifications. Based on these empirical data, recommendations were made on how the original SNP could be further improved so that branch prediction can be made more accurate.

Benchmark results show that adopting these recommendations can lead to significant improvements in performance.

A range of further experiments were then performed to verify the SNP's design choices empirically on both integer and floating point benchmarks and to suggest further refinements for better accuracy. Although many of these design choices are justified, better prediction rates can be attained by modifying some of the originally proposed parameters, and some of the modifications depend on whether they are applied to integer or floating point benchmarks. Overall, extended SNP proved to be a powerful approach, and should result in a significant practical application of neural networks in the future.

References

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition. San Francisco: Morgan Kaufmann, 1996, pp. 380-383.

[2] T. Y. Yeh and Y. Patt, "Two-level adaptive branch prediction," in *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture (MICRO'91)*, 1991.

[3] S. McFarling, "Combining branch predictors," Digital Western Research Laboratory, Tech. Rep. TN-36m, 1993.

[4] E. Sprangle, R. S. Chappell, M. Alsup and Y. Patt, "The Agree predictor: A mechanism for reducing negative branch history interference," in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*, 1997.

[5] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[6] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, 2001.

[7] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. New York: Spartan, 1962.
[8] A. Seznec, "Redundant history skewed perceptron predictors: Pushing limits on global history branch predictors," IRISA, Tech. Rep. 1554, 2003.
[9] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*, 2005.
[10] A. Seznec, "A 256 kbits L-TAGE branch predictor," in *Journal of Instruction-Level Parallelism (JILP) Special Issue: The second Championship Branch Prediction Competition (CBP-2)*, vol. 9, 2007.
[11] D.A. Jimenez, "Fast path-based neural branch prediction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, 2003.
[12] A. Renee, D.A. Jimenez, D. Burger, "Low-Power, High-Performance Analog Neural Branch Prediction," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*, 2008.
[13] T. M. Austin and D. Burger. The SimpleScalar Tool Set, Version 3.0, 1998.