Two Level, Perceptron, and Piecewise Linear Branch Prediction

Yitaek Hwang and Stephen Hughes

Abstract

Branch prediction is critical to the overall performance of any pipelined processor. Mispredicted branches will flush the subsequent instructions from the pipeline while the correct branch target is fetched. In this paper, we discuss various dynamic branch prediction techniques. Dynamic implementations use branch history behavior collected from the program at runtime in order to predict later branches. We first introduce Yeh and Patt's two-level branch prediction scheme, discuss its various configurations, and analyze the overall performance through a pre-existing sim-outorder implementation. Next, we introduce and implement (through sim-outorder) the perceptron and piecewise linear based schemes. In forming an overall comparison between these three predictors, it is necessary to consider practical hardware constraints. For each scheme, we analyze the percentage of mispredicted branches as a function of the hardware budget. In doing so, we hope to gain a better understanding into the arguments for and against different branch prediction mechanisms.

1. Introduction

The premise of improving branch prediction accuracy, and thereby overall performance, lies on the processor's ability to recognize patterns and speculate with high confidence. It is then logical to apply machine learning algorithms, which evolved from computational learning theory and data pattern recognition, to dynamic branch prediction. Advanced algorithms such as neural networks build complex nonlinear relationships to accurately make predictions. However, such implementations require significant hardware and computation complexity that make producing a prediction within a few cycles difficult. Therefore, simple algorithms such as the perceptron and piecewise linear predictors are both practical and effective in branch prediction. In this project, we implemented both schemes and compared the performances of each to Yeh and Patt's two-level, gshare implementation. The remainder of the paper is organized as follows. Section 2 summarizes the two-level branch predictor, which will be used as a baseline against the neural based predictors. Section 3 describes the perceptron implementation, including background information and our modifications to the SimpleScalar simulator. Section 4 reviews the piecewise linear predictor in a similar manner. In Section 5, all three predictors are compared, and the results are discussed. Finally, in Section 6, we conclude our findings and refer to literature to determine why neural based predictors have not been adopted in modern processors.

2. Two Level Branch Prediction

Yeh and Patt's two-level branch prediction scheme maintains two tables (L1 & L2), which are used in conjunction to form a prediction for the outcome of a branch instruction. The program counter for a given branch instruction indexes into the L1 table to access a Branch History Register; this holds information about the outcomes of recent branches. This history, and potentially the program counter, is used to index into the L2 Pattern Table. The L2 table stores the branch history pattern, which is used to form a prediction for a given branch. In analyzing the performance of the two level predictor, we used the

built-in BPred2Level code in *sim-outorder.c*, *bpred.c*, and *bpred.h*. This implementation uses two bits per L2 entry as a saturating counter (values 0 or 1 predict not taken, 2 and 3 predict taken). This predictor is depicted in Figure 1.

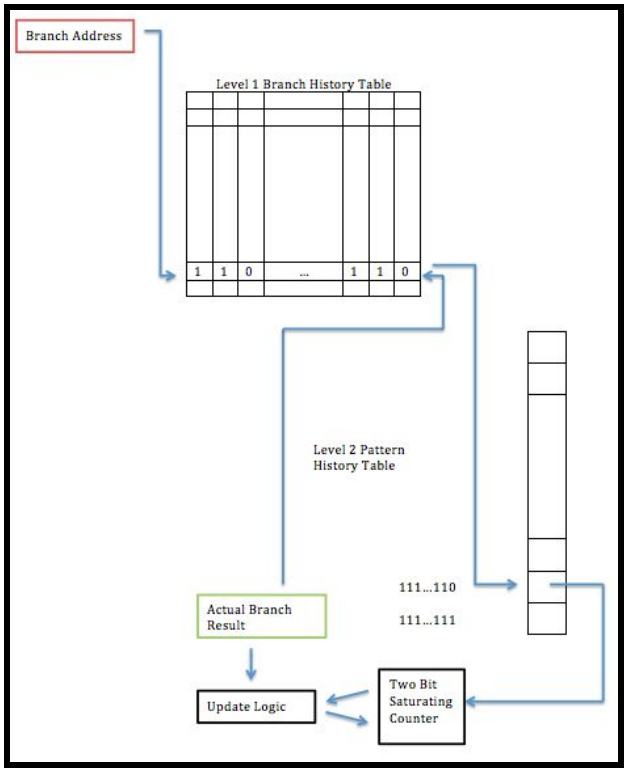


Figure 1: General Two Level Branch Prediction Scheme

2.1 Experimental Methodology

We determined to find an optimal (in terms of reducing the percentage of mispredicted branches within a reasonably sized hardware budget) configuration for the two-level scheme, as to later compare it to the perceptron and piecewise linear predictors. Various models introduced in "Alternative Implementations of Two-Level Adaptive Branch Prediction" [1] were analyzed and are described below:

GAg - Uses one global branch history register of *h* bits (L1). There are 2^h entries in the Global Pattern History Table (L2), one for each possible combination of global history bits.

PAg - Maintains a branch history register for each unique branch address. Our configuration caps the the L1 size at 512 entries to match that of Yeh and Patt's implementation. The L2 table remains the same as the GAg, meaning branch addresses with the same local history will be mapped to the same L2 entry.

PAp - Like the PAg, this model holds an L1 entry for each branch address, and we again cap this size at 512 entries. However, the PAp assigns each entry in the L1 a unique Pattern History Table in the L2. *N* entries in the L1 would yield $N*2^h$ entries in the L2.

