Analysis of Branch Prediction

A brief study

Anirudh Badri 1410110054 Atish Majumdar 1410110081 Prasanna Natarajan 1410110298 Vedant Chakravarthy 1410110489

***Abstract*—Modern procedures have large and complicated pipelines that take immense effort to stop and restart. In order to ensure the efficiency of the processor, steps must be taken to prevent such stalls in the pipeline. Dynamic branches are notorious for causing stalls since it is only in the late stages of a pipeline that we know where the next instruction can be found. As a result, to prevent wastage of CPU cycles, modern compilers extensively use a method known as branch prediction. This paper will discuss different methods and techniques involved in branch prediction.**

i. INTRODUCTION

Branches change the normal flow of control in programs in unexpected manner which interrupts the normal fetch and issue operation of instructions. To resume the normal fetch, it takes considerable number of cycles until the outcome of branches and new target is known.

Branches are very frequent in general purpose programs – around 20% of the instructions are branches – and simplest solution to deal with branches is to stall the pipeline. Stalling pipeline doesn't violate the correctness of program. However, it does degrade the overall IPC which is even severe in longer pipelines. In order to maintain the high throughput branch prediction in modern high performance microarchitecture has become essential.

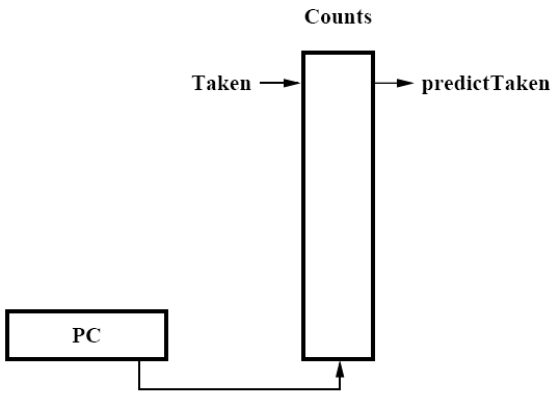
Static branches are not hard to predict and in most of the systems, compilers can provide very good coverage for such branches. Branches whose outcome is determined on run-time behaviour are difficult to predict using compilers. In general, the branch outcomes are not random and they do have some sort of bimodal distribution. For such dynamic branches it is advantages to use a prediction mechanism which adapts in runtime and captures the run-time state of the system in order to make the predictions. Although dynamic branch prediction techniques do provide very good prediction accuracy but there are still some caveats.

The thrust for higher performance coupled with the predictability of branch outcomes suggest to use some prediction mechanism to know the branch outcome even before the branches are executed. Even after having a good predictor, in most of the cases, the penalty of branch misprediction is as bad as stalling the pipeline. This is the reason precisely why we need really smart branch predictors.

ii. BASIC BRANCH PREDICTION METHODS

A. Bi-Model Branch Predictors

- A bimodal branch predictor or Direct History Table (DHT) is one of the most elementary forms of branch predictions. A table of n-bit entries is indexed, where the least significant bits of the branch addresses are stored. It is dissimilar to caches in the sense that, bimodal predictor entries typically do not contain tags as part of them. Consequentially, a particular entry may be mapped to different branch instructions (branch interference or branch aliasing), and so accuracy decreases in these cases.

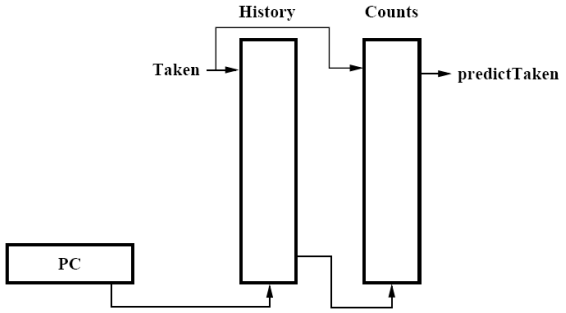


- As in the above figure, the branch address, stored in the PC is compared with the Direct History Table. Branch prediction bits predict the outcome, and when the actual outcome is known, the entry in the table that corresponds to it is updated.
- Measured hit rate of bimodal branch predictors with 4KB entries for SPEC95 benchmarks is around 90%.
- On the SPEC'89 benchmarks, very large bimodal predictors saturate at 93.5% correct, once every branch maps to a unique counter.

