Journal of Instruction-Level Parallelism 7(2005) Submitted 01/05; published 04/05

Deconstructing the Frankenpredictor for Implementable Branch Predictors

Gabriel H. Loh LOH@CC.GATECH.EDU
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332-0280

Abstract

The Frankenpredictor entry for the Championship Branch Prediction contest proposed several new optimizations for branch predictors. The Frankenpredictor also assimilated many previously proposed techniques. The rules of the contest were such that implementation concerns were largely ignored. In this context, many of the proposed optimizations may not actually be feasible in a realizable predictor. In this paper, we revisit some of the Frankenpredictor optimizations and attempt to apply them to conventional predictor organizations that are more typical of what is described in the literature. In particular, reinterpreting perceptron weights with non-linear translation functions and using different sized tables with different widths of counters shows promise and is practical. The Frankenpredictor's branch history register update rules for unconditional branches also provide improvements in prediction accuracy for gshare and path-based neural predictors with very little implementation overhead.

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. The Championship Branch Prediction (CBP) contest has provided a common ground to compare branch prediction algorithms.

The rules of the CBP competition limit the state storage requirements of the branch predictors, but largely ignore any other implementation concerns such as latency, ease of pipelining, logic complexity, impact on overall performance, and power. As a result, many of the techniques incorporated into the final predictors may not be practical in a real predictor. This paper considers the *Frankenpredictor* entry which contains many different ideas of varying impact [6]. We study the application of the more practical techniques in isolation on conventional implementable branch predictors. This allows us to quantify the *realizable* benefits of the proposed techniques. The more promising candidates could potentially be immediately implemented in the next generation of processors. We also present the overall prediction performance of the complete Frankenpredictor.

2. Experimental Methodology

We use the same methodology for all results in this paper. Specifically, we used the branch prediction framework distributed for the Championship Branch Prediction (CBP) contest with the twenty distributed traces. We report the performance of branch prediction algorithms in thousands of mispredictions per instruction (Misp/Kinst), averaged (arithmetic mean) over the twenty traces. Whenever possible, branch predictor configurations have been chosen to be close to 8KB.

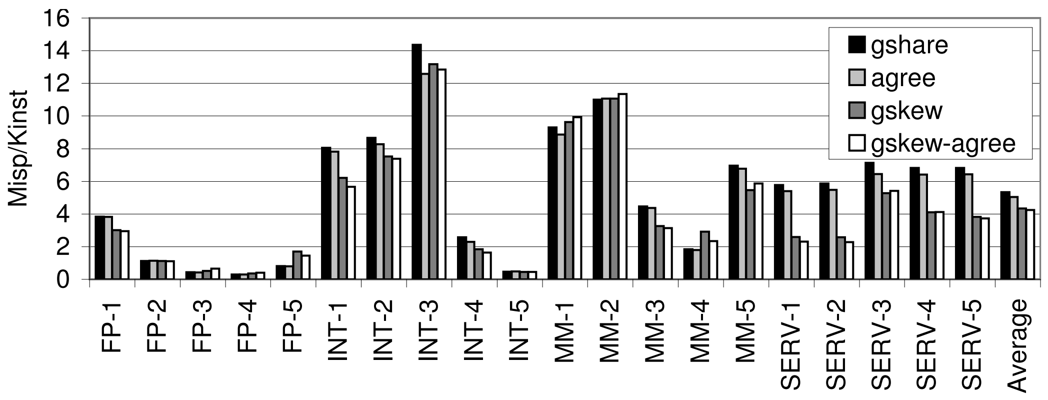

Figure 1: Prediction accuracy impact of interference mitigation techniques.

3. Combining Interference Avoidance and Interference Tolerance

In this section, we consider the parts of the Frankenpredictor that are designed to handle interference due to a large number of branches mapping to a fixed number of pattern history table (PHT) entries. In particular, we study gskew, agreement and the combination of the techniques.

3.1 Gskew-Agree

Many branch prediction algorithms employ simple tables of counters, typically indexed by a combination of a branch's address (program counter or PC) and the branch outcome history (the last n taken/not-taken predictions). For most applications, the number of unique program counter $\times$ branch history pairs far exceeds the number of entries in the PHTs. As a result, unrelated branches may collide in the table which results in *negative* or *destructive interference*.

