

The Frankenpredictor

Stitching Together Nasty Bits of Other Branch Predictors

Gabriel H. Loh

Georgia Institute of Technology

College of Computing

loh@cc.gatech.edu

Abstract

Our branch predictor proposed for the Championship Branch Prediction (CBP) contest is a hybrid/mutation of many different ideas previously proposed in the literature, combined with a few new techniques. The mix of traces requires the predictor to handle both capacity (large tables) and deep correlation extraction (long branch histories), both made difficult by the relatively small hardware budget. We use a gskewed-agree predictor to provide capacity, combined with a path-based neural predictor that simultaneously acts as both a long-history predictor and a fusion/meta-predictor.

1 Introduction

The need for accurate branch prediction algorithms for large-window, deeply-pipelined microprocessors is well known. While much research has gone into branch prediction over the past two decades, many algorithms have been difficult to compare due to differing benchmarks, architectures and simulation infrastructures used by different research groups. For the Championship Branch Prediction (CBP) contest [1], we propose a *Frankenpredictor* that scavenges parts from many other previously proposed branch predictors, combines new ideas, and pieces everything together into a predictor that results in very high prediction accuracy.

The traces provided for the contest are divided into four groups, each with different behaviors. In particular, the integer and multimedia traces appear to benefit from long branch history correlation, while the server traces have a very large branch footprint and benefit from less complex but more spacious tables. We viewed reconciling these conflicting objectives as the main challenge in designing a branch predictor that would perform well on all traces. To deal with these multiple objectives, we have taken parts of many different predictors to form a “Frankenpredictor.”

2 The Frankenpredictor

The high-level overview of the Frankenpredictor is a gskewed global history predictor [6] combined with a path-based neural predictor [3]. The gskewed component provides capacity for traces with large working sets. The neural predictor provides the ability to mine long-history correlations, and it also acts as the hybridization agent by using the gskewed predictions as bits in its input vector. Figure 1

shows the tables and logic of the Frankenpredictor. Note that the logic used to generate the individual indices for all of the perceptron weight lookups is not illustrated (this is represented by the example locations of weights).

2.1 Gskew-Agree Predictor

The primary motivation for the gskewed component is to provide capacity for the large-footprint server traces. The gskewed predictor already provides substantial anti-aliasing capabilities [6]. To augment this, our gskewed predictor also implements an agree predictor [10]. If the prediction is true, then the gskew-agree predictor agrees with the static prediction determined by the branch target direction (BTFNT). Skewing provides interference avoidance benefits, while agree-prediction provides interference tolerance benefits by converting destructive interference to neutral interference.

Similar to the 2bcgskew predictor, every two counters in the PHTs share a hysteresis bit [8]. We apply this technique for all three PHTs, as opposed to only a few as done for the original 2bcgskew predictor. Also similar to the enhanced-gskewed and the 2bcgskew, the first PHT uses only the branch address for indexing and therefore implements a simple bimodal predictor (albeit a bimodal-agree predictor) [9]. We also use a partial-update policy similar to the 2bcgskew update algorithm.

2.2 Path-Based Neural Predictor

Our neural component is primarily based on the path-based neural predictor [3]. While the central structure is very similar to the path-based neural predictor, we introduce many modifications that are new to our Frankenpredictor.

2.2.1 Fusing the Gskew-Agree Prediction(s)

Typically, the input vector of a neural branch predictor consists of k branch outcomes (x_1, x_2, \dots, x_k) in the global branch history register, plus a hard-wired 1 (x_0) for the bias. We expand the perceptron input vector to incorporate the prediction information of the gskew-agree predictor. In particular, we introduce four new inputs x_{g0}, x_{g1}, x_{g2} and x_{gM} , where $x_{gi, i \in \{0,1,2\}}$ is the direction prediction determined by PHT i of the gskew predictor and the static BTFNT prediction. The input x_{gM} is the prediction determined by the branch target direction and the overall majority vote of agreement. In this fashion, the neural component