B. Local Branch Predictors

- A shortcoming of bimodal branch predictors is that they incorrectly predict the exit of every loop. For loops that usually have the same loop count each time (many other branches that display repetitive behaviour), we can improve on the bimodal system. In this predictor system, two tables are maintained. The first table is the local branch history table. It is indexed by the low-order bits of each branch instruction's address, and it records the taken/not-

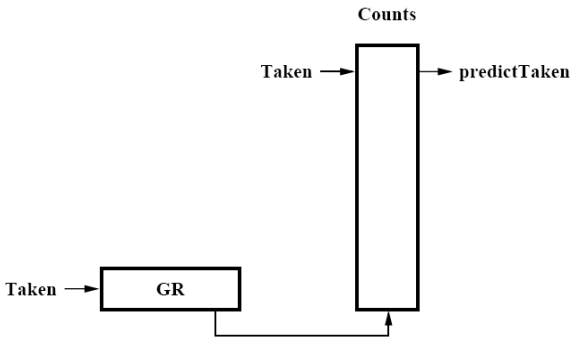
taken history of the n most recent executions of the branch. The other table is the pattern history table. Like the bimodal predictor, this table contains bimodal counters; however, its index is generated from the branch history in the first table. To predict a branch, the branch history is looked up, and that history is then used to look up a bimodal counter which makes a prediction. This correlation-based branch predictor suggested by Yeh and Patt's also referred to as Yeh's Algorithm and has been proven to be very successful.



- These provide a marginally better hit rate than bimodal predictors.

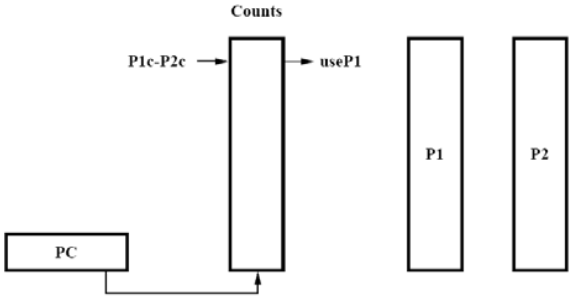
C. Global Branch Predictors

- The concept that global branch predictors are built on is that branch behaviour is often strongly dependant on the history of branches taken in the recent past. To keep track of the history of branches, one shift register can be updated with the recent history of every branch executed, this value can be used contextually for indexing into a table of bimodal counters. On the whole, this is an improvement on the bimodal scheme when considering large table sizes.



D. Combining Branch Predictors

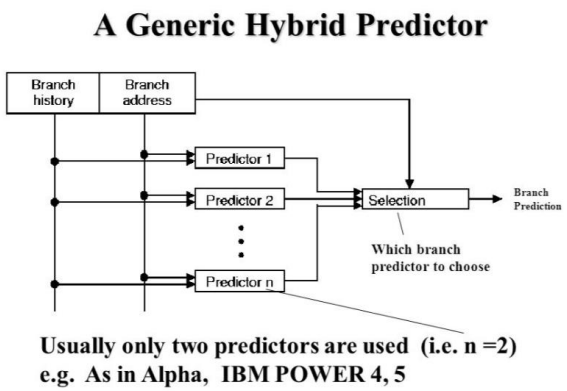
- Scott McFarling proposed combined branch prediction in his 1993 paper. It has about the accuracy of local prediction, speed-wise could almost compare to global prediction. Combined branch prediction parallel uses three predictor models: bimodal, gshare, and a bimodal-like predictor to pick one of the other two models to use on a branch-by-branch basis. The choice predictor is yet another 2-bit up/down saturating counter, in this case the MSB choosing the prediction to use. In this case the counter is updated whenever the bimodal and gshare predictions disagree, to favour whichever predictor was actually right.



