Feature **A**dded **B**ranch **P**rediction **I**n **M**ultilevel **P**ipelines

Jam Jenkins and Jeff M Phillips

December 1, 2003

Abstract

This report presents a novel extension to neural branch prediction methods. These techniques are highly accurate, encode a large amount of history data, and can be made to execute in 1 cycle with pipelined branch prediction techniques. However, the flexibility of the neural technique is ignored in previous work. Therefore, a demonstration of how additional information can easily be incorporated into this scheme is presented and tested. This report extensively tests a particular additional feature–an independent hash of the PC–at a fixed hardware budget–2 Kilobytes. Other features which could be used as input are suggested. This reports presents similar results to previous work and presents some new findings.

1 Introduction

Fast accurate branch prediction is essential in modern pipelined processors. Pipelining instructions breaks the execution of a single instruction of several shorter cycles instead of one longer cycle. The instruction executes one stage of the pipeline at every step, while other instructions are also being executed, but at different stages of the pipeline. This gives enormous speedup, but as the cycles get shorter, determining whether a branch is taken or not taken on the next instruction is pushed several stages into the pipeline. Since a branch occurs about every 7 instructions, waiting for the branch value to be computed wastes most of the efficiency saved by pipelining. Predicting the branches (even only 75% of the time), leads to major gains in efficiency. When false predictions occur, the pipeline needs to be flushed up to a certain point, but the cost of this is often not much more than doing no prediction. The faster and more accurate the branch predictions get, the faster the overall processor speed becomes.

FABPIMPs in Perspective Our work is primarily based on work by Daniel A. Jimenez. In [5] Jimenez describes a very accurate and fast technique for branch predicting using a single perceptron with the branch history table (BHT)–a fixed-length boolean list of when branches

have or have not been taken. These inputs are stored as 1 for taken and -1 for not taken. In short, each element of the BHT can be multiplied by a weight value to get a score. These scores are summed, and if they are positive, the branch is predicted taken, if nonpositive, it is predicted not taken. This can be quickly computed using a Wallace Tree [2] adder in about 3 cycles. Jimenez suggests that this should be used as the second part of a hybrid branching system, where a 1 cycle branch predictor makes an initial guess and this more accurate branch predictor refines that guess after 2 more cycles. The addition of high speed arithmetic to the perceptron has made it promising for use in future architectures, and it continues to be the most accurate single-component branch predictor in the literature [4].

Majumdar and Weitz [6] implemented a complex 3 layer neural network for branch prediction. They use the output of the gshare branch predictor as the input to their neural network and use back propagation in order to train the system. Their neural network does on average outperform gshare, but a prediction using their system is estimated to take 8 cycles (due to multiplications and table lookups), and 12 cycles for updates. They indicate "this implementation seems infeasible to use in practice" under the current hardware constraints.

Jimenez extended his earlier perceptron to include path-based information. In addition to branch history, he hashes together previous branch target addresses as an additional input into the perceptron. Out of the 17 SPEC CPU benchmarks he uses (all of SPEC CPU 2000 and SPEC CPU 95, duplicates removed), his approach outperforms on average all of the branch predictors in his tests (gshare.fast, Fixed-Length Path Predictor, 2Bc-gskew Global/Local Perceptron Predictor) and performs best in 9 of the 17 benchmarks [3]. Our perceptron predictor differs from Jimenez's in that our predictor uses the branch address rather than the branch target address.

FABPIMP Overview The proposed neural predictor uses a more diverse set of inputs to the perceptrons, but keeps in mind the hardware speed necessary for these tests to be feasible. The implementation proposes within each set of perceptrons, using as additional inputs the PC (program counter, which describes the location in the source code where this branch lives, a unique identifier) hashed to a few bits. This will allow the predictor to space-efficiently further differentiate between different branches evaluated on the same set of perceptrons. We also discuss the effectiveness of using differences in instruction number from previous branch as input to the perceptrons, but due to limited shared hardware available for testing, no formal tests of this idea are presented.

Section 2 of the report will more thoroughly explain the previous work in neural branch prediction. Section 3 will describe the FABPIMP branch predictor and its advantages. Section 4 will detail how it was implemented and tested. Section 5 will highlight the important results from our tests. And Section 6 will discuss the importance of these new techniques and possible future work.