A simplistic implementation of each scheme uses the history bits obtained from the L1 alone as an index into the local or global L2 table.

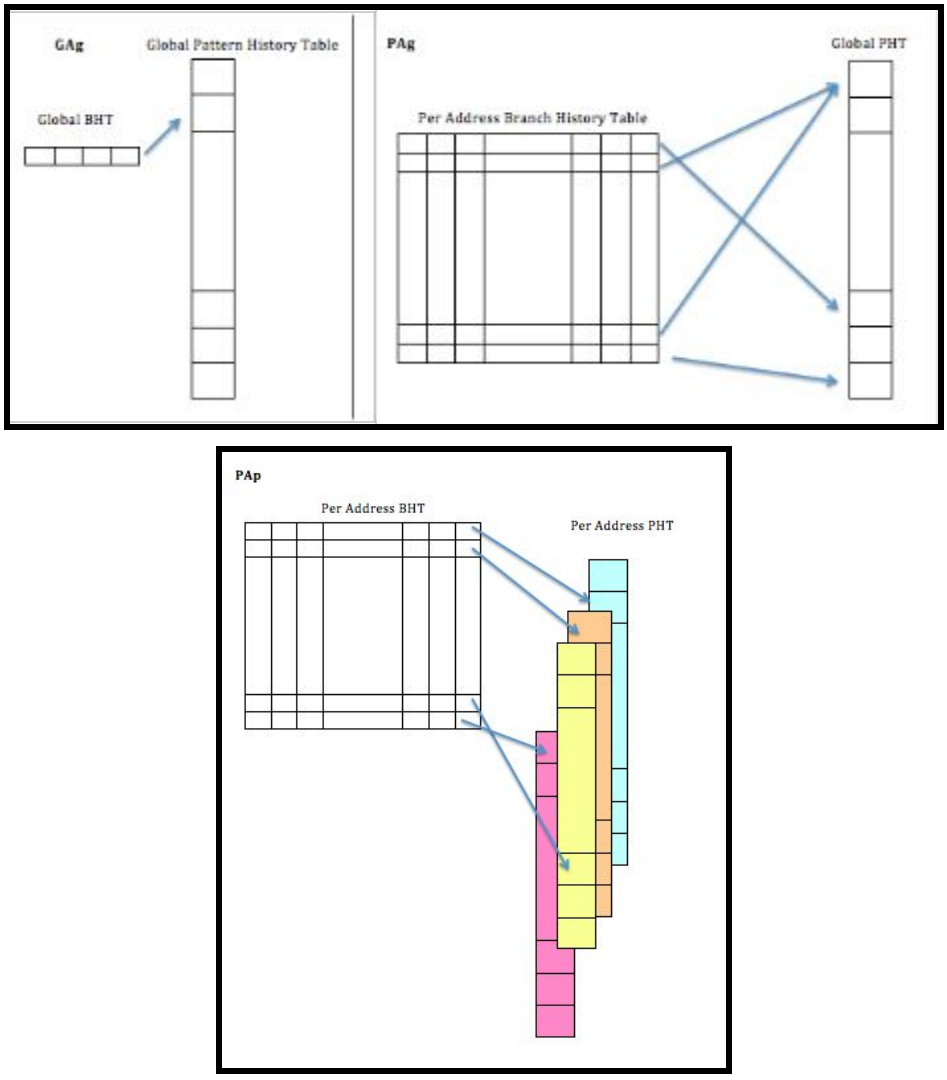


Figure 2: Two Level Branch Prediction Configurations

To determine a baseline misprediction rate for the two-level prediction scheme, each of the three models were run across five different benchmarks. For each model, the size of the history bits was varied between h = 1, 2, 4, 8, 16. Each simulation, including those later in this paper, skips the first 20 million and executes the next 50 million instructions (longer simulation were run, but percent mispredicted were comparable to 50 million instruction simulations). For PAg and PAp schemes, the L1 entry size was capped at 512 because Yeh and Patt [1] found that four-way set-associative 512-entry branch history table performed very close to an ideal branch history table.

2.2 General Performance Evaluation

We evaluated the performance of the branch prediction scheme by measuring the percentage of mispredicted branches. The results for the GAg, PAg, and PAp schemes are displayed in Figure 3.