The Frankenpredictor uses a *gskew-agree* branch predictor component that combines a gskew predictor with an agreement scheme. The gskew predictor avoids interference by mapping potential conflicting branches to different entries across three different tables [8]. This provides three different predictions, from which a final prediction is made by taking the majority vote. Interference in one table is not a problem because the other two tables do not suffer from the same interference. The agreement approach uses a default prediction or *bias* for each branch. The pattern history table (PHT) entries then predict whether the branch outcome *agrees* with the bias or not [11]. Two branches that map to the same PHT entry with different outcomes would normally cause destructive interference. If the biases are set correctly, the two branches would still map to the same PHT entry in an agree predictor, but the entry would correctly predict "agree with bias" for both branches. In this fashion, an agree predictor can tolerate interference by converting destructive interference into neutral interference.

Figure 1 shows the performance of the different interference-mitigating techniques. The agreement predictors are made with respect to a static prediction of backwards-taken/forward-not-taken (BTFNT) which means the default prediction is taken if the branch target address is lower (backward) than the current program counter.

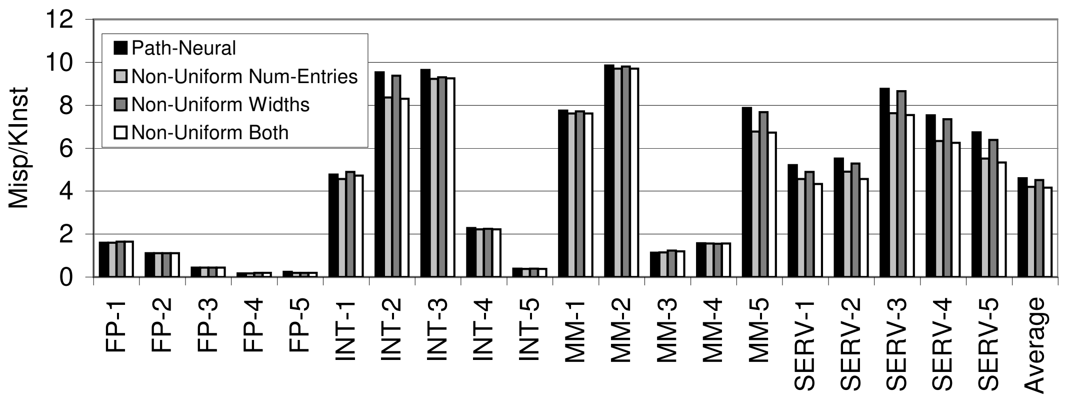

Figure 2: Path-based neural predictor with non-uniform allocation of resources.

The impact of the agreement mechanism is less than the gskew technique. For a few applications the agree mechanism actually performs better than gskew, and the average reduction in mispredictions is 5.3%. For the majority of the traces, gskew is a far superior technique for dealing with PHT interference, resulting in a 18.8% reduction in mispredictions. The combination of both gskew and agree has a slightly positive effect on average (20.2% reduction), although there are some cases where one of the individual techniques outperforms the combination.

3.2 Default Branch Bias

The results from Figure 1 show that augmenting a gskew predictor with the agreement technique can provide a slight improvement in prediction rates. While the absolute improvement is not large, the agree technique is still very attractive because it does not require the branch predictor to store any additional state. The previous results use a BTFNT static branch prediction as the basis for agreement. Unfortunately, the target direction may not be known until after the decode stage of the pipeline. This forces the agreement technique to either use some other information that is readily available at prediction time, or the technique can only be applied to second-level *overriding* branch predictors that occur in the decode stage [4]. We re-simulated the gskew-agree predictor using different bits from the branch's program counter to serve as the static prediction, but this approach does not improve the prediction rates over a conventional non-agreeing gskew predictor.

4. Modifications to the Basic Perceptron

The Frankenpredictor incorporates a neural prediction component based off of the path-based neural predictor [2]. The neural predictor uses a large table of weights, where the rows of the table correspond to different branch addresses (PCs), and the columns correspond to different past branch outcomes (bits in the branch history register). This section explores the impact of changing the allocation of the total number of weights, the size of individual weights, and the interpretation of the value of the weights.

