A Power-Aware Alternative for the Perceptron Branch Predictor

Kaveh Aasaraai and Amirali Baniasadi

University of Victoria, Victoria BC, V8P 3Y9, Canada
{aasaraai,amirali}@ece.uvic.ca

Abstract. The perceptron predictor is a highly accurate branch predictor. Unfortunately this high accuracy comes with high complexity. The high complexity is the result of the large number of computations required to speculate each branch outcome.

In this work we aim at reducing the computational complexity for the perceptron predictor. We show that by eliminating unnecessary data from computations, we can reduce both predictor's power dissipation and delay. We show that by applying our technique, predictor's dynamic and static power dissipation can be reduced by up to 52% and 44% respectively. Meantime we improve performance by up to 16% as we make faster prediction possible.

1 Introduction

The perceptron branch predictor is highly accurate. The high accuracy is the result of exploiting long history lengths [1] and is achieved at the expense of high complexity.

Perceptron relies on exploiting behavior correlations among branch instructions. To collect and store as much information as possible, perceptron uses several counters per branch instruction. Such counters use multiple bits and record how each branch correlates to previously encountered branch instructions. The predictor uses the counters and performs many steps before making the prediction. These steps include reading the counters and the outcome history of previously encountered branches and calculating the vector product of the two.

In this study we introduce power optimizations for perceptron. We show that while the conventional scheme provides high prediction accuracy, it is not efficient from the energy point of view. This is mainly due to the observation that not all the computations performed by perceptron are necessary. In particular, computations performed on counter lower bits are often unnecessary as they do not impact the prediction outcome. We exploit this phenomenon and reduce power dissipation by excluding less important bits from the computations.

We rely on the above observation and suggest eliminating the unnecessary bits from the computation process. We propose an efficient scheme to reduce the number of computations and suggest possible circuit and system level implementations.

L. Choi, Y. Paek, and S. Cho (Eds.): ACSAC 2007, LNCS 4697, pp. 198–208, 2007.

Power optimization techniques often trade performance for power. In this work, however, we not only reduce power but also improve processor performance. Performance improvement is possible since eliminating unnecessary computations results in faster and yet highly accurate prediction. We reduce the dynamic and static power dissipation associated with predictor computations by 52% and 44% respectively while improving performance up to 16%.

The rest of the paper is organized as follows. In Section 2 we discuss perceptron background. In Section 3 we discuss the motivation. In Sections 4 we introduce our optimization. In Section 5 we explain our simulations methodology and report results. In Section 6 we discuss related work. In Section 7 we offer concluding remarks.

2 Background

The perceptron branch predictor [1] uses multiple weights to store correlations among branch instructions. For each branch instruction, perceptron uses a weight vector which stores the correlation between the branch and previously encountered branch instructions.

As presented in Figure 1, the perceptron predictor takes the following steps to make a prediction. First, the predictor loads the weight vector corresponding to the current branch instruction. Second, each weight is multiplied by the corresponding outcome history from the history vector. Third, an adder tree computes the sum of all the counters. Fourth, the predictor makes prediction based on the sum's sign. For positive summations, the predictor assumes a *taken* branch otherwise the predictor assumes a *not taken* branch.

The outcome history is essentially a bit array, in which "0"s and "1"s represent not taken and taken outcomes respectively. However, in the multiplication

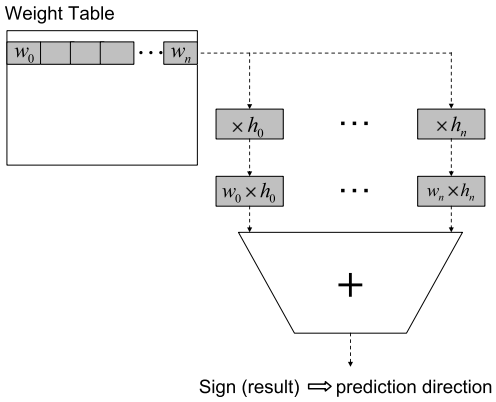

Fig. 1. The perceptron branch predictor using weight vector and history vector. The dot product of the two vectors is used to make the prediction.

process, "0"s are treated as "-1", meaning that the corresponding weight must be negated.