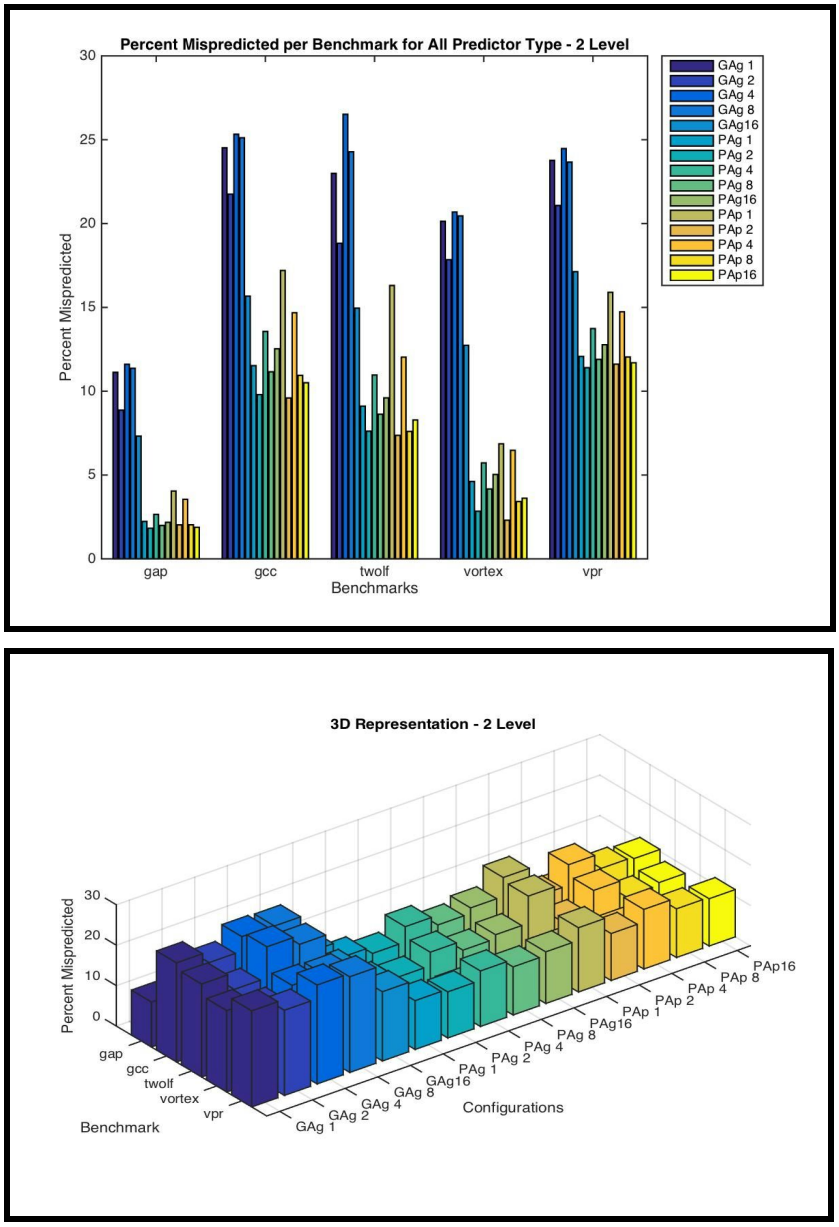


Figure 3: Percent of Mispredicted Branches for GAg, PAg, and PAp where h = 1, 2, 4, 8, 16

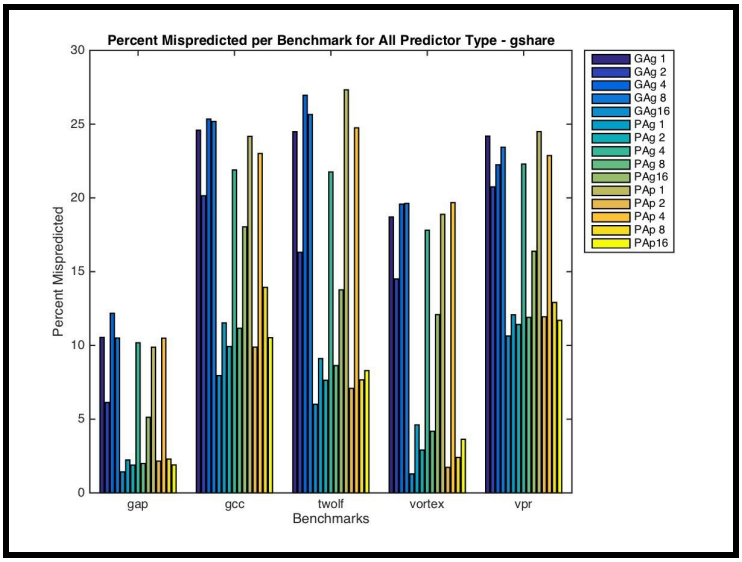
Our analysis considers the trend of the two-level prediction schemes across all five benchmarks:

GAg - In general, increasing the number of global history bits decreases the percentage of mispredicted branches. As more history is maintained, the prediction scheme is able to recognize larger global patterns. GAg performance suffers with 4-8 bit history register, because every branch updates the same history register, causing significant aliasing [1].

PAg - The best performance is seen when using 2 bit local branch history registers. Increasing the number of history bits would intuitively reduce the amount of aliasing that occurs when mapping from L1 to L2 entries. Per our results, a 2 bit local history register is sufficient in reducing the rate of aliasing.

PAp - Like the PAg scheme, we see the lowest miss rate using a 2 bit local history register. Unlike PAg, increasing the number of history bits will not reduce the aliasing from L1 entries into the L2 table (as each L1 entry is assigned its own portion of the L2 table). Therefore, increasing the history length beyond 2 bit yields small advantages to improving branch prediction accuracy.

Overall, PAp yields the best performance, followed by PAg and lastly the GAg configuration. This is due to GAg suffering from aliasing in the pattern history - essentially, different segments of code that yield the same pattern history will be mapped to the same L2 entry. To mitigate the aliasing problem in GAg scheme, we ran simulations using the gshare approach introduced in McFarling's paper [2]. Gshare xor's the branch address with the global history to provide a simple yet more effective hashing function into the L2 pattern history table.