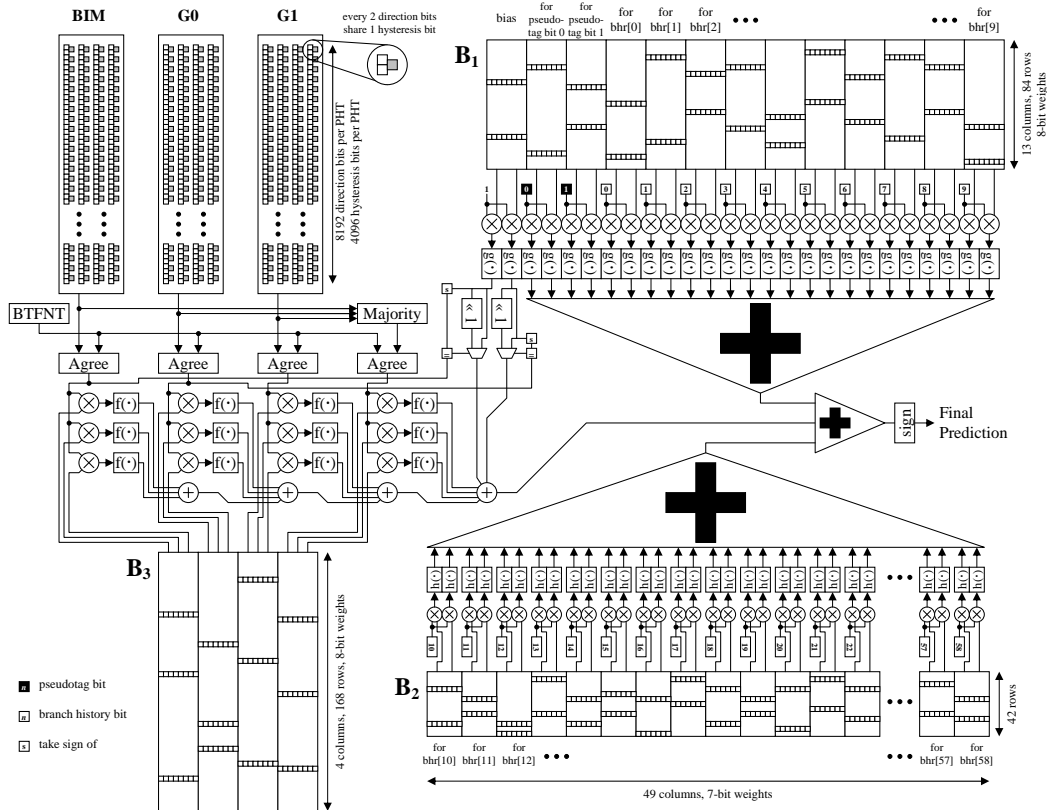


Figure 1. The Frankenpredictor tables and logic. Index generation logic, branch history register and path history register are not shown.

acts as both a long-history predictor and a prediction fusion mechanism for the gskew-agree inputs [5].

2.2.2 Non-Uniform Weight Allocation

Depending on the purpose of the weight, we provide a different number of entries in the table of weights. The table of weights consists of three banks. The first bank B_1 corresponds to the bias, pseudotag bits [7], and the most recent history bits. The second bank B_2 provides the weights for older history bits. The third bank B_3 provides the weights to fuse the gskew-agree predictions. Each bank is sized differently and even the widths of the counters vary. The exact sizes of the tables and counters are shown in Figure 1.

2.2.3 Non-Linear Learning Functions

Research in machine learning has shown that linear learning functions do not converge as fast as other functions for perceptron-like algorithms [4]. To approximate the effects of non-linear learning curves, we still use integer weights with additive updates, but we interpret the value of a weight differently by using a weight translation function. These functions are denoted as $f(\cdot)$, $g(\cdot)$ and $h(\cdot)$ in Figure 1.

The translation functions used for $f(\cdot)$ and $g(\cdot)$ are the same. Looking at only positive inputs to this function, illustrated in Figure 2(a), the domain is divided into four

quadrants, each of which is linear and the entire function is piecewise continuous. The binary-encoded value stored by a weight is translated through this function before being added to the perceptron’s sum. For example, a counter value of 23 only contributes a value of 11 to the final sum. The function is odd-symmetric for negative-valued inputs. The translation function $h(\cdot)$ used for the older global branch history has a sharper concavity and is shown in Figure 2(b). Note that the domain and range for $h(\cdot)$ are smaller because the counters used in table B_2 are narrower.

There are two benefits of using a learning curve that is concave instead of linear. The early part of the curve provides a slow-start effect, which prevents transient or coincidental correlations from having too great of an impact on the final prediction. The latter part of the curve has a larger slope which allows the predictor to more quickly “unlearn” correlations when program behavior changes. The reasoning for a deeper curve for $h(\cdot)$ is that older history bits are more likely to be very important or to not matter at all.