At the update time, the behavioral correlation among branch instructions is recorded. The predictor updates the weights vector using the actual outcome of the branch instruction. Each weight is incremented if branch's outcome conforms to the corresponding outcome history. Otherwise the weight is decremented.

3 Motivation

As presented in Figure 1, the perceptron predictor uses two vectors per branch instruction. For every direction prediction, the predictor computes the dot product of the two vectors.

The complexity of the computations involved in the dot product calculation makes it a slow and energy hungry one. This process requires an adder tree, with a size and complexity proportional to the size of the vectors and weights' widths. The wider the weights are, the more complex the summation process will be. Note that the perceptron predictor proposed in [1] uses 8-bit counters to achieve the best accuracy. Furthermore, in order to achieve high accuracy, long history lengths, resulting in long vectors, are required [1]. This also substantially increases adder tree's size and complexity.

In this study we show that the conventional perceptron predictor is not efficient from the energy point of view. This is the result of the fact that not all counter bits are equally important in making accurate predictions. Accordingly, higher order bits of the weights play a more important role in deciding the final outcome compared to lower order bits. In Figure 2 we present an example to provide better understanding.

History / Weight Values

h_i	-1	-1	1	1	-1	1	-1	-1
w_i	25	3	-5	10	2	0	-9	7
$w_i \times h_i$	-25	-3	-5	10	-2	0	9	-7
Binary	100111	111101	111011	001010	111110	000000	001001	111001

Result (All Bits) = -25 + -3 + -5 + 10 + -2 + 0 + 9 + -7 = -23 → Not Taken
Result (High Bits) = -4 + -1 + -1 + 1 + -1 + 0 + 1 + -1 = -6 → Not Taken

Fig. 2. The first calculation uses all bits and predicts the branch outcome as "not taken". The second one uses only higher bits (underlined) and results in the same direction.

To investigate this further, in Figure 3 we report how often excluding the lower n bits of each counter impacts prediction outcome. As reported, on average, 0.3%, 1.0%, 4.0%, 13.7%, and 25.8% of time eliminating the lower one to five bits results in a different outcome respectively. This difference is worst (45%) when the lower five bits are excluded for *bzip2* (see Section 5 for methodology).

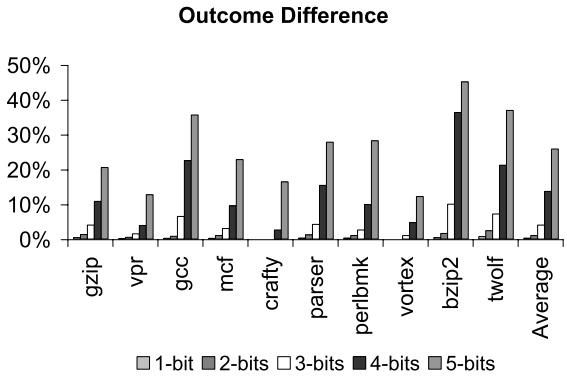

Fig. 3. How often removing the lower n bits from the computations results in a different outcome compared to the scenario where all bits are considered. Bars from left to right report for scenarios where one, two, three, four or five lower bits are excluded.

We conclude from Figure 3 that eliminating lower order bits (referred to as LOBs) of the weights from the prediction process and using only higher order bits (referred to as HOBs) would not significantly affect predictor's accuracy. We use this observation and reduce predictor's latency and power dissipation.

4 LOB Elimination

Considering the data presented in Section 3, we suggest eliminating the LOBs of the weights from the lookup and summation process. We modify the adder tree to bypass the LOBs of the weights, and perform the summation only over HOBs.