iii. COMMERCIAL BRANCH PREDICTORS

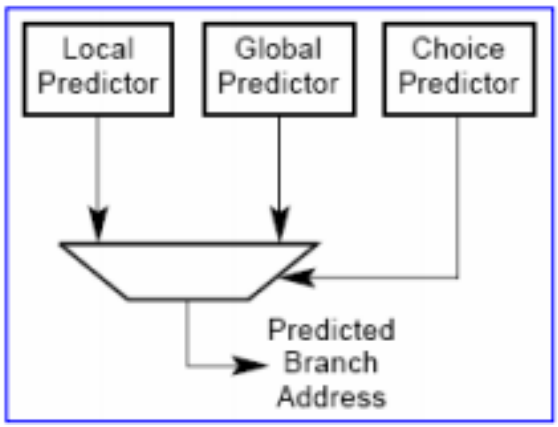
E. POWER4

- The POWER4 microprocessor's branch prediction unit is similar in design to the combined predictor proposed by Scott McFarling. The POWER4 branch prediction unit consists of three sets of branch history tables. The first one is called a Local Predictor, which has 16K entries and each entry is 1 bit. The second table is called Global Predictor, which has an 11-bit history register/vector. The content of the history vector is XOR'ed with branch addresses before indexing the history table. Other than these two tables, another table exists which is called the selector table. This has 16K 1-bit entries and keeps track of the better predictor.
- If the first branch encountered in a particular cycle is predicted as not taken and a second branch is found in the same cycle, the POWER4 processor predicts and acts on the second branch in the same cycle. In this case, the machine will register both branches as predicted, for subsequent resolution at branch execution, and will redirect the instruction fetching based on the second branch.
- Dynamic branch prediction can be overridden by hint bits in the branch instructions. This is useful in cases where knowledge at the application level exists that can result in better predictions than the execution-time hardware prediction methods. It is accomplished by setting two previously reserved bits in conditional branch instructions, one to indicate a software override and the other to predict the direction. When these two bits are zero, the hardware branch prediction previously described is used. Since only reserved bits are used for this purpose, 100 percent binary compatibility with earlier software is maintained. The POWER4 processor also has target address prediction logic for predicting the target of branch to link and branch to count instructions, which often have repeating and therefore predictable targets.



F. ALPHA 21264

- Similar to POWER4 Alpha 21264 branch predictor is also composed of three units – Local predictor, Global predictor, and Choice predictor. The Local predictor maintains the per-branch history and each entry is a 2-bit saturation counter. There are a total of 1K entries in local predictor. The Global predictor uses 12 most recent branches to predict the outcome of a branch. There are total 4K entries in global predictor and each entry is 2-bit wide.
- Choice predictor monitors the history of local and global predictors and chooses the best of two. It has 4K entry and each entry is 2-bit saturation counter.



- The Alpha 21264 branch predictor uses local history and global history to predict future branch directions since branches exhibit both local correlation and global correlation. The property of local correlation implies branch direction prediction on the basis of the branch's past behaviour. Local-history predictors are typically tables, indexed by the program counter (branch instruction address), that contain history information about many different branches. Different table entries correspond to different branch instructions. Different local-history formats can exploit different forms of local correlation. In some simple (single-level) predictors, the local-history table entries are saturating prediction counters (incremented on taken branches and decremented on not-taken branches), and the prediction is the counter's uppermost bit. In these predictors, a branch will be predicted taken if the branch is often taken.

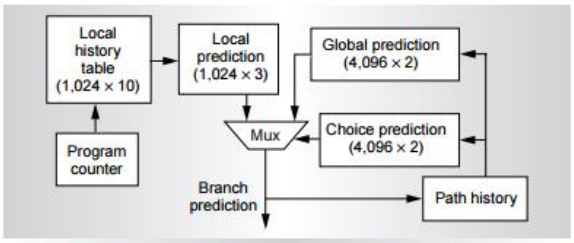


Figure 4. Block diagram of the 21264 tournament branch predictor. The local history prediction path is on the left; the global history prediction path and the chooser (choice prediction) are on the right.