2 Neural Branch Prediction

Neural branch prediction uses the adaptive and flexible behavior of neural networks [1] to predict whether a processor level instruction will take a branch or not using the limited history and state available to the processor. They are also constrained by a fixed hardware budget in which information can be stored and a speed requirement related by the number of cycles necessary to make the prediction. Neural networks can efficiently store a large amount of data and through a clever organization by Jimenez [5, 3], can be made quite efficient, if they are restricted to single level neural networks.

The only described implementation [5] which takes into account the space and time requirements, uses only the branch history table as input to the perceptron network. **Algorimth** 1 gives psuedo code for the neural branch prediction process.

Algorithm 1 Neural Prediction

```
1: h = HASH(PC)
2: lookup preceptron h in preceptron table
3: sum = 0
4: for each t in BHT do
5:    sum+ = h.BHT(t) · h.weight(t)
6: end for
7: if (sum < 0) then
8:    return NOT TAKEN
9: else
10:   return TAKEN
11: end if
```

More explicitly, a neural branch predictor keeps several perceptrons (as many as the hardware budget will allow). Tests showed that using only a single perceptron could not outperform always predicting that the branch was taken. At the beginning of each branch prediction, a perceptron in the table is indexed with some hashed value of the PC. This is later illustrated in **Figure** 2. In the specific set of perceptrons, each weight corresponding with certain depth D in the branch history is added or subtracted from the sum based on whether that corresponding branch in the BHT was taken or not. Although, the implementation is quite different, the calculation is equivalent to setting all taken branches to 1 and all not taken branched to -1. Then the output value is equivalent to:

$$sum = \sum_{d=0}^{D} BHT(d) \cdot weight(d)$$

The prediction is decided by whether or not the resulting *sum* is positive (**taken**) or nonpositive (**not taken**).

This can be more clearly seen in the example in **Figure** 1. This example is shown with a depth 4 branch history (starting, from least recent, ⟨ **taken**, **not taken**, **taken**, **not taken** ⟩). Each weight can only be held in 5 bits, so no weight can be greater than 15 or less than −15. In this example the next branches are predicted as **taken**, **not taken** and in actuality are **taken** both times.