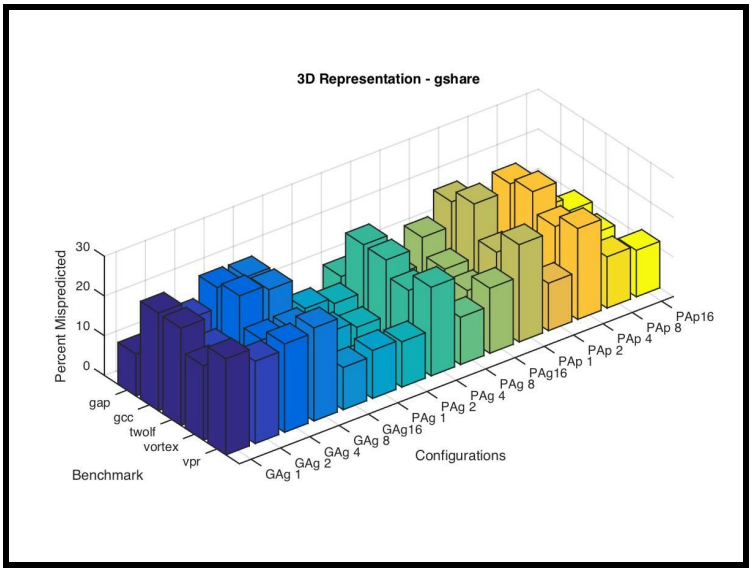


Figure 4: Percent of Mispredicted Branches with gshare for GAg, PAg, PAp where h = 1, 2, 4, 8, 16

The gshare configuration slightly improves the performance of each configuration. Interestingly, percent mispredict actually increases for PAg and PAp schemes for history lengths of 4 bits. We believe that at 4 bits, using xor to hash into the pattern history table actually increases aliasing (similar to how GAg experienced high aliasing with 4-8 bit history lengths). As history lengths continue to increase, aliasing within PAg and PAp schemes is already rare; thus, there is little benefit to using gshare for long history lengths in local branch history register configurations.

2.2 Hardware Analysis

It is necessary to consider reasonably sized hardware structures when analyzing the two-level scheme. We approximate the size of a branch predictor by considering the total amount of bits needed for storage. As a whole, the two-level prediction scheme maintains a size of

$$\#L1\ Entries * \frac{history\ bits}{L1\ Entry} + \#L2\ Entries * \frac{2\ bits}{L2\ Entry},$$

again noting that the sim-outorder simulation uses a two bit saturating counter within each L2 entry. The above equation is a simplified estimate of the hardware analysis as it ignores base costs for the storage, the decoder, the comparator, the multiplexer, the shifter, and the finite-state machine for branch prediction [1].

From [3] and [7], we determined that reasonably sized branch prediction hardware budgets lie between 1 KB and 64 KB. For the GAg scheme, we solved for the appropriate history length that would yield each hardware size, as depicted in Table 1.

Hardware Budget	History Length
1 KB	12
2 KB	13
4 KB	14
8 KB	15
16 KB	16
32 KB	17
64 KB	18

Table 1: Hardware Budget and Corresponding History Length for GAg

As seen from the hardware budget analysis, the two-level predictor cannot implement long history lengths since the hardware costs (number of second level entries) grow exponentially with history length. Shorter history lengths fail to find patterns in correlated branches when the distance between the branches exceed the length of the global history register. Variable length path branch prediction by Stark et al was one approach to avoid this capacity problem [5]. However, this implementation was impractical for real architecture due to its complex profiling and compiler-feedback mechanism. Thus, we next studied and implemented Jimenez and Lin's perceptron branch predictor where the hardware cost scales linearly with history length, thereby allowing long history lengths to find more correlation between branches.

3. Perceptron Branch Prediction

Jimenez and Lin [3] describe an application of perceptrons, a simple neural method, to predict a branch outcome. A perceptron is a linear classifier that constructs a decision by taking the dot product of a "weight" vector (w_n, associated with a PC) and a branch history register (x_i, either global or local). Thus, the output of the perceptron is $y = w_0 + \sum_{i=1}^{n} x_i w_i$. If the dot product yields a positive value, the branch is predicted taken and vice versa for negative value. Once the outcome of the branch is known, the weight vector is updated appropriately. A branch outcome that matches or does not match a given entry in the branch history register will increment/decrement the associated weight, respectively. The weights are capped at a given threshold to indicate when the perceptron weights have been sufficiently trained. This threshold value represents the confidence level of perceptron prediction and also determines the time it takes to train the perceptrons. The x_0 value is always set to 1 to give perceptron training a bias toward taken branch at initialization.