G. Intel Branch Predictors

- The 386 and 486 microprocessors did not possess any hardware based dynamic branch prediction block, all branches were statically predicted as NOT TAKEN. The Pentium III processor has a two-level local history based branch predictor, where each entry is a 2-bit saturating (Lee-Smith) counter.
- The Pentium M processor combined three branch predictors- the Bimodal, Global and Loop Predictor. Loop predictor analyses the branches to see if they exhibit any loop behaviour.

iv. FACTORS INVOLVED IN THE EFFICIENCY OF BRANCH PREDICTION

Since Branch Prediction is a technique that is designed to reduce the waiting time caused by a branch, the efficiency of the branch is vitally important. Branch prediction is affected by a variety of factors and these factors are listed below. Since there are many types of branch prediction, this section will only deal with factors that affect all these types and not restrict itself to the discussion of the factors involved in a single type of branch prediction.

1) Interference

- Interference (also called branch aliasing) in branch prediction arises when a resource, most commonly a row in a table (branch addresses, bit modal counters) is being used by two different branches. For example, two different branch instructions in the instruction stream might be mapped to the same row of a Pattern History Table (PHT).
- There are commonly three types of interference- positive, negative and neutral. In the example above, if the branches are unrelated, the outcome of one branch is interfering with the prediction of another completely unrelated branch. This is negative interference and it is empirically shown that this is the most common type of interference. If the conflicting use of the resource does not affect either branch, it is neutral interference, and if the branches are similar enough that one branch instruction helps another, it is called positive interference.
- Negative interference invariably leads to wasted cycles and in some cases this causes so many missed cycles that it is equivalent to waiting for the branch to complete. As a result it is an important factor in ensuring the efficiency of branch prediction and minimising the interference, specifically the

negative interference is an important aspect in ensuring that the branch prediction is accurate and efficient.

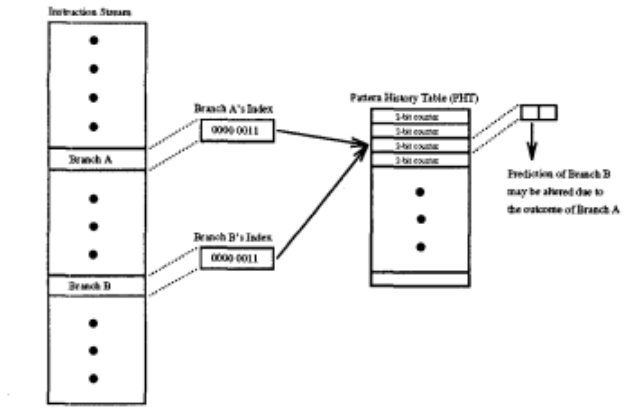


Figure 2: Interference in a two-level predictor.

2) There are three fundamental ways to reduce interference:

- Increasing predictor table size, causing conflicting branches to map to different table locations: This is the obvious solution. Here to avoid conflicts, we simply increase the number of entries to the table which would decrease the chances for conflicts.
- Selecting a table indexing scheme that best distributes history state among the available counters: Indexing needs to be done to decide where in the table a specific value is stored. Most commonly this is done with the help of a hashing function. If the hash function that is chosen is very inefficient and maps multiple values to the same table index, this will also increase the chances of negative interference. As a result, a hash function that is optimal for the given data set must be chosen.
- If branch address merged with global history register it reduces the misprediction rate further. Another variation of this, known as gshare, does the XOR of branch history and branch address and then indexes the history table. Some of the more efficient hash functions enable better use of the table as the history is evenly distributed in table.
- Separating different classes of branches so that they do not use the same prediction scheme, and thus cannot possibly interfere: This can be achieved by providing different classes of branches with a specific table each. By doing this negative interference will become impossible. This does not affect the possibility of there being either a neutral or a positive interference.