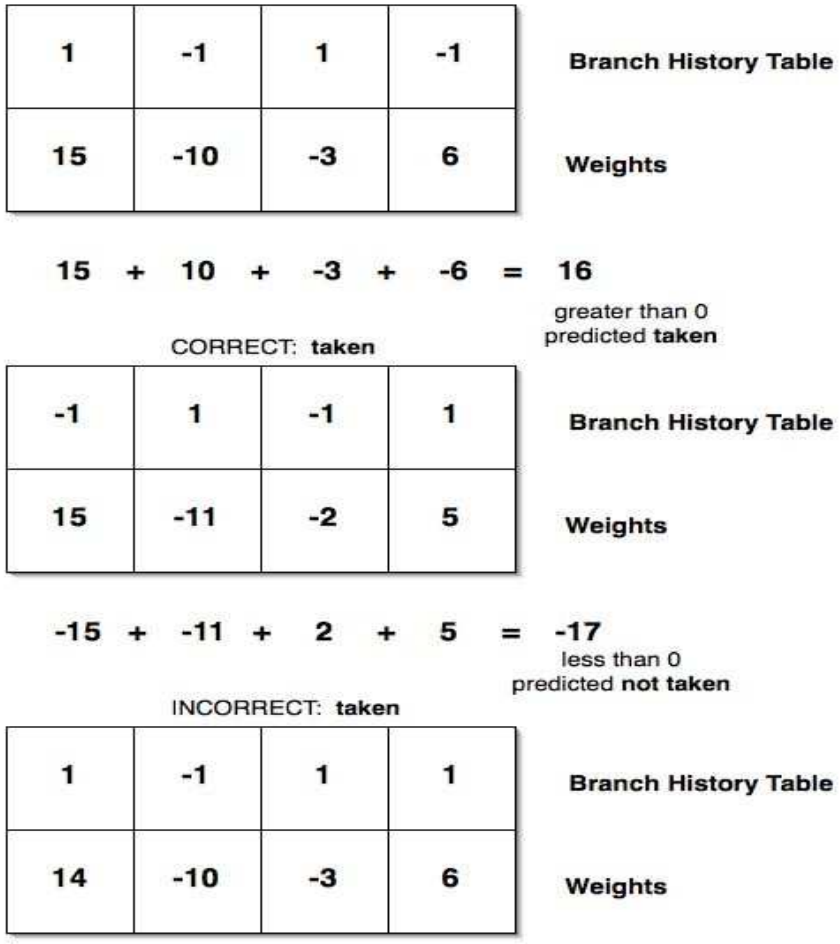

Figure 1: Example of nonFABPIMP Neural Branch Predictor

On an update, the weights are changed in such as way so that if the exact same prediction was made again, each weight/history pair would give a value 1 greater than the last prediction. Then each item in the branch history is shifted to 1 greater depth. This calculation is easily made by the equivalent of adding the old history bit ($BHT(d)$) times the actual branch direction b to the old weight where each $BHT(d)$ bit and b bit are 1 for **taken** and -1 for **not taken**, as seen in **Algorithm** 2.

The update step is out of the critical path and the prediction steps can be accomplished in 3 cycles: 1 for table lookup, and 2 for computing the sum and sending it to the right part of the

Algorithm 2 Neural Update

1: **for** $d = 0$ **to** D **do**
2: $weight(d) += BHT(d) \cdot b$.
3: **end for**

processor pipeline.

3 FABPIMP algorithm

The FABPIMP branch prediction technique works very similarly to **Algorithm** 1, except in Step 4-6, it has weights stored for information bits other than just what is in the BHT. Specifically, these bits may represent a hashed value of the current program counter, or previous program counters. If each PC here is hashed to m bits, then if a history of h hashed values of PC counters are kept, then $m \cdot h$ additional weights are stored and used in the branch prediction for each perceptron.

Since, the PC is already hashed to determine which perceptron to compute the branch prediction with, if the same hash function is used, then all h hash bits in a specific perceptron will always have the same value. But as long as these two hash functions are independent, then this will allow the perceptrons to effectively store more different sets of branches, only some sets are grouped together and use similar history weights but are given some individual bias-like weighting. For notational purposes, label the first hash function which points to a perceptron $hash1$ and the second hash functions which gives input to the perceptrons $hash2$. These can easily be implemented to be independent by letting $hash1$ grab the first few relevant bits and $hash2$ grab the next few relevant bits.

Analysis of data density Without using $hash2$, with a fixed hardware budget H in bits, the branch predicition has p perceptron, a depth of b BHT, and weight which saturates at s bits. This describes the equation $H = (p)(b)(2s)$. Since optimal values for b and s can be found empirically as in [5], this gives us $p = H/(2bs)$. However, if we distinguish additional sets of branches within perceptrons by using $hash2$ to add inputs in to a specific perceptron, a new equation is described. Let h be the number of PC values hashed with $hash2$ and used as perceptron inputs, letting $hash2$ output 4 bits. Now $H = (p)(2s)(b + 4h)$, where our total number of sets of perceptrons distinguished is $\hat{p} = p \cdot 2^4$ as long as $h \geq 1$. So with $h = 1$, $\hat{p} = 2^4 H/(2s(b+4)) >> H/(2sb)$, a big win. Also, since using an independent hash function for $hash2$ may rely on some less important bits, this may leave some of the sets of branches which are distinguished unused. But because they are only hidden inside of a perceptrons, this space is not wasted, it just makes the other PC values which hash to that perceptron more accurate.

An example of the data layout for this two-level hashing is outlined in **Figure** 2. In this figure, each pair of columns of the rectangle represents a set of perceptrons. The BHT bits are on

the left along with a *bias* bit which is always 1 or **taken** and the *hash*2 output values. The *bias* bit and the *hash*2 output values do not need to all be explicitly stored because the most recent *hash*2 output value can be computed in parallel with the *hash*1 value, and the *bias* bit is always the same.