4.1 Non-Uniform Resource Allocation

A conventional neural prediction weight table provides the same number of weights for all branch history bits considered. Typically, the most recent outcomes in the branch history register have the greatest influence of the outcome of the current branch prediction. The inverse of this statement implies that the older outcomes have less influence. The Frankenpredictor exploits this imbalance in the importance of branch history bits by allocating more weights for recent history and fewer weights for older history. For example, a 128-entry, 63-bit history neural table (8KB total) can be reallocated such that the weights that correspond to the most recent history (15 bits + bias) receive twice as many (256) entries, and the older 48 history bits receive only 85 entries (still 8KB).

The difference in the importance of older history bits and more recent history bits can be further exploited. Since the older history bits tend to be less reliable, it makes sense to limit the influence these weights have on the final outcome. To do this, the Frankenpredictor reduces the width of the weights that correspond to older history. Reducing the width of the weights reduces the number of bits required to implement the old-history table of weights. These bits can then be used to further increase the number of weights used for the recent-history weights. Finally, both of these techniques can be combined together to further reduce the amount of state spent on older branch history bits..

Figure 2 shows the prediction rates for the path-based neural predictor and variants that use non-uniform resource allocation. Similar to the impact of the gskew and agree techniques discussed in Section 3, the non-uniform allocation of the number and widths of weights has one dominant technique and one minor technique. The non-uniform allocation of the number of weights has the biggest impact, reducing the average number of mispredictions by 8.8%. The non-uniform weight widths only reduce mispredictions by 2.0%. Using both a non-uniform number of weights and weight widths, we can further increase the number of weights allocated to the recent branch history outcomes. This results in an overall misprediction reduction of 9.6%.

4.2 Non-Linear Learning Functions

The perceptron learning algorithm uses a linear learning function. That is, the perceptron algorithm update rule always increments or decrements its weights by a constant amount. Research in the machine learning field has shown that non-linear learning functions can converge on the critical feature set (the set of strongly correlated branch history bits) faster than linear learning functions. For n features/history bits, a perceptron-based algorithm will make $O(n)$ mispredictions to complete its training. A technique such as the Weighted Majority algorithm which uses a multiplicative update rule can converge with only $O(\log n)$ mispredictions.

To simulate the non-linear update of a perceptron's weights, we use a weight translation function to reinterpret the value of a weight. For example, a conventional perceptron with a weight of 18 contributes exactly 18 to the final summation. Considering the weight translation function illustrated in Figure 3(a), the same weight value of 18 would only contribute 9 to the final summation. The function shown in Figure 3(a) has a slow-start effect where branches with low amounts of correlation contribute less to the overall summation. The extreme ranges of the function have much steeper slopes, which provide a quick back-off effect that is beneficial to quickly unlearn correlations when those correlations no longer hold. Figure 3(b) shows another weight translation function that has a

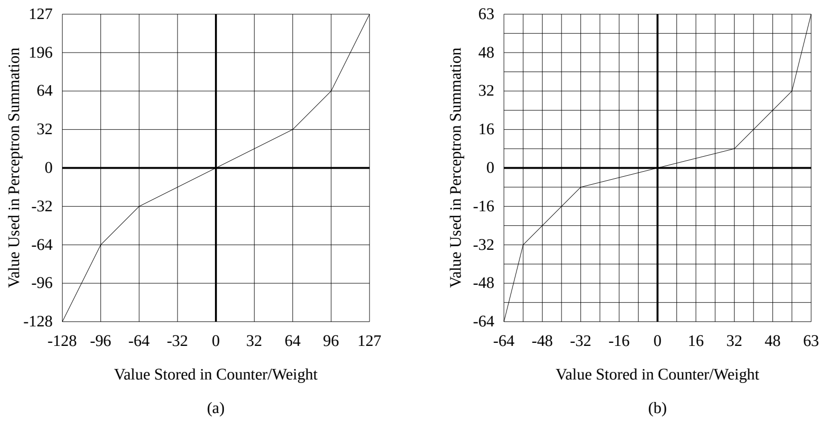

Figure 3: Two non-linear weight translation functions.