Excluding LOBs from the summation process reduces the size and complexity of the adder tree required. Therefore, a smaller and faster adder can be used. This will result in a faster and more power efficient lookup process.

As we show in this work, LOBs have very little impact on the prediction outcome at the lookup stage. However, it is important to maintain all bits, including LOBs, at the update stage. This is necessary to assure recording as much correlation information as possible. Therefore, we do not exclude LOBs at the update stage and increment/decrement weights taking into account all counter bits.

In Figure 4 we present the modified prediction scheme. The adder tree does not load or use all counter bits. Instead, the adder tree bypasses the LOBs of the weights, and performs the summation process only on the HOBs. Eliminating LOBs reduces power but can, in principle, impact accuracy and performance.

4.1 Accuracy vs. Delay

By eliminating LOBs from the lookup process, we reduce the prediction latency at the cost of accuracy loss. However, a previous study on branch prediction

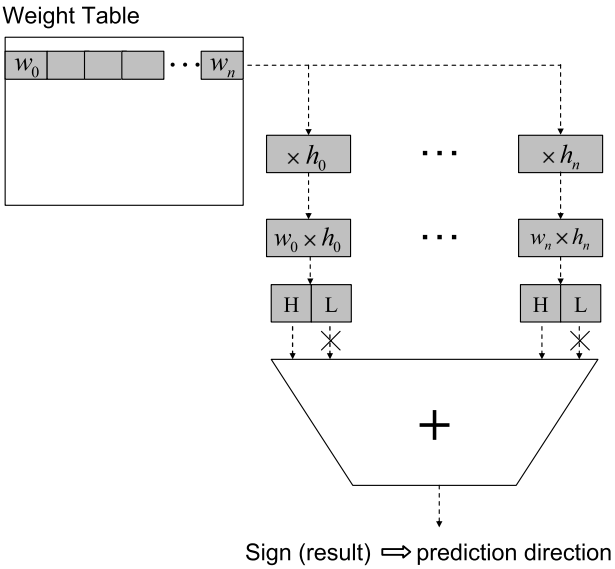

Fig. 4. The optimized adder tree bypasses LOBs of the elements and performs the addition only on the HOBs. Eliminating LOBs results in a smaller and faster adder tree.

delay shows that a relatively accurate single-cycle latency predictor outperforms a 100% accurate predictor with two cycles of latency [2].

To investigate whether the same general trade-off is true for perceptron, we study if the prediction speedup obtained by eliminating LOBs is worth the accuracy cost. In Section 5 we show that for the benchmarks used in this work the performance improvements achieved by faster prediction outweigh the cost associated with the extra mispredictions.

4.2 Power

By eliminating LOBs from the lookup process, we reduce both the dynamic and the static power dissipated by the predictor. First, fewer bits are involved in the computations, reducing the overall activity and dynamic power. Second, as we reduce the adder tree's size, we exploit fewer gates, reducing the overall static power.

As we eliminate LOBs from the computations necessary at the prediction time, reading all bit lines of the weight vector is no longer necessary. One straightforward mechanism to implement this is to decouple LOBs and HOBs. To this end, we store LOBs and HOBs in two separate tables.

As presented in Figure 5, at the prediction time, the predictor accesses only the tables storing HOBs, saving the power dissipated for accessing LOBs in the conventional perceptron predictor. Note that while we save the energy spent on wordline, bitline and sense amplifiers, we do not reduce the decoder power dissipation as we do not reduce the number of table entries.

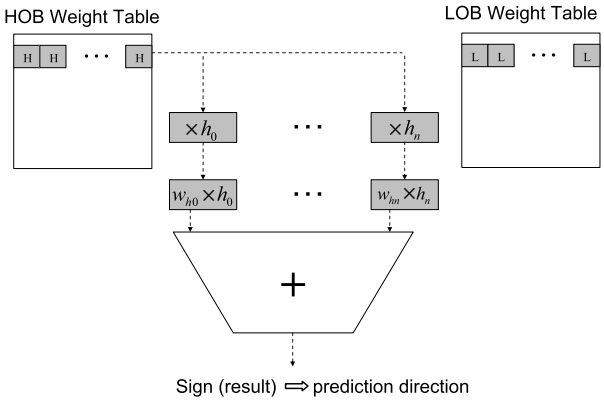

Fig. 5. Predictor table is divided to HOB and LOB tables. Only the HOB table is accessed at the prediction time.