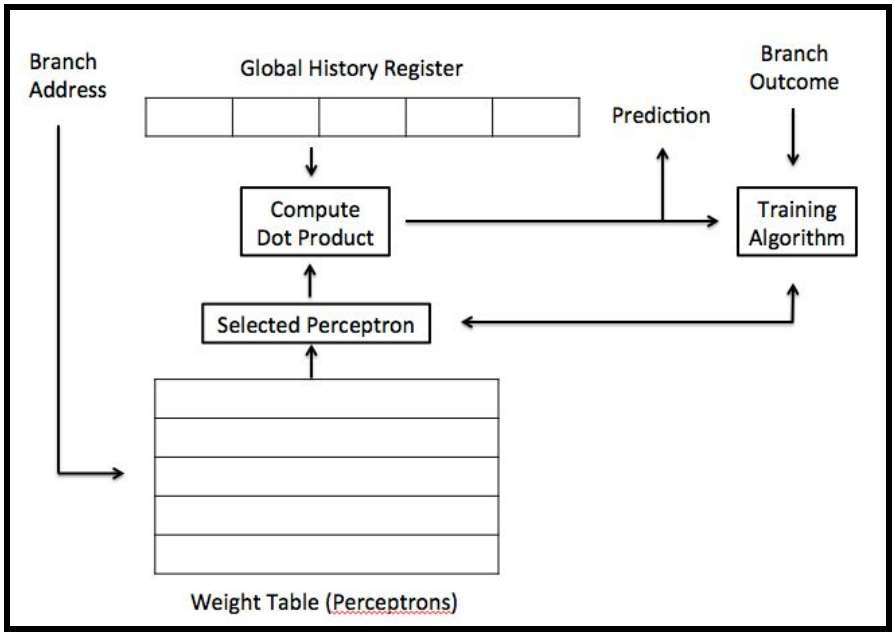


Figure 5: Perceptron Implementation Diagram

3.1 Sim Outorder Implementation

We used SimpleScalar to simulate the perceptron prediction scheme - specifically, we altered *bpred.c*, *bpred.h*, and *sim-outorder.c*. We constructed a two dimensional array (l1_size x history_size) of ints to model the branch history registers, representing taken and not taken branches with values of 1 and -1, respectively. In addition, a two dimensional array (l2_size x history_size) of ints was used to hold the various perceptron weights. Within *bpred.c*, the *bpred_dir_lookup* command was altered to incorporate the BPredPerceptron scheme - the code maps a branch address to both a BHT (level 1) and perceptron weight (level 2) entry. The dot product is taken between the branch history and the weight vector, and a positive result results in a predicted taken branch. The *bpred_update* command was also altered - the perceptron weights are updated on a mispredict, or if there is currently an insufficient amount of training within the system.

3.2 Perceptron Hardware Analysis and Experimental Methodology

To compare the performance of perceptron branch predictor to two-level gshare implementation, the following analysis was done to approximately match the hardware costs. When considering the perceptron implementation, it is important to consider the necessary number of bits to hold each weight entry. The algorithm prevents a weight from exceeding a threshold value, θ. In the worst case, it takes $\log_2\theta$ bits to represent the magnitude of the weight, and 1 bit to represent the sign. The perceptron branch predictor therefore maintains a size of:

$$\#L1\ Entries * \frac{\#\ history\ entries}{L1\ Entry} * \frac{\#\ bits}{history\ entry} + \#L2\ Entries * \frac{\#\ weight\ \ entries}{L2\ Entry} * \frac{1 + \log_2\theta\ bits}{weight\ entry}$$

where the optimal value of θ is $1.93 * (\#\ history\ entries) + 14$, determined empirically [3].

The analysis makes simplifying assumptions (ignoring the cost of digital logic associated with perceptrons) to match the parameters of the perceptron implementation to each hardware budget. The optimal history length was taken from literature, and corresponding L2 entry sizes for the global perceptron implementations were determined [3].

Hardware Budget	L1 Entries	L2 Entries	Optimal History Length
1 KB	1	109	12
2 KB	1	109	22
4 KB	1	165	28
8 KB	1	263	34
16 KB	1	493	36
32 KB	1	555	59
64 KB	1	1111	59

Table 2: Hardware Budget and Corresponding Configuration for Global Perceptron Scheme

As shown in the table, the perceptron implementation can utilize longer history lengths than two-level g-share scheme. However, with a capped hardware budget, it also reduces the number of table entries, potentially inducing an aliasing problem.

4. Piecewise Linear Branch Prediction

One of the shortcomings of the perceptron implementation is that it is only capable of learning linearly separable functions [6]. In simple terms, this means that given n-dimensional inputs, the perceptron can only perfectly learn outputs where all the true instances can be separated by the false instances by a hyplane (i.e. solution to $0 = w_0 + \sum_{i=1}^{n} x_i w_i$). Although many programs contain linearly separable branches, perceptron performs worse than two-level schemes that can learn any boolean function given sufficient training time.

Jimenez proposed a generalized neural branch predictor that subsumes perceptron and path-based predictors. Here we describe the idealized version of Jimenez's piecewise linear branch predictor. Like the other predictors described in this paper, the algorithm is dynamic and uses past branch history to predict an outcome. The algorithm constructs a linear function that acts as a predictor for each unique path that leads to a given branch.

Consider a dynamic sequence of instructions that leads to a branch - we denote this branch instruction as *I_B*. Within this sequence, there is a certain path of branches (which have either been taken or not taken)

that have led to *I_B*. The piecewise linear implementation correlates both the outcome of the last *h* branches leading to *I_B*, and for each branch its' position in the branch history, with the outcome of *I_B*. For each branch address, a bias weight is maintained in a 3 dimensional weight table at

[*I_B* address][0][0]

and is incremented / decremented each time that branch is taken / not taken. Assume we store the last *h* branches in an array noted as *BPA* (branch path addresses). To obtain the correlation between *I_B* and an instruction in the i^{th} position in *h*, a 3 dimensional weight table is indexed by

[*I_B* address][address of BPA(i)][i + 1]

Like the perceptron, the piecewise linear approach sums the initial bias weight with the dot product of the branch history register and the appropriate weights vector. Clearly, implementing a matrix like this in hardware is infeasible, as both *I_B* address and *BPA*(i) are unbounded. Therefore, a realistic implementation hashes both the *I_B* address and *BPA*(i) address such that each exists within the range of indices specified by the matrix. We denote the total *I_B* address entries as *n_size* and the total *BPA*(i) address entries as *m_size*.