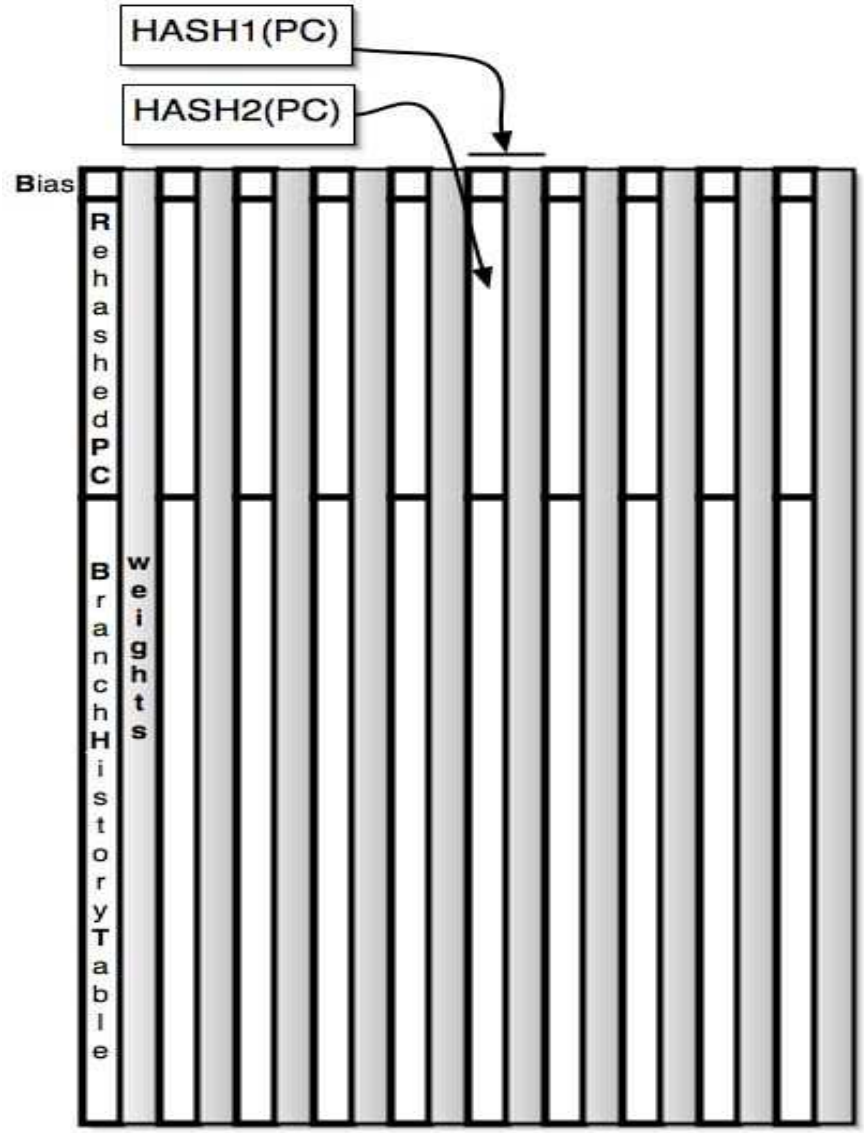

Figure 2: Data Layout of FABPIMP Neural Branch Predictor

An example of the FABPIMP algorithm can be seen in **Figure** 3. In this example a 4 deep branch history (starting with history, from least recent, ⟨ **taken**, **not taken**, **taken**, **not taken** ⟩). The saturation point of a weight, s, is 4 bits, so no weight can be greater than 15 or less than -15. The PC is rehashed to 2 bits which are on successive calls ((**taken, not taken**), (**taken, taken**), (**not taken, not taken**)). And the bias bit is always 1. The weights begin at

$\langle 15, -10, -3, 6, 4, -3, -2 \rangle$ for the BHT, rehashed PC and bias bits respectively. In this example the next branches are predicted as **taken**, **not taken** and in actuality are **taken** both times.