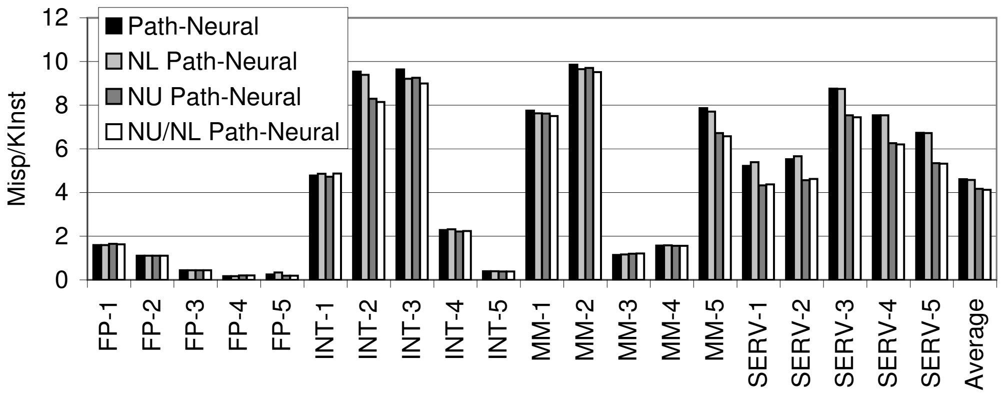

Figure 4: Accuracy impact of non-linear (NL) learning functions on conventional and non-uniform (NU) resource path-based neural predictors.

greater concavity. The Frankenpredictor uses this "deeper" translation function for interpreting old history weights.

We simulated the impact of the non-linear learning function shown in Figure 3(a) on the path-based neural predictor. We considered both a conventional path-based neural predictor as well as one that uses non-uniform resource allocation. The benefit of non-linear learning functions is dependent on the resource allocation, as shown in Figure 4. For both predictors, the non-linear translation functions provide an average misprediction decrease of about 1%. While this is not a very large difference, non-linear learning functions can be easily incorporated into existing neural-predictor design with no additional state. The use of a piecewise-linear translation function with convenient slopes (the function shown in Figure 3(a) has slopes of 1, 2 and $\frac{1}{2}$ which can be easily implemented with shifts) can result in a relatively simple hardware implementation of this optimization.

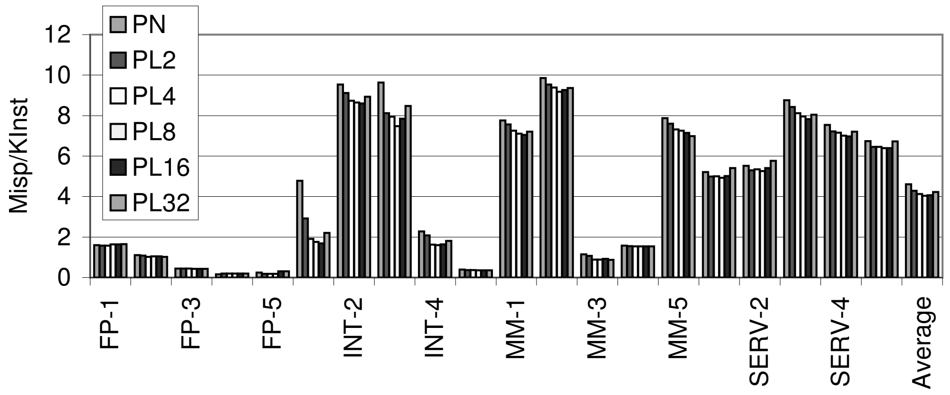

Figure 5: Accuracy impact of piecewise linear branch prediction.

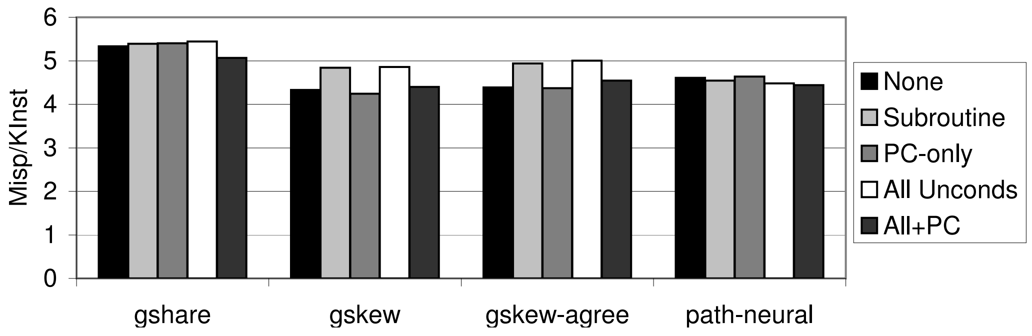

Figure 6: The impact of different branch history register update policies on unconditional branches.