2.2.4 Skewed Redundant Indexing

The tables of weights for neural predictors typically do not have very many entries due to the large number of weights per entry. This results in a large amount of interference. To help combat the aliasing problem, we use multiple (redun-

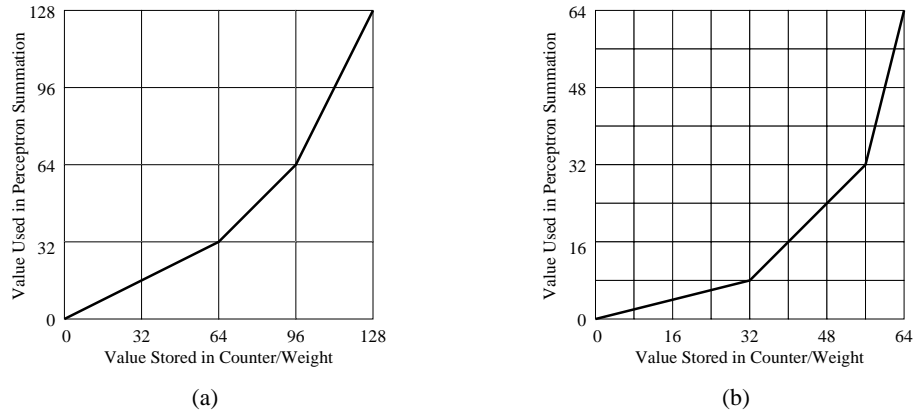


Figure 2. Perceptron weight translation functions for (a) B_1 and B_3 , and (b) for B_2 .

dant) weights for each bit of the information vector. The idea is that the probability of strong destructive aliasing in multiple rows is less likely than that for a single row. Banks B_1 and B_2 provide two weights for each bit of the input vector, and bank B_3 (for the gskew-agree inputs) provides three weights. Index computations are discussed in Section 2.5.

2.2.5 Synergistic Reinforcement

Since the first bias weight of the perceptron is indexed by the branch address only, this weight just provides a bimodal prediction with a large amount of hysteresis. The first bank of the gskew-agree predictor is also a bimodal predictor. When both the gskew-agree bimodal table and the bias weight agree in the direction of the branch, we increase the contribution of the bias weight. The rationale behind this “synergistic reinforcement” is that if both the bimodal prediction and the bias weight agree, then it is likely that the bias weight more accurately reflects the bias of the branch. A similar reinforcement is performed when the second (redundant) bias weight and the second PHT (G0) agree.

2.3 Branch Address Modifications

We found that the exclusive-OR of the least significant bits of the program counter with the most recent branch history bits slightly increases the amount of aliasing. We permuted the bottom eight bits of the PC by performing a right barrel shift by two positions which helped to alleviate some of the aliasing. We found that the branch target direction provided useful information for indexing, and so our effective PC is equal to our permuted PC concatenated with the branch target direction. Note that for the purposes of indexing, we actually compute a “pseudo-target” direction based on comparing the target address *with the permuted PC* (for the gskew-agree, we use the real target direction). In the Pentium 4 on 90nm processor, the static prediction is not a strict BTFNT prediction, but it varies based on the

magnitude of the distance to the target as well [2]. We hypothesize that by comparing the target against the permuted PC, we capture some of this effect.

2.4 Unconditional Branches

Because opcode information is available to the predictor, unconditional branches are always correctly predicted as taken. For the update phase of the algorithm, unconditional branches do not cause updates on any of the gskew-agree counters or the perceptron weights. Instead, we make modifications to the branch history register so that later branches are “aware” that an unconditional branch occurred recently. In particular, on a subroutine call we shift in eight zeros into the history register, on a subroutine return we shift in eight ones into the history register, and on all other unconditional branches such as indirect jumps we shift in eight alternating zeros and ones (0x55). After shifting in the 8-bit pattern, we also hash in the lowest eight bits of the unconditional branch’s PC (the original PC without permutation or target direction).

2.5 Path History and Indexing B_1 , B_2 and B_3

Whereas the PC we use for indexing contains the pseudo-target bit, at update time the actual branch direction is known. For updating the path history, we use the original PC concatenated with the actual branch direction and the previous branch direction. This provides slightly better indexing into the perceptron tables because the recent branch outcomes provide more useful information than the upper bits of the PC.