4.1 Piecewise Linear Implementation

A single array is used to hold the global branch history, and an additional array is used to maintain the last *h* branches in the program's history. A three dimensional array, referred to as *weight_table* (*n_size* x *m_size* x *1+history_size*), of ints is used to hold the various perceptron weights.

A BPredPiecewiseLinear case was added to the *bpred_dir_lookup* function, which hashes down the address of the current branch instruction to serve as the first index into the matrix, which we denote as *i_index (0 <= i_index < n_size)*. First, the bias weight for the branch is obtained by accessing *weight_table[i_index][0][0]* and is the initial value in the overall sum that is computed. For each of the last *h* branch addresses, we denote their position in the branch address array as *i*. For each address, we hash the value into an index referred to as *j_index (0 <= j_index < m_size)*, and add the weight at *weight_table[i_index, j_index, i + 1]* to the overall sum. A total sum greater than zero predicts a taken branch, otherwise not taken.

We again altered the *bpred_update* command. For a mispredicted branch or matrix without a sufficient level of training, the weights in the matrix were appropriately updated. First, the bias weight is incremented if the actual branch was taken, otherwise decremented. Next, for each *weight_table[i_index, j_index, i + 1]* entry, if the global history register at index *i* agrees with the branch outcome, the weight is incremented, and otherwise decremented. Lastly, the global branch history and array of the most recent *h* branches were shifted to incorporate this most recent branch.

4.2 Piecewise Linear Hardware Analysis and Experimental Methodology

Jimenez [7] suggests that weight values remain between -128 and 127, such that it takes 8 bits to represent one weight in the 3D matrix. Accordingly, it takes

$$\frac{8\ bits}{history\ entry} * history\ length + \frac{8\ bits}{weight\ entry} * n_{size} * m_{size} * (1 + history\ length)$$

bits of storage for the piecewise linear implementation (accounting for the global branch history table and the 3 dimensional weight matrix).

The optimal piecewise linear parameters were taken from [7] for hardware budgets between 4KB - 64KB. To compare the performance of two-level gshare and perceptron to the piecewise linear implementation, parameters for hardware budget of 1KB and 2KB had to be extrapolated. Since branch matrix entries was already at the minimum point of 1 at 4KB, it was kept at 1 for both 1KB and 2KB. Next, a cubic polynominal function was fit to the data (hardware cost vs. history length) to get a least-squares curvefit with r^2 value over 0.95. The model produced the following function to approximate optimal history length for 1KB and 2KB, which in turn were used to backsolve for branch path matrix entries from above equation:

$$y = (3.144 \times 10^{-6})\, x^3 - 0.002282\, x^2 + 0.5172x + 15.34$$

where y is history length and x is hardware budget. The complete parameters per hardware budget are tabulated below.

Hardware Budget	Branch Matrix Entries	Branch Path Matrix Entries	History Length
1 KB	1	59	16
2 KB	1	119	16
4 KB	1	215	19
8 KB	2	176	19
16 KB	4	138	23
32 KB	8	118	26
64 KB	8	151	43

Table 3: Hardware Budget and Corresponding Configuration for Global Piecewise Linear Scheme

5. Overall Performance Evaluation

The optimal g-share, perceptron, and piecewise linear implementations (using a global branch history register) were run across five benchmarks, with the results depicted in Figure 6.