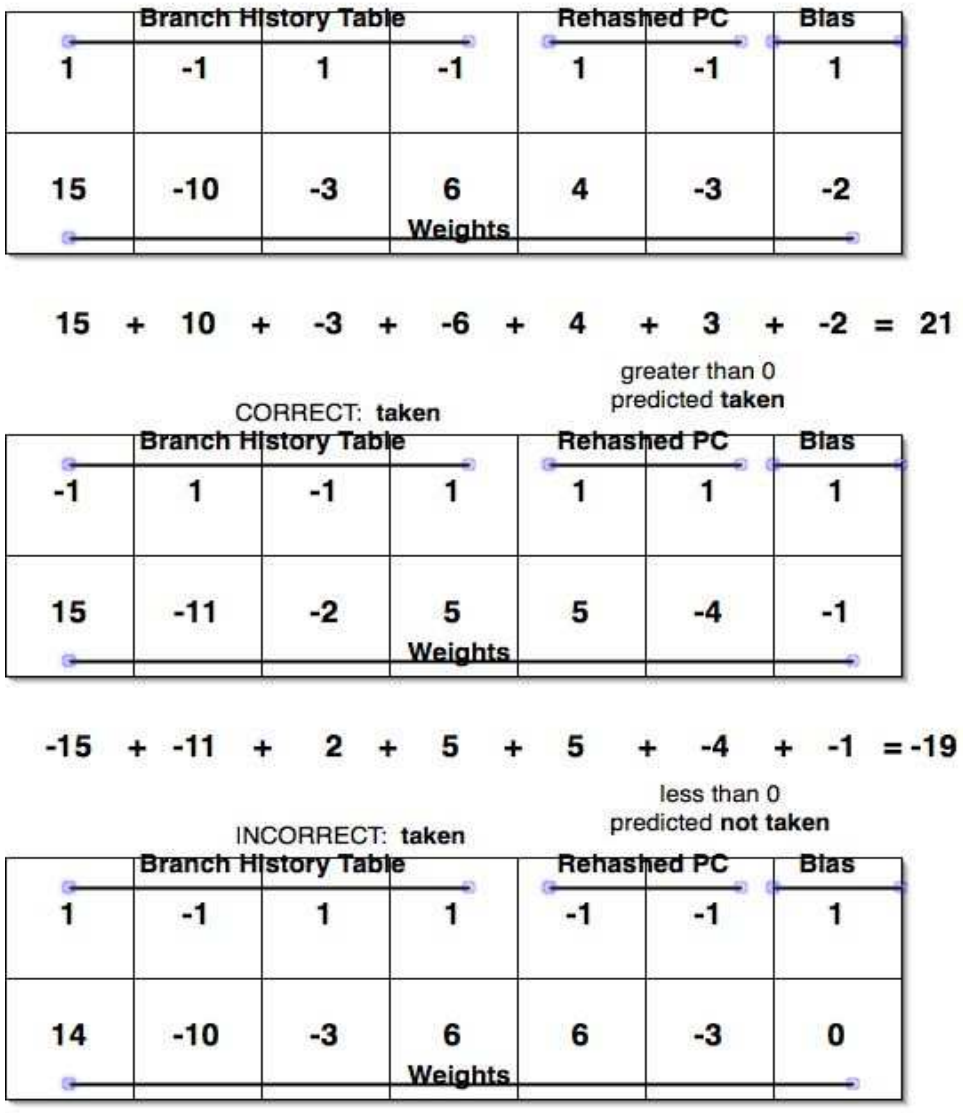

Figure 3: Example of FABPIMP Neural Branch Predictor

Other Features not Added Other features from the state of the program can also be easily incorporated into the perceptron to aid in the branch prediction. Of course, adding them is a tradeoff between space and increased accuracy. Another viable feature to add, which due to time and testing hardware limitations was not tested, is the difference between instruction numbers of successive branches predicted. This date could be kept globally with weights kept for each individual perceptron. This feature would hopefully give additional information about

the path of the branches taken. Jimenez has quite recently found success in a similar path-based technique [3].

4 Implementation

The Feature Added Branch Predictor In Multilevel Pipeline simulator was written in Java and designed to target all instruction set architectures with branch instructions. In this way our simulator could be verified on the largest possible set of benchmarks available, including those for not yet released ISAs.

The input to our simulator is a file containing all branches executed. This is somewhat of a simplification from what branches will actually be encountered during normal execution due to the exectution of mispredicted code. This simplification enables a fair and independent evaluation of the prediction of branches not skewed by the prediction of branches which would not affect execution time.

The input file contains 3 pieces of information about each branch - the instruction number, taken/not taken, and address of instruction. The instruction number reperesents the logical order in which the branch was taken. This information is not currently used, but can be used in the future to determine how the hit rate is affected by the proximity of branch instructions within an execution (which could be of particular importance in a multi-stage pipeline, or in general when branches tend to be processed out of order). Taken/not taken is simply 1 for **taken** and 0 for **not taken**. The address of the instruction is the location of the branch within the program code.

The FABPIMP works by reading in each branch, predicting its outcome, and comparing the predicted result against the actual result read from the input file. The results of the comparison are used both to update the perceptron and keep a running total used for computing the hit rate. This hit rate is output every 1 million instructions and finally at the termination of reading the input file.