When indexing the history portion of the perceptron (for B_1 and B_2), we need two indices for the original index and the redundant index. The first index uses the three least significant bits of our current PC (which includes the pseudo-target direction) and the least significant bits of the path history. The redundant index is similar, but it uses the three

Bits	Description/Justification
36864	Three gskew PHT banks, each with 8192 entries, at 1.5 bits per entry (due to shared hysteresis bits)
8736	B_1 has 84 entries, with 13 weights per entry (10 history + 1 bias + 2 pseudotag), and 8-bit weights
14406	B_2 has 42 entries, with 49 weights per entry (49 older bits of history), and 7-bit weights
5376	B_3 has 168 entries, with 4 weights per entry (3 gskew-agree predictions + majority), and 8-bit weights
59	59-deep branch history register
197	Path history buffer: The path-history indices use three bits from the current PC, and so the number of address bits stored per path address can be reduced. 61 entries (59 history + 2 pseudotag bits): the first two entries require at most 8-3=5 bits (to index into the 168 entries of B_3 , only using the current PC and the two most recent branch addresses); the next ten entries require at most 7-3=4 bits (to index into the 84 entries of B_2 , which has 13 weights per entry, but the bias uses the current PC, and the pseudotag bits reuse the addresses already stored for table B_3 , which leaves only ten addresses); the last 49 entries only require 6-3=3 bits each (to index into the 42 entries of B_2).
65638	Total (with 154 bits to spare)

Table 1. Tabulation of all of the state used in the Frankenpredictor.

most recent branch outcomes from the BHR instead of the three lowest PC bits.

For the gskew-agree perceptron weights (B_3), the predictor must generate three indices per input. For these indices, we use a variety of hashes that incorporate different selections from the current PC, the two most recent previous PCs, and the branch history register. The indices are computed by choosing two or three of these values, possibly shifting some of them, and taking the bitwise exclusive-OR.

3 Implementation

The details of the implementation of the Frankenpredictor algorithm are documented in the submitted code. Due to the constraint on size only, we did not worry about the access latency, port requirements of the structures, and the complexity of the logic used. In particular, redundant indexing greatly increases the number of inputs into the perceptron adder tree (although the Multiply-Add Contribution format of the MAC-RHSP predictor could largely offset this effect) [7]. Pipelined indexing schemes such as that used in the path-based neural predictor would have to be adapted to address the port requirements of the perceptron tables [3]. Non-power-of-two sized tables would not be practical, but then in a real design, there likely would not be a hard-limit on the amount of state.

The total state budget is 64K bits + 256 bits = 65792 bits. We use 59 bits of history, with the 10 most recent branches considered as “young” and tracked by B_1 , and all older branches tracked by B_2 . The state accounting is shown in Table 1. The path history buffer contains some bits from the branch history. It is possible to further reduce the predictor’s total state by taking those bits directly from the

branch history register instead of explicitly copying them into the path history register.

4 Summary

While the Frankenpredictor as described has some serious implementation challenges, those shortcomings are all within what is permitted for the contest, and with some clever engineering may be overcome. The (omitted) results show that even beyond a global-local path-based neural predictor, better prediction accuracy is still achievable.

Acknowledgments

Equipment and funding support provided by Intel Corporation.

References

- [1] The 1st JILP Championship Branch Prediction Competition (CBP-1). <http://www.jilp.org/cbp>.
- [2] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, B. Toll, and K. S. Venkatraman. The Microarchitecture of the Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), February 2004.
- [3] D. A. Jiménez. Fast Path-Based Neural Branch Prediction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 243–252, San Diego, CA, USA, December 2003.
- [4] N. Littlestone and M. K. Warmuth. The Weighted Majority Algorithm. *Information and Computation*, 108:212–261, 1994.
- [5] G. H. Loh and D. S. Henry. Predicting Conditional Branches with Fusion-Based Hybrid Predictors. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, USA, September 2002.
- [6] P. Michaud, A. Seznec, and R. Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, Boulder, CO, USA, June 1997.
- [7] A. Seznec. Revisiting the Perceptron Predictor. PI 1620, Institut de Recherche en Informatique et Systèmes Aléatoires, May 2004.
- [8] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, Anchorage, AK, USA, May 2002.
- [9] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, Minneapolis, MN, USA, May 1981.
- [10] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 284–291, Boulder, CO, USA, June 1997.