3) Size of History

- Generally speaking, branches show some sort of correlation with their own previous outcomes or with the outcome of other branches. Co-relation with their own outcome is known as local correlation whereas branches which are related with other branches globally are said to be globally correlated.
- Branch predictors which want to use this correlation (either local or global) must store some sort of a history table to make use of past outcomes. The main problem here arises when we wish to decide how much history should be stored. Making this decision is complicated and may require branch profiling.
- Given below is a graph that maps the rate of misprediction for 10 million instructions of differing amounts of history. (GHR or global history register denotes the size of the history stored)

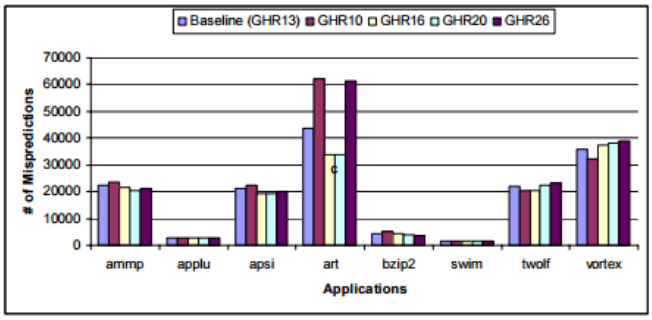


Fig 7: Misprediction for various history lengths

- While it may seem intuitive that increasing the size of the history stored will decrease misprediction, this is not always true, as seen from the above figure. It is possible that the reason for this is that a huge history size leads to connections based on outdated data and thus the created correlations are not valid.

1) Effectiveness of the use of the Pattern History Table (PHT)

- Pattern History Tables are used to track particular patterns and use the occurrence of these patterns to predict possible branches. The problem arises because not all patterns are used uniformly. If the table stores only a specific number of patterns, and if slots are allocated and used by patterns that occur very infrequently, this increases the possibility of interference and also wastes the increased efficiency that can be obtained by the Pattern History Table.
- The size of the GHR also has an effect on the utilisation of the PHT. For different GHR, some of the branches are mapped to completely different entries, which can affect the effectiveness of the PHT. This is shown in the graph below.

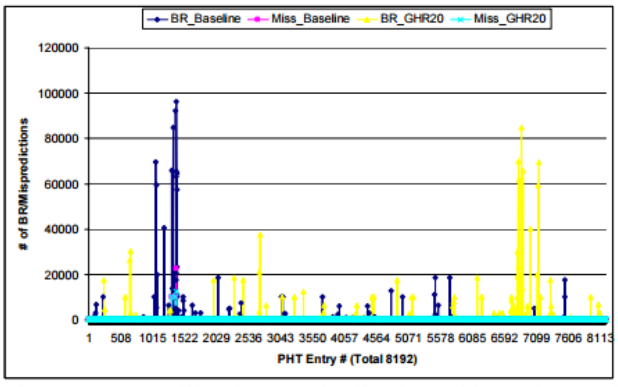


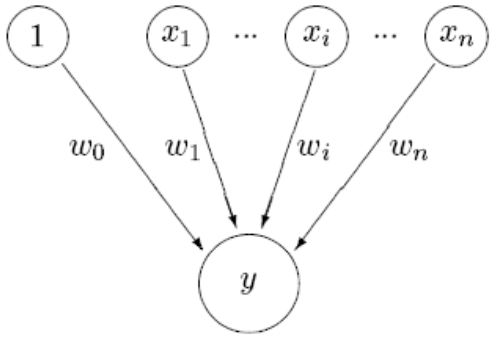
Fig 10: PHT Utilization – Distribution of branches and misprediction per entry for various GHR size

v. NEURAL BRANCH PREDICTION: A MACHINE LEARNING APPROACH

Modern computer architectures increasingly rely on speculation to boost instruction-level parallelism. As seen earlier, data that are likely to be read in the near future are speculatively pre-fetched, and predicted values are speculatively used before actual values are available. Accurate prediction mechanisms have been the driving force behind these techniques, so increasing the accuracy of predictors increases the performance benefit of speculation.
Machine learning techniques offer the possibility of further improving performance by increasing prediction accuracy. The conditional branch prediction is a Machine learning problem where machine learns to predict the outcome of conditional branches. Artificial neural network tries to model the neural networks in brain cells. It is very efficient to learn to recognize and classify patterns. Similarly, the branch predictor also uses the classification of branches and tries to keep the co-related branches together.