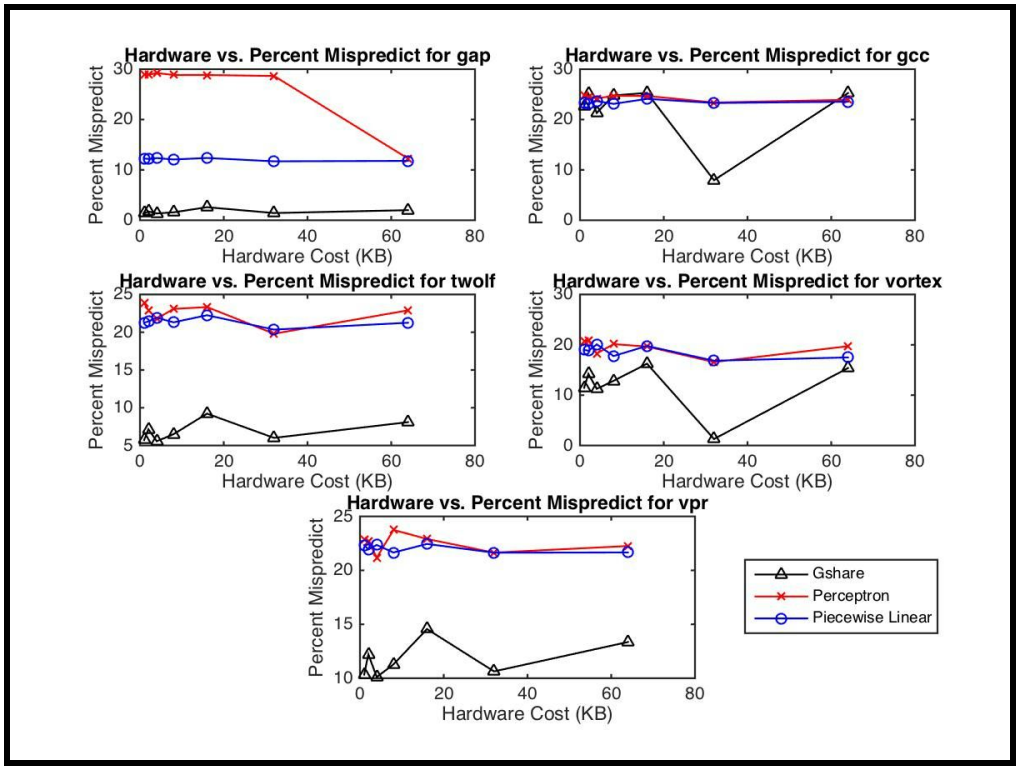


Figure 6: Hardware vs. Percentage Mispredict for GAg-GShare, Perceptron, and Piecewise Linear

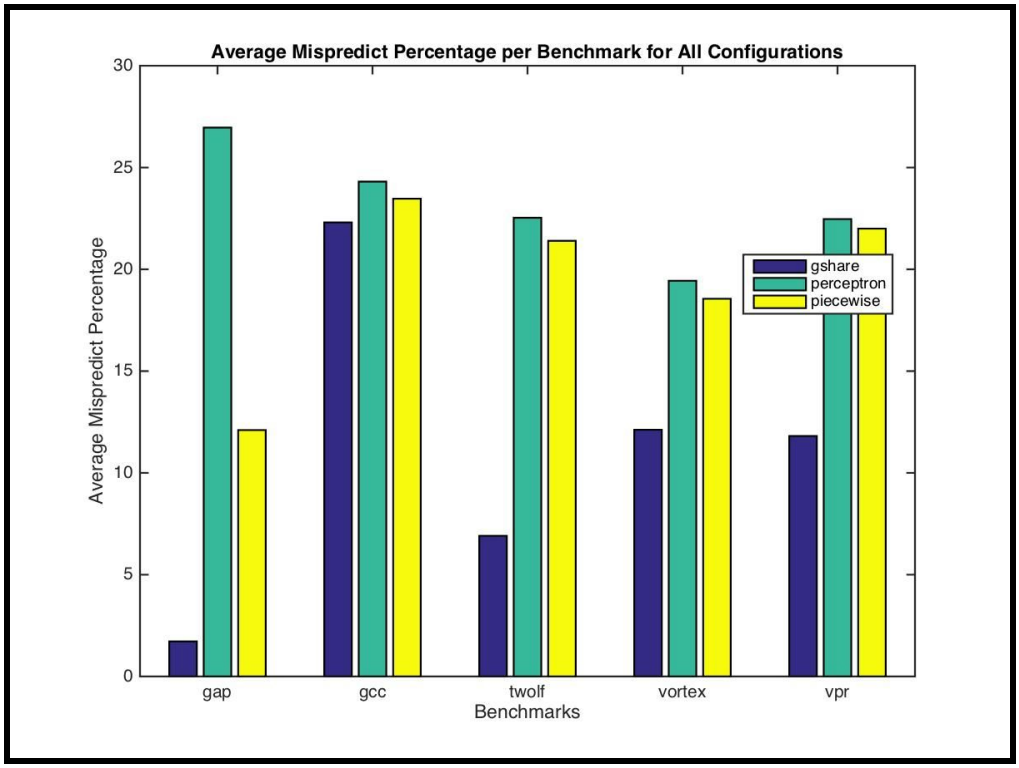


Figure 7: Average Mispredicted Percentage per Benchmark for GAg-GShare, Perceptron, and Piecewise Linear

Clearly, the gshare approach outperforms both perceptron and piecewise linear across the majority of simulation and hardware budget configurations, as depicted in Figure 6. Figure 7 aggregates the findings, and in each benchmark the gshare approach has the lowest average mispredict percentage. For a given simulation and hardware budget, the perceptron and piecewise linear scheme perform similarly. However, when aggregated, it is clear that the piecewise linear approach has a slight performance advantage over the perceptron.

Based on Jimenez and Lin's papers, we expected improved performance for the latter two machine learning implementations over the original two-level scheme. Therefore, we attempted to determine inefficiencies in our implementation to obtain better performance.

First, we suspected that significant aliasing was occurring in the branch history register as we used the default hashing function used for the two-level implementation in sim-outorder. So we varied the hashing technique which maps a branch address to a particular branch history register or matrix entry. However, the overall mispredict rate remained relatively unchanged across various benchmarks. Next, we suspected that simulations we ran were too small for both perceptron and piecewise linear weights to train properly. After increasing the number of instructions to 500 million and 5 billion, we still saw similar mispredict rates for both neural based prediction methods. We then debugged our *bpred.c* code by printing out the weight entries in the matrices to see if the weight vectors were being incremented or decremented properly. Lastly, we examined the original implementation by Jimenez and Lin to double check our logic [4].