5 Methodology and Results

For our simulations, we modify the SimpleScalar tool set [3] to include the conventional perceptron branch predictor and our proposed optimization. We use Simpoint [4] to identify representative 500 million instruction regions of the benchmarks. We use a subset of SPEC2K-INT benchmarks for our simulations.

Table 1 reports the baseline processor used in our study. For predictor configuration, we use the 64Kb budget global perceptron predictor proposed in [1].

For predictor power and timing reports, we use Synopsys Design Compiler synthesis tool assuming the 180 nm technology. We use the high effort optimization option of the Design Compiler, and optimize the circuit for delay. We simulated both the conventional and the optimized perceptron predictors.

Since we assume that table read time remains intact, for timing reports, we only measure the time the adder tree requires.

For our simulations, we assume the processor has a 1GHz frequency.

For simplicity, we use the notation of PER-n to refer to a perceptron predictor modified to eliminate the lower n bits from the computations.

Table 1. Processor Microarchitectural Configurations

Fetch/Decode/Commit	6
BTB	512
L1 I-Cache	32 KB, 32B blk, 2 way
L1 D-Cache	32 KB, 32B blk, 4 way
L2 Unified-Cache	512 KB, 64B blk, 2 way
L2 Hit Latency	6
L2 Miss Latency	100
Predictor Budget	64Kbits

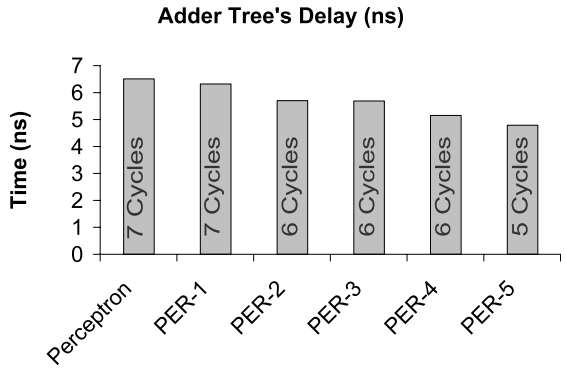

Fig. 6. Time/cycle required to compute the dot product

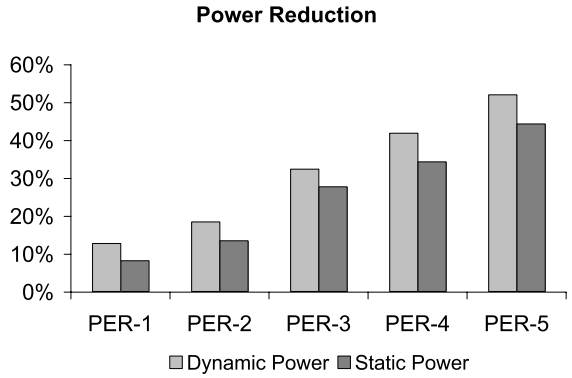

Fig. 7. Power reduction for the adder tree compared to the conventional perceptron. Results are shown for PER-1, PER-2, PER-3, PER-4 and PER-5.

5.1 Timing

Figure 6 reports time (in nanoseconds) and the number of cycles required to compute the predictor computation result. We report results for the original perceptron predictor and five optimized versions. As reported, the original predictor takes 7 clock cycles to compute the result. By eliminating one bit from the computation process no clock cycle is saved. However, removing two, three or four bits saves one clock cycle and removing five bits saves two clock cycles. We use these timings in our simulations to evaluate the optimized predictors.