1) Perceptron Based Predictors

- A perceptron is a learning device that takes a set of input values and combines them with a set of weights (which are learned through training) to produce an output value. In our predictor, each weight represents the degree of correlation between the behaviour of a past branch and the behaviour of the branch being predicted. Positive weights represent positive correlation, and negative weights represent negative correlation. To make a prediction, each weight contributes in proportion to its magnitude in the following manner. If its corresponding branch was taken, we add the weight; otherwise we subtract the weight. If the resulting sum is positive, we predict taken; otherwise we predict not taken. To make this solution work, the branch history uses 1 to represent taken and −1 to represent not taken. The perceptron is trained by an algorithm that increments a weight when the branch outcome agrees with the weight's correlation and decrements the weight otherwise.
- The perceptron was introduced in 1962 as a way to study brain function. We consider the simplest of many types of perceptron's, a single-layer perceptron consisting of one artificial neuron connecting several input units by weighted edges to one output unit. A perceptron learns a target Boolean function t (x1, ..., xn) of n inputs. The inputs to a neuron are branch outcome histories (x1, …, xn) and the target function predicts whether a particular branch will be taken. Intuitively, a perceptron keeps track of positive and negative correlations between branch outcomes in the global history and the branch being predicted.



- The Figure above shows a graphical model of a perceptron. A perceptron is represented by a vector whose elements are the weights. For our purposes, the weights are signed integers. The output is the dot product of the weights vector, w0...n, and the input vector, x1...n (x0 is always set to 1, providing a "bias" input). The output y of a perceptron is computed as

$$y = w_0 + \sum_{i=1}^{n} x_i w_i.$$

- The inputs to our perceptron is bipolar; that is, each xi is either −1, meaning not taken or 1, meaning taken. A negative output is interpreted as predict not taken. A nonnegative output is interpreted as predict taken.
- The inputs to our perceptron is bipolar; that is, each xi is either −1, meaning not taken or 1, meaning taken. A negative output is interpreted as predict not taken. A nonnegative output is interpreted as predict taken.

2) Training Perceptrons

- Once the perceptron output y has been computed, the following algorithm is used to train the perceptron. Let t be −1 if the branch was not taken, or 1 if it was taken, and let θ be the threshold, a parameter to the training algorithm used to decide when enough training has been done.

```
if sign(y_out) ≠ t or |y_out| ≤ θ then
    for i := 0 to n do
        w_i := w_i + t x_i
    end for
end if
```

- Since t and xi are always either −1 or 1, this algorithm increments the ith weight when the branch outcome agrees with xi, and decrements the weight when it disagrees. Intuitively, when there is mostly agreement (i.e., positive correlation), the weight becomes large. When there is mostly disagreement (i.e., negative correlation), the weight becomes negative with large magnitude. In both cases, the weight has a large influence on the prediction. When there is weak correlation, the weight remains close to 0 and contributes little to the output of the perceptron.

3) Using Perceptron in Branch Predictors

- We can use a perceptron to learn correlations between particular branch outcomes in the global history and the

behaviour of the current branch. These correlations are represented by the weights. The larger the weight, the stronger the correlation, and the more that particular branch in the global history contributes to the prediction of the current branch. The input to the bias weight is always 1, so instead of learning a correlation with a previous branch outcome, the bias weight w0 learns the bias of the branch, independent of the history. The processor keeps a table of N perceptron in fast SRAM, similar to the table of two-bit counters in other branch prediction schemes. The number of perceptron N is dictated by the hardware budget and number of weights, which itself is determined by the amount of branch history we keep. Special circuitry computes the value of y and performs the training.