Even after thorough debugging, our implementation of perceptron and piecewise linear predictors performed worse than gshare. We believe that the most probable explanation for the degraded performance is failing to keep track of both speculative and real global history in our implementation; Jimenez used both a speculative global history and real global history in his implementation [4]. When a branch, *B1*, is initially predicted, the speculative global history is immediately updated with the speculated prediction. Assume that a later branch, *B2*, now needs to be predicted. Additionally, assume the *bpred_update* function has not yet been called to update the real global history that would be changed to reflect the outcome of *B1*. The speculative global history allows *B2* to use the predicted outcome of *B1* to form its prediction. When the *bpred_update* function is eventually called for *B1*, the real global history is updated to reflect the actual outcome. Here, if we realize that *B1* has been mispredicted, the speculative global history is updated to reflect the last known real global history state. On a mispredict, it does not matter that *B2* has predicted using an incorrect prior history state, as both *B1* and *B2* will be squashed.

In our simulations, we failed to take this scenario into account. We presumed that the *bpred_update function* (which updates the branch history register and weights) would be called immediately after the *bpred_dir_lookup* (which forms a prediction).

However, we do note that it was encouraging to see slight performance improvements of piecewise linear predictor over perceptron implementation given the same hardware budget. Although our sample size is small, this may indicate piecewise linear scheme's ability to learn non-linearly-separable functions given the same hardware budget.

6. Conclusion and Future Work

In this paper, we evaluated three different types of branch predictors: two-level gshare, perceptron, and piecewise linear. First, the configurations for the two-level predictor were optimized and the baseline performance measurements were recorded for comparisons to neural based predictors. Next, we examined Jimenez and Lin's perceptron predictor and implemented it in SimpleScalar. Similarly, Jimenez's piecewise linear predictor was studied and coded in C.

Unfortunately, our results show worse performance for the two neural based predictors, most likely due to our failure to keep track of both speculated history and real history register. However, we did see slight improvements in piecewise linear implementation over perceptrons. Throughout this analysis, we gained insight into the fundamentals of machine learning based algorithms and several problems with each implementation.

Although power and circuit level implementations of each configurations were not covered in this project, we did look in literature to determine why modern processors do not use neural based predictors despite their high accuracy. The major shortcoming of neural based methods as cited by Jimenez and several other authors [8][9] is its complexity, both in terms of computation and hardware. Although the algorithm itself may be "simple," the computational latency of taking the dot-product or similar algorithms based on summation of weights may exceed a single cycle. As history length increases, it becomes uncertain whether the time to hash the function, load values from SRAM, and compute the dot product can be executed before a prediction is obtained [10]. VLSI and analog circuit implementation of neural based predictors have been proposed [11][12], but such designs that attempt to mitigate SRAM access delay by ahead-pipelining schemes suffer from hardware complexity in terms of history table size, pipeline depth, checkpointing overhead, as well as energy consumption and die-area constraints [8]. Although currently Andre Seznec's L-TAGE predictor is considered the most accurate branch predictor available, with advances in flash and phase-change memory cells, more complex and accurate neural predictors may be implementable in the near future with reasonable power, hardware complexity, and latency [13][14].

If given more time, we would like to revisit our implementation and modify the code to consider speculative history. Once our perceptron and piecewise linear methods show similar performance increases as reported in literature, we would like to explore other algorithms in machine learning to apply to branch prediction. We would also like to take a deeper look at hardware implementation issues, specifically regarding hardware complexity, latency, and power consumption for each of these implementations and see if neural based methods can be used with modern processors in the future as semiconductor device technology continues to improve.

References

[1] T-Y Yeh and Y.N. Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction”, 25 years of the international symposia on Computer Architecture (selected papers), p451-461, June 27-July 02, 1998, Barcelona, Spain

[2]. McFarling, Scott. “Combining Branch Predictors.” WRL-Technical Note TN-36. Jun. 1993.

[3] Daniel A. Jiménez , Calvin Lin, “Dynamic Branch Prediction with Perceptrons.” Proceedings of the 7th International Symposium on High-Performance Computer Architecture, p.197, January 20-24, 2001

[4] Daniel A. Jimenez, Calvin Lin, perceptron.c. University of Texas at Austin. 2001.

[5] Stark, J., Evers, M., Patt, Y. N. 1998. Variable length path branch prediction. In Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, 170–179.

[6] Faucett, L. 1994. Fundamentals of Neural Networks: Architectures, Algorithms and Applications. Prentice-Hall, Englewood Cliffs, N.J.

[7] Daniel A. Jimenez. “Idealized Piecewise Linear Branch Prediction” Journal of Instruction-Level Parallelism 7 (2005) 1-11.

[8] Daniel A. Jimenez, Gabriel H. Loh. “Controlling the Power and Area of Neural Branch Predictors for Practical Implementation in High-performance Processor.”

[9] Guangyu Shi. “Constructing Neural Branch Prediction with memristive Device.”

[10] Chi Ho Yue. Ilya Katsnelson. “Applications of Machine Learning Techniques to Systems.” Stanford University. EE392C: Advanced Topics in Computer Architecture - Polymorphic Processors.

[11] A. H. Kramer, “Array-based analog computation,” IEEE Micro, vol. 16, no. 5, pp. 20–29, October 1996.

[12] O. Kirby, S. Mirabbasi, and T. M. Aamodt, “Mixed-signal neural network branch prediction,” unpublished manuscript.

[13] Andre Seznec. “The L-TAGE Branch Predictor” Journal of Instruction-Level Parallelism 9 (2007) 1-13.

[14] Renee Amant, Daniel A. Jimenez, Doug Burger. “Low-Power, High-Performance Analog Neural Branch Prediction.” Proceedings of the 41st Annual IEEE?ACM International Symposium on Microarchitecture.