5. Indices and Hashing

5.1 Piecewise Linear Prediction

Piecewise Linear (PL) perceptron branch predictors exploit the different paths that lead to the same branch to predict otherwise linearly inseparable branches [3]. Like the PL perceptron, the Franken-predictor incorporates a few bits of the current program counter into the path history. Figure 5 shows the effects of varying the number of bits from the current branch address that are incorporated into the branch path addresses. PN is the original path-based neural predictor, and PLn is a piecewise linear version that uses $\log_2 n$ PC bits to distinguish between n distinct paths. While this technique is effective at reducing the misprediction rate, requiring the current branch address for each weight makes the overall predictor very difficult to pipeline and hence compromises the implementability of the path-based neural predictor.

5.2 Unconditional Branches

Typically, unconditional branches are less interesting for branch predictors because their outcomes are always taken. The Frankenpredictor does not modify any of the counter/weight state on an unconditional branch. Instead, the predictor makes modifications to the branch history register so that later branches are aware that an unconditional branch had recently occurred. In particular, on a subroutine call the Frankenpredictor shifts in eight zeros into the history register, on a subroutine return the predictor shifts in eight ones into the history register, and on all other unconditional branches such as indirect jumps it shifts in eight alternating zeros and ones (0x55). After shifting in the 8-bit pattern, the predictor also hashes in the lowest eight bits of the unconditional branch's PC.

The accuracy impact of the different unconditional branch updates of the branch history register (BHR) varies depending on the predictor and the combination of updates. For example, Figure 6 shows that for gshare, any of the individual BHR updates results in an *increase* in mispredictions, but the simultaneous application of all BHR updates provides a 4.9% reduction in mispredictions. The path-based neural predictor behaves similarly with a 3.6% prediction improvement when all updates are applied. The gskew and gskew-agree predictors are less able to make use of the unconditional branch updates. The advantage of the unconditional branch update scheme is that it does not require storing additional state. The disadvantage is that it requires knowledge of whether the branch is unconditional or not, which may not be available until the decode stage if extra pre-decode bits are not available in the branch target buffer (BTB).

6. The Frankenpredictor

6.1 Overall Hardware Organization

The high-level overview of the Frankenpredictor is a gskewed global history predictor [8] combined with a path-based neural predictor [2]. At a high level, the organization of the Frankenpredictor is very similar to the Desmet, Vandierendonck, De Bosschere predictor [1]. 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 7 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). The three PHTs that comprise the gskew-agree component use shared hysteresis bits as described by Seznec et al. [10]. The perceptron weights are partitioned into three distinct tables with different numbers of rows (non-uniform allocation) and different weight widths. Similarly, the oldest weights are interpreted with the translation function depicted in Figure 3(b) and the other weights use the function from Figure 3(a).

The Frankenpredictor incorporates the other optimizations discussed earlier in this paper such as piecewise linear prediction and updating the branch history register on unconditional branches. The Frankenpredictor also uses *Skewed Redundant Indexing* for the neural part of the predictor. For each input bit in the neural predictor, we compute multiple indices into the perceptron table to provide multiple weights for the input. This reduces the effects of negative interference because even if two branches collide in one weight entry, it is unlikely that they will also collide in the

Configuration	Average Misp/KInst	Relative Increase
No BHR Update on Unconditional Branch	3.404	+10.30%
Linear Learning Only	3.198	+3.59%
Fuse Majority Only	3.196	+3.56%
No Skewed Redundant Indexing	3.160	+2.38%
No Agree (plain gskew)	3.132	+1.46%
No Pseudotag Bits	3.119	+1.04%
Pure Path History	3.103	+0.55%
No PC Bit-Shuffling	3.101	+0.48%
No Pseudo-target	3.091	+0.13%
No Synergistic Reinforcement	3.088	+0.05%
Base Frankenpredictor	3.086	—

Table 1: Misprediction rate impact due to omitting optimizations.

redundant weight. Redundant weights are costly in terms of hardware complexity because it requires additional read/write ports for the weight table SRAM. The neural component fuses the predictions made by the gskew-agree predictor (one each for the three PHTs and one for the majority vote) by treating these values as additional bits in the perceptron input vector [7]. The perceptron input vector includes two bits from the program counter, also known as *pseudotag* inputs [9].