- The above figure represents the circuitry for perceptron branch predictor. When the processor encounters a branch in the fetch stage, the following steps are conceptually taken.

1. The branch address is hashed to produce an index i ∈ 0...N − 1 into the table of perceptron.
2. The ith perceptron is fetched from the table into a vector register P0...n of weights.
3. The value of y is computed as the dot product of P and the global history register.
4. The branch is predicted not taken when y is negative, or taken otherwise.
5. Once the actual outcome of the branch becomes known, the training algorithm uses this outcome and the value of y to update the weights in P.
6. P is written back to the ith entry in the table.

- It may appear that prediction is slow because many computations and SRAM transactions take place in Steps 1 through 5. However, it can be showed that a number of arithmetic and micro-architectural tricks enable a prediction in a single cycle, even for long history lengths.

1) Advantages and Disadvantages

- Neural prediction could be incorporated into future CPUs. Accuracy is very good however complexity is still a bottleneck. The main advantage of the neural predictor is its ability to exploit long histories while requiring only linear resource growth. Classical predictors require exponential resource growth.
- The main disadvantage of the perceptron predictor is its high latency. Even after taking advantage of high-speed arithmetic tricks, the computation latency is relatively high compared to the clock periods of even modern Microarchitectures.

CONCLUSION

There are two aspects of branch predictions. First is to determine the outcome (Taken or Not taken) of branch and if taken then the knowledge of target address. A static branch prediction scheme relies on the information incorporated in prior to runtime whereas the dynamic (adaptive) branch prediction logic tries to extract the information and predicts the behaviour of branches in run-time. Most of the adaptive predictors use two piece of information. The history of last k branches and their specific behaviour s is taken into account in order to implement the high performance branch prediction logic. The reason why dynamic branch prediction beats static is because the kind of data appears in run time is very much different from the sample (trace) data which is used for profiling.

Bi-model predictor is simplest kind of predictor which uses the branch address and n-bit saturating counter based pattern table to predict the outcome. If miss predicted the pattern table entry is corrected by incorporating the actual outcome. In such predictors, the prediction accuracy is function of size of pattern table and the maximum achievable hit rate is 93 – 94%. Local branch predictors add one more level of hardware which is known as History Table in addition to n-bit saturating counter based pattern table.

The perceptron is a basic prediction mechanism with neural networks. The key advantage of perceptrons is their ability to use long history lengths without requiring exponential resources. These long history lengths lead to extremely high accuracy. In particular, for the SPEC 2000 integer benchmarks, our new global/local perceptron has 36% fewer mispredictions than a McFarling-style hybrid predictor, which is the most accurate known predictor that has been implemented in silicon. Our global/local perceptron is also more accurate than the multicomponent predictor of Evers et al., which was previously the most accurate known predictor in the literature.

REFERENCES

- http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/
- S. McFarling Combining Branch Predictors Digital Western Research Lab (WRL) Technical Report, TN-36, 1993
- http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/
- Predictors, Branch. "WRL Technical Note TN-36." (1993).
- http://www.redbooks.ibm.com/redbooks/pdfs/sg247041.pdf
- http://www.ece.rochester.edu/~parihar/pres/Pres_BranchPrediction-Survey.pdf
- Kessler, Richard E. "The alpha 21264 microprocessor." Micro, IEEE 19.2 (1999): 24-36.
- Raj Parihar: Branch Prediction and Techniques
- (http://www.ece.rochester.edu/~parihar/pres/Paper_BrPrediction.pdf)
- Eric Spangle, Robert S Chappell, Mitch Alsup, Yale N Patt: The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference.
- Maria-Dana Tarlescu, Kevin B. Theobald, and Guang R. Gao. Elastic History Buffer: A Low-Cost Method to Improve Branch Prediction Accuracy. ICCD, October 1996.
- https://www.cs.utexas.edu/~lin/papers/tocs02.pdf