The perceptrons of FABPIMP are configurable using four parameters which are indicative of the projected hardware budget necessary for implementing the predictor in hardware. The first parameter s is the maximum magnitude (either positive or negative) that each perceptron weight may realize. The branch history table parameter b describes the number of previous branches used as inputs per perceptron. The order of these inputs are always maitained by how recent they have been taken/not taken. The third parameter p is the number of pereceptrons. The goal of the FABPIMP is to maintain a separate perceptron for each branch instruction address. Hardware limitations of course preclude this, so perceptrons must be reused and shared. The final parameter h describes the number of previous branch instruction addresses used as perceptron inputs. This is used in order to indicate the path taken to the current branch being predicted.

For each benchmark, 60 sets of parameters were chosen to satisfy a $2K$ hardware budget.

Because $hash1$ was implemented by stripping the lower $\log p$ from the address, we restricted our choice of p to powers of 2. The choice of s was similarly restricted to powers of 2. Then $b + 4h$ must be a power of 2 as well. These restrictions limited the reasonable number of tests to 60.

After designing 60 combinations of parameters, we chose three diverse benchmarks to examine the affect of the parameters on the hit rate: anagram, gcc, and go. In addition to addressing the relationship and effect of our choice of parameters, our choice of benchmarks allowed us to compare our FABPIMP against other known branch predictors through the use of simplescalar. We constructed a driver program to run these series of tests over several days on several UNIX workstations within the Duke Computer Science Department. The output from these runs was used to measure the influence of the parameters and the performance of FABPIMP. This will be further discussed in the results section.

5 Results

FABPIMP was tested against 4 other branch predictors on the benchmarks go, gcc, and anagram. The results for the *twolev*, *bimod*, and *hybrid* branch predictors were gathered by running simple scalar with the appropriate command line arguments. The result for Jimenez's perceptron branch predictor and FABPIMP were generated using the testing framework described again where the hit rate reported is taken from the most effective set of parameters. Perceptron is the most effective with a value for h of 0 and FABPIMP's hit rate was gathered by the best hit rate parameters with a non-zero value of h.

	anagram	gcc	go
hybrid	.9785	.9062	.7976
twolev	.9766	.8849	.7809
bimod	.9616	.8389	.7220
perceptron	.9662	.7848	.6992
FABPIMP	.9663	.7810	.6974

Surprisingly, the brach predictors implemented in SimpleScalar outperformed both *perceptron* and FABPIMP. Jimenez [5] mentions that his *perceptron* branch predictor was designed to be part of a hybrid system, and this may increase its hit rate as it did for the branch predictors in SimpleScalar. The details of this hybrid implementation of a perceptron hybrid branch prediction where not focused on in prior work and are outside the scope of this project. Due to these results further analysis will focus on the comparison between *perceptron* and FABPIMP.

To compare FABPIMP and *perceptron* branch predictors the correlation of each of the four input parameters [number of perceptrons, saturation value of weights, BHT depth, and path length of rehashed PC] with hit rate was taken. Graphs depicting this correlation are plotted in **Figures** 4, 5, 6, and 7. As seen in **Figure** 4 the number of perceptrons used in the branch predictor is directly proportional to the hit rate and has a high correlation. Conversely, the saturation value

of the weight was inversely proportional to the hit rate. This pattern may be due to the tradeoff necessary to get a large number of perceptrons, the saturation value of the weights needed to be reduced in order to stay within the same hardware budget. Also, a low saturation value allows the perceptrons to easily adapt to quickly changing environments. Surprisingly, the depth of the BHT had virtually no correlation with the hit rate, as is evident in **Figure** 6. Also notable, the path length stored by rehashing the PC value for additional inputs to the perceptrons also had no correlation with the hit rate. This indicates that the path as indicated by the branch instruction address has little effect upon the performance of the perceptron. Perhaps other explorations of path-based input should be explored such the difference in instruction number.