The Frankenpredictor also incorporates several minor "bit-twiddling" optimizations that impact the indexing of various tables. We permute the eight least significant bits of the program counter by performing a right barrel shift by two positions. When computing the direction (backward or forward) of the branch target address for the purposes of determining the default BTFNT prediction, we use the permuted program counter to compute a *pseudo-target*. The last optimization increases the bias weight of the perceptron when the sign of the weight agrees with the prediction made by the first gskew-agree PHT. The rationale behind this idea of *synergistic reinforcement* is that if both the PHT prediction and the bias weight agree, then it is likely that the bias weight more accurately reflects the true bias of the branch.

6.2 Accuracy

Figure 8 shows the per-trace misprediction rates for an optimized gshare, a global-local perceptron [5] that uses both global and local branch history, a global-local path-based neural predictor, and the Frankenpredictor. For all but two traces, the Frankenpredictor mispredicts the least out of the four predictors.

In the previous sections we applied specific optimizations to conventional predictors, whereas we now show the results for a complementary set of experiments where we remove optimizations from the Frankenpredictor. Table 1 shows the average misprediction rates for the Frankenpredictor and configurations where certain Frankenpredictor features have been disabled. The table also includes the relative increase in the average misprediction rate for disabling each feature.

The ten optimizations listed in Table 1 have varying impact on the average misprediction rates. In particular, it is very important to update the branch history register on unconditional branches. Not doing so causes a 10% increase in the misprediction rate. Similarly, omitting non-linear learning

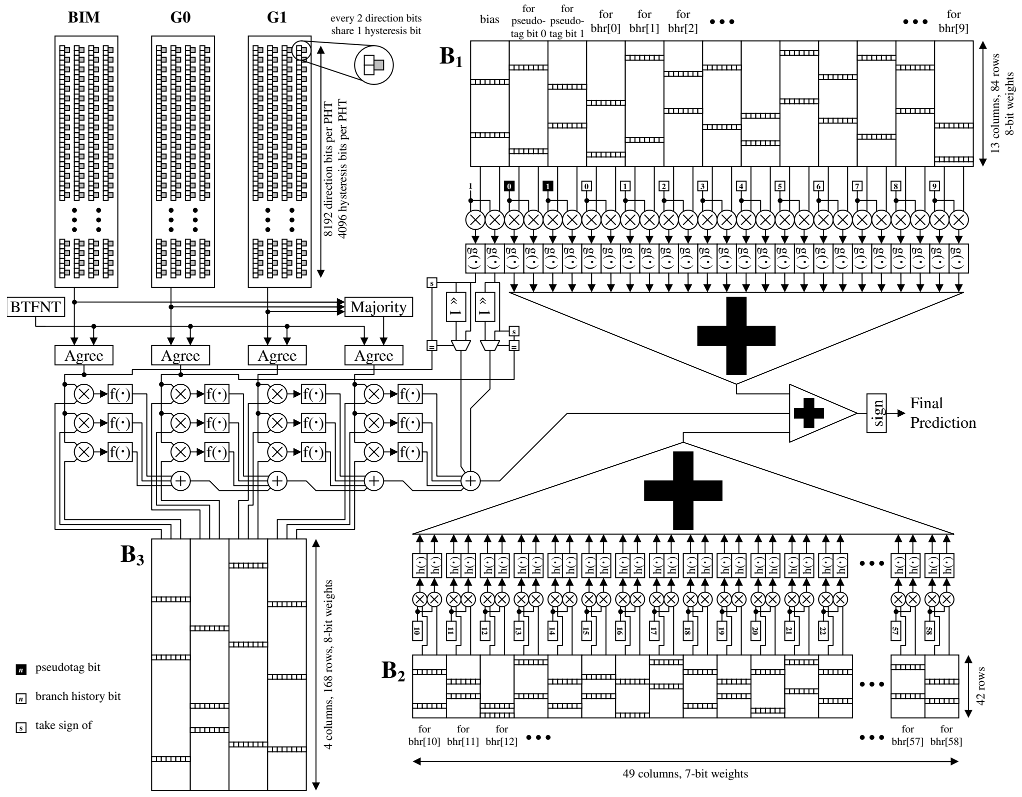

Figure 7: The Frankenpredictor. Indexing logic not shown.