5.2 Power Dissipation

Figure 7 reports the reduction in both static and dynamic power dissipation for the predictor's adder tree. Results are obtained by gate level synthesis of the circuit. Eliminating one to five bits saves from 13% to 52% of the dynamic power

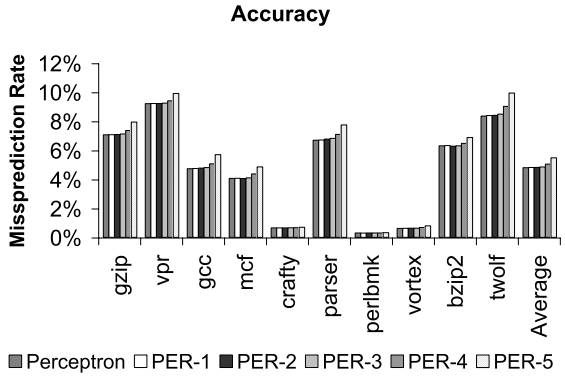

Fig. 8. Prediction accuracy for the conventional perceptron predictor and five optimized versions, PER-1, PER-2, PER-3, PER-4 and PER-5. The accuracy loss is negligible except for PER-5.

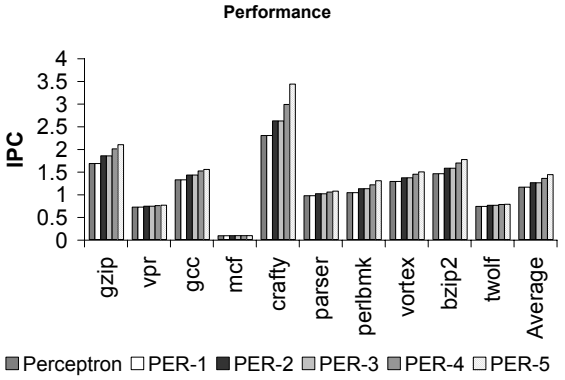

Fig. 9. Performance improvement compared to a processor using the conventional perceptron predictor. Results are shown for processors using PER-1, PER-2, PER-3, PER-4 and PER-5 predictors.

and 8% to 44% of the static power dissipation. This is the result of exploiting smaller adders.

5.3 Prediction Accuracy

As we use fewer bits to make predictions, we can potentially harm accuracy. To investigate this further in Figure 8 we compare prediction accuracy for six different predictors: The original perceptron predictor, PER-1, PER-2, PER-3, PER-4 and PER-5. As reported, average misprediction is 4.80% 4.81% 4.82% 4.85% 5.04% 5.48% for perceptron, PER-1, PER-2, PER-3, PER-4 and PER-5 respectively.

5.4 Performance

Figure 9 reports processor's overall performance compared to a processor using the original perceptron predictor. We report for five different processors using PER-1, PER-2, PER-3, PER-4 and PER-5 branch predictors. As reported, average IPCs are 1.15 1.15 1.25 1.25 1.35 1.43 for Perceptron, PER-1, PER-2, PER-3, PER-4 and PER-5 respectively.

Although the optimized perceptron predictor achieves slightly lower accuracy compared to the original one, the overall processor performance is higher. As explained earlier, this is the result of achieving faster prediction by eliminating LOBs.

6 Related Work

Vintan and Iridon [5] suggested Learning vector quantization (LVQ), a neural method for branch prediction. LVQ prediction is about as accurate as a table-based branch predictor. However, LVQ comes with implementation difficulties.

Aasaraai and Baniasadi [6] used the same technique used in this work on the O-GEHL branch predictor. They showed that power savings are possible by eliminating lower order bits from computations involved in the O-GEHL branch predictor. They also use disabling technique in [7] to improve the power efficiency of the perceptron branch predictor. They reduced perceptron power dissipation by utilizing as much resources as needed according to the branch behavior, effectively reducing overall number of computations.