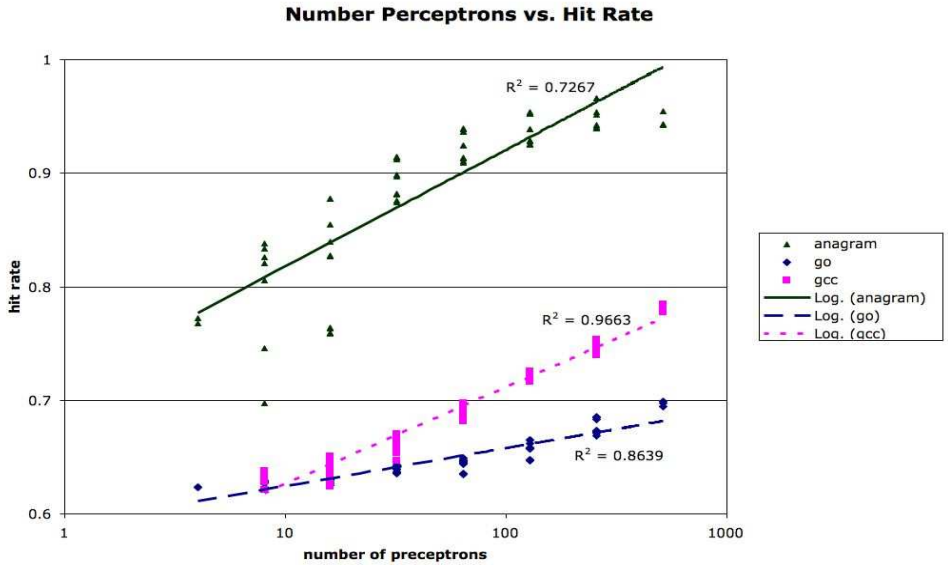

Figure 4: Correlation of the Number of Perceptrons

Hit rate was exclusively used for comparison between different branch predictors. This is not entirely fair. Different branch predictors may be more accurate, but take many more cycles, such as the predictor described in [6]. However, recent research [4] suggest that many branch predictors may be precomputed or pipelined so that they provide only one cycle delays, despite being increasingly complicated. It appears FABPIMP fits into this catagory of branch predictors, with mild modifications. The limited number of possible different immediate states could be precomputed and stored in a table, for quick look up. This would require a slightly larger space overhead than a *perceptron* predictor, but not significantly more. The implementation of this is again out of the scope of this project. This could easily implemented in Java, but would provide no further interesting analysis.

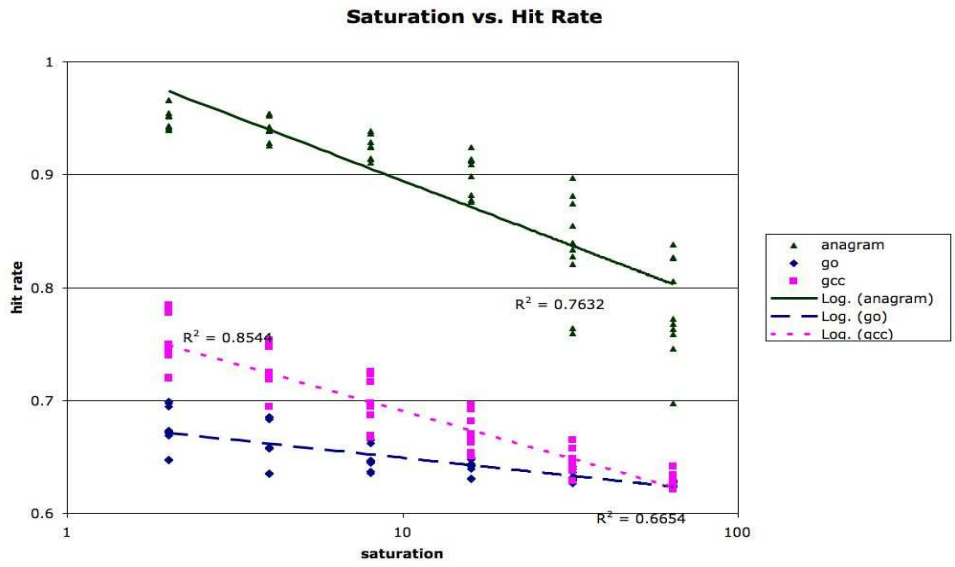

Figure 5: Correlation of the Saturation Value

6 Discussion and Future Work

This report demonstrate that the design of perceptron type branch predictors is not yet fully realized, nor understood. Promising results in this area from Jimenez [5, 3] suggests further research should be performed, particularly using the selection of parameters correlated with the hit rate.

With more time and resources, more benchmarks could be tested on a broader range of parameters and inputs to perceptrons.

References

[1] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 2000.