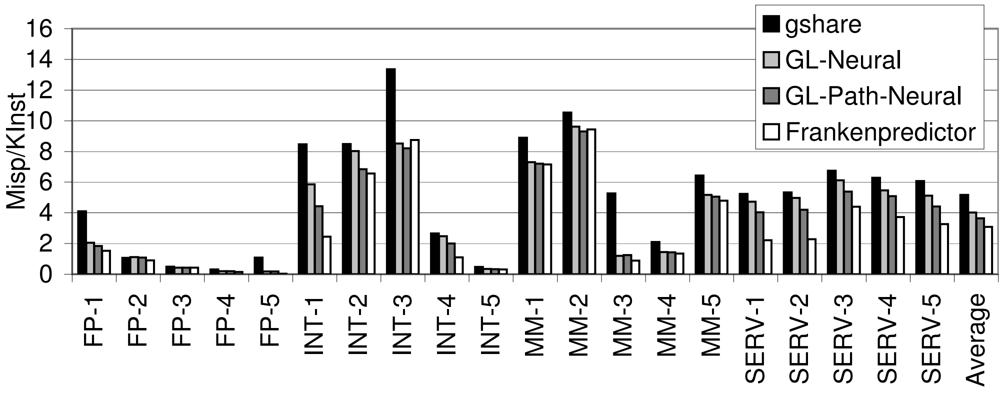

Figure 8: Prediction accuracy of three baseline predictors and the Frankenpredictor.

functions, fusion of the outputs from the three gskew-agree PHTs, and skewed redundant indexing all result in a 2-4% increase in the misprediction rate. Leaving out agree prediction and the pseudotag bits cause a smaller 1-2% increase in mispredictions, but on the other hand it makes sense to include these optimizations because the hardware cost in terms of state, logic and complexity is quite low. Except for synergistic reinforcement, the remaining optimizations are basically tweaks to the hashing that have a relatively low impact on overall accuracy. The synergistic reinforcement is very specific to the organization of the Frankenpredictor, and it has so little impact that the benefit likely does not justify the additional logic required to implement it.

7. Conclusions

The Championship Branch Prediction competition provided a large amount of flexibility in designing the prediction algorithms with respect to practicality for real hardware implementations. In this paper, we revisit some of the optimizations of the Frankenpredictor to study how these techniques affect more conventional prediction algorithms. Some of the optimizations are too complex, too specific to the Frankenpredictor, or provide too little benefit to be useful in general. Simple piecewise linear translation functions may be realized with some simple logic consisting of not much more than some shifters, multiplexers and other basic combinatorial logic. Non-uniform resource allocation for the perceptron table of weights involves a straightforward reorganization of the SRAM arrays. Our initial results indicate that these techniques provide a prediction accuracy benefit on existing algorithms.

References

[1] Veerle Desmet, Hans Vandierendonck, and Koen De Bosschere. A 2bcgskew Predictor Fused by a Redundant History Skewed Perceptron Predictor. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[2] Daniel 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.

[3] Daniel A. Jiménez. Idealized Piecewise Linear Branch Prediction. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[4] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The Impact of Delay on the Design of Branch Predictors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 4–13, Monterey, CA, USA, December 2000.

[5] Daniel A. Jiménez and Calvin Lin. Neural Methods for Dynamic Branch Prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.

[6] Gabriel H. Loh. The Frankenpredictor. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[7] Gabriel H. Loh and Dana 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.

[8] Pierre Michaud, Andre Seznec, and Richard 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.

[9] Andrè Seznec. Revisiting the Perceptron Predictor. PI 1620, Institut de Recherche en Informatique et Systèmes Aléatoires, May 2004.

[10] Andrè Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis 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.

[11] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale 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.