Loh and Jimenez [8] introduced two optimization techniques for perceptron. They proposed a modulo path-history mechanism to decouple the branch outcome history length from the path length. They also suggested bias-based filtering exploiting the fact that neural predictors can easily track strongly biased branches whose frequencies are high. Therefore, the number of accesses to the predictor tables is reduced due to the fact that only bias weight is used for prediction.

Parikh et al. explored how branch predictor impacts processor power dissipation. They introduced banking to reduce the active portion of the predictor. They also introduced prediction probe detector (PPD) to identify when a cache line has no branches so that a lookup in the predictor buffer/BTB can be avoided [9].

Baniasadi and Moshovos introduced Branch Predictor Prediction (BPP) [10]. They stored information regarding the sub-predictors accessed by the most recent branch instructions executed and avoided accessing underlying structures. They also introduced Selective Predictor Access (SEPAS) [11] which selectively accessed a small filter to avoid unnecessary lookups or updates to the branch predictor.

Huang et al. used profiling to reduce branch predictor's power dissipation [12]. They disabled tables that do not improve accuracy and reduced BTB size for applications with low number of static branches.

Our work is different from all the above studies as it eliminates unnecessary and redundant computations for the perceptron predictor to reduce power. Unlike many previously suggested techniques, our optimizations do not come with any timing or power overhead as we do not perform any extra computation or use any additional storage.

7 Conclusion

In this work we presented an alternative power-aware perceptron branch predictor. We showed that perceptron uses unnecessary information at the prediction time to perform branch direction prediction. We also showed that eliminating such unnecessary information from the prediction procedure does not impose substantial accuracy loss. We reduced the amount of information used at the prediction time, and showed that it is possible to simplfiy the predictor structure, reducing both static and dynamic power dissipation of the predictor.

Moreover, we showed that by avoiding such computations it is possible to achieve faster branch prediction. Consequently, we improved the overall processor performance despite the slightly lower prediction accuracy.

References

1. Jimenez, D.A., Lin, C.: Neural methods for dynamic branch prediction. ACM Transactions on Computer Systems, 369–397 (2002)
2. Jimenez, D.A., Keckler, S.W., Lin, C.: The impact of delay on the design of branch predictors. In: Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33), pp. 66–77 (2000)
3. Burger, D., Austin, T.M.: The simplescalar tool set version 2.0. Technical report, Technical Report 1342, Computer Sciences Department, University of Wisconsin (June (1997)
4. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2002)
5. Vintan, L.N., Iridon, M.: Neural methods for dynamic branch prediction. In: International Joint Conference on Neural Networks, pp. 868–873 (1999)
6. Aasaraai, K., Baniasadi, A., Atoofian, E.: Computational and storage power optimizations for the o-gehl branch predictor. In: ACM International Conference on Computing Frontiers, ACM Press, New York (2007)
7. Aasaraai, K., Baniasadi, A.: Low-power perceptron branch predictor. Journal of Low Power Electronics 2(3), 333–341 (2006)
8. Loh, G.H., Jimenez, D.A.: Reducing the power and complexity of path-based neural branch prediction. In: Proceedings of the 5th Workshop on Complexity Effective Design (WCED), pp. 1–8 (2005)
9. Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M.: Power issues related to branch prediction. In: Proceedings of the Eighth International Symposium on High-Performance Computer Architecture. p. 233 (2002)

10. Baniasadi, A., Moshovos, A.: Branch predictor prediction: A power-aware branch predictor for high-performance processors. In: Proceedings of 20th International Conference on Computer Design (ICCD 2002), pp. 458–461 (2002)
11. Baniasadi, A., Moshovos, A.: Sepas: A highly accurate energy-efficient branch predictor. In: Proceedings of the 2004 International Symposium on Low Power Electronics and Design. pp. 38–43 (2004)
12. Huang, M.C., Chaver, D., Pinuel, L., Prieto, M., Tirado, F.: Customizing the branch predictor to reduce complexity and energy consumption. In: Proceedings of 33rd International Symposium on Microarchitecture, pp. 12–25 (2003)