[2] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithm.* McGraw Hill, 1990.

[3] Daniel A. Jimenez. Fast path-based neural branch prediction. *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, 2003.

[4] Daniel A. Jimenez. Reconsidering complex branch predictors. *Proceedings of Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, February 2003.

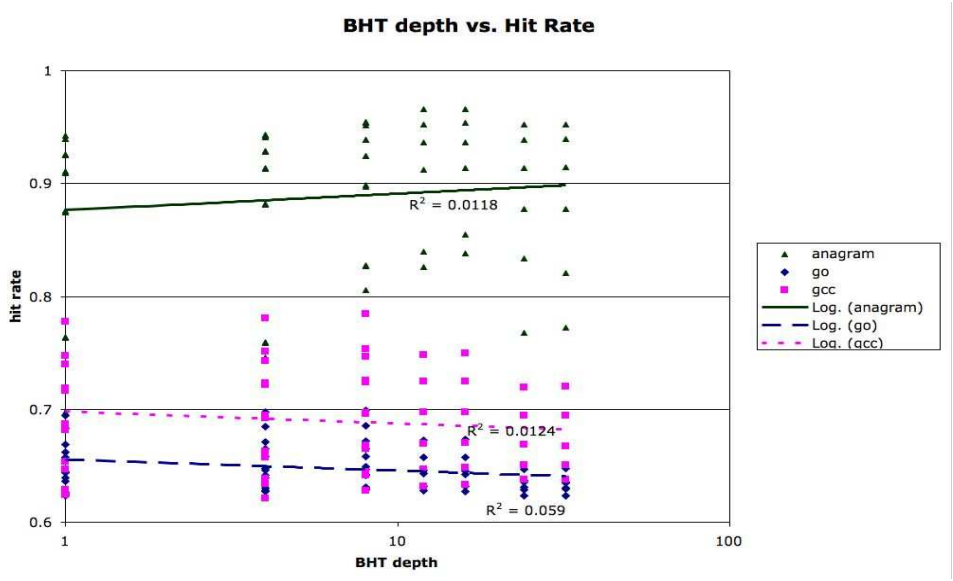

Figure 6: Correlation of the Depth of the BHT

[5] Daniel A. Jimenez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20-4, November 2002.

[6] Rupak Majumdar and Dror Weitz. Branch prediction using neural nets. *Unpublished Manuscript: Department of Electrical Engineering and Computer Science, University of California, Berkeley.*

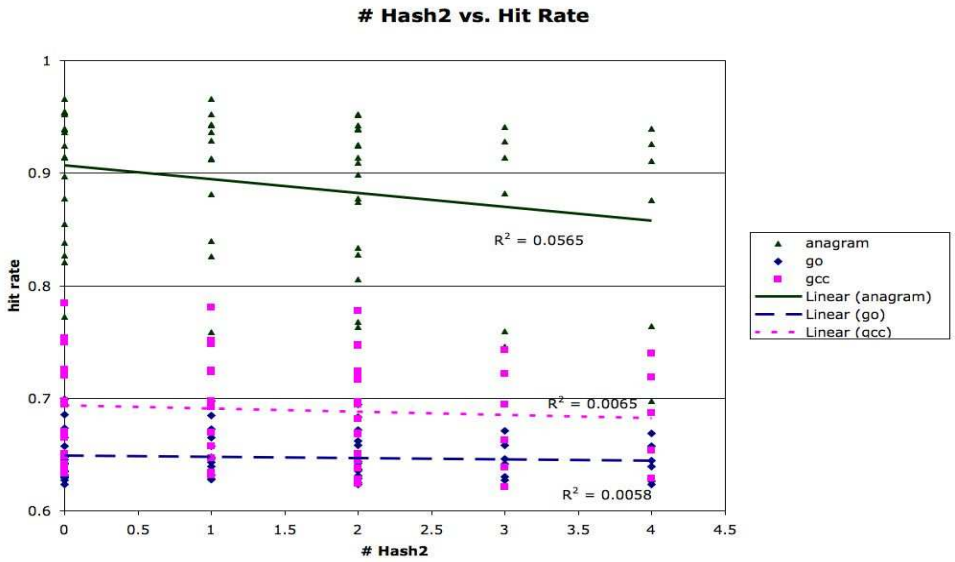

Figure 7: Correlation of the Number of PCs Rehashed