Multiperspective Perceptron Predictor

Daniel A. Jiménez
Department of Computer Science and Engineering
Texas A&M University

Abstract

I present a branch predictor based on the idea of viewing branch history from multiple perspectives. The predictor is a hashed perceptron predictor using previous outcomes and addresses of branches organized in ways beyond the traditional global and local history. This *multiperspective perceptron predictor* achieves a mean mispredictions per 1000 instruction (MPKI) rate of 5.335 given an 8.25KB hardware budget, 4.147 for a 64.25KB hardware budget, and 3.238 for an unlimited hardware budget.

1 Introduction

Branch predictors typically use global and/or local (*i.e.* per-branch) history to match or find correlations between previous branch outcomes and the current branch. However, there are many other ways to organize history beyond global and local. For example, inspired by the work of Albericio *et al.* [1], Seznec *et al.* propose using the inner-most loop iteration counter (IMLI) [10] as a feature in a hashed perceptron predictor together with local and global history as an additional component of a TAGE-SC-L predictor.

This paper describes a hashed perceptron predictor [12] that incorporates many kinds of branch history to make a prediction.

2 Background

2.1 Hashed Perceptron Predictor

The hashed perceptron predictor is similar to an idea of Loh and Jiménez called *modulo path-history* [6], while O-GEHL [7] is a very specific instance of the general technique. The idea is to have several tables, each indexed by a different hash of branch history. The tables have somewhat wide saturating confidence weights. The selected weights are summed and the prediction is taken if the sum is at least zero, not-taken otherwise. On a mispredict or low-confidence correct prediction, the corresponding weights are incremented if the branch is taken, decremented otherwise. The hashed perceptron predictor, like modulo path-history and O-GEHL, improves over the original perceptron predictor [5] by breaking the one-to-one correspondence between weights and history bits, allowing a more efficient representation. Perceptron-based predictors are currently in use in products made by AMD and Oracle.

3 Multiperspective Perceptron Predictor

The multiperspective perceptron predictor is a hashed perceptron predictor that uses not only hashed global path and pattern histories, but a variety of other kinds of features based on various organizations of branch histories. To index the prediction tables, the hash value of a feature is computed using recent history information, hashed together with the address of the branch to be predicted, then taken modulo the size of the prediction table. The weight corresponding to that index in the table is read, then all such weights for all features are summed and thresholded to make a prediction of taken or not taken. After exploring many organizations, I found the following features useful for branch prediction:

3.1 Traditional Features

The following features from traditional branch predictors are used:

GHIST Global history is the outcome of a branch shifted into a large register, with 0 meaning not taken and 1 meaning taken. It is hashed by bitwise exclusive-ORing multi-bit blocks of histories. Parameters to this feature give the starting and ending indices of the history register to hash.

PATH Path history is the sequence recent branch addresses. Addresses truncated to 16 bits are shifted into an array. Parameters include a depth, a shift, and a mixing style. The array is hashed by accumulating a hash value up to the given depth and doing, depending on the mixing style, either shifting the accumulator and adding the next address, or exclusive-ORing a bit of the accumulator with a given bit of the next address.

LOCAL Local history is a first-level table of per-branch shift registers selected by a hash of the branch address, then hashed as an index into a second level table of perceptron weights.

GHISTPATH This is a combination of GHIST and PATH. The GHIST and PATH features are exclusive-ORed together as they are computed.

SGHISTPATH This is an alternate formulation of GHISTPATH where a range of histories can be specified. It is inspired by the strided sampling hashed perceptron predictor [4] from the last competition.

BIAS The value of this feature is 0. It will be exclusive-ORed with the branch address to form an index into the table. This feature tracks the tendency of a branch to be taken or not, regardless of other branch history.

3.2 Novel Features

The following novel features are used:

IMLI The inner-most loop iteration counter [10] is used as a feature, but with an additional twist. In the original formulation, when a backward branch is encountered, the IMLI count is incremented if the branch is taken, otherwise it is reset. This captures the behavior of loops with back-edges at the bottom, which smart compilers endeavor to produce when optimizing for performance. Some compilers are not smart or optimize for size, so I explore an alternate IMLI: when a forward branch is encountered, the IMLI count is incremented when the branch is not taken, and reset when it is taken, representing a loop exit. This represents the case where the loop exit is at the top of the loop body, followed by an unconditional jump at the bottom. Both the backward and forward formulations of IMLI turn out to be useful for the CBP2016 traces. For the 8.25KB hardware budget, only the forward version of IMLI is used since the backward (original) version does not seem to help.

MODHIST Some (most) branches in the global history are not correlated to branch outcome, but a single bit different in two histories can lead to two different hash values, causing longer training times and increased aliasing pressure. Traditional (one-to-one) perceptron predictors can overcome this problem since they find correlations between individual branch outcomes and not hashed histories, they are still susceptible to what I call *branch misalignment*. Suppose a branch branches over another branch. The second branch sometimes appears in the branch history and sometimes does not appear, leading to the same branch outcomes appearing in different locations in the global history. Neither traditional nor hashed perceptrons, nor other hashed-based predictors such as TAGE, can handle this problem without expending additional table entries and increasing training time. I propose "modulo history" where only branches with addresses congruent to 0 modulo some modulus are recorded. Incongruent branches causing misalignment are filtered out of the history and ignored. The problem is that those filtered branches may indeed have some correlation; that correlation is hopefully captured by the other features. Modulo history has two parameters: the modulus (a small integer), and the history length.

MODPATH Modulo path history is the same as modulo history, but uses branch addresses instead of outcomes. The parameters are the same and the hash is computed similarly to the PATH feature.

GHISTMODPATH This feature combines modulo history with modulo path history in a way analogous to GHISTPATH above.

RECENCY The predictor keeps a fixed-depth recency stack of recently encountered branches managed with least-recently-used replacement. The feature hashes the addresses in the stack. The parameters are the depth into the stack to hash, a shift by which to shift the accumulator after hashing, and a mixing style parameter similar to the PATH feature.

RECENCYPOS The depth in the recency stack where a branch address is encountered, or the fact that the branch is not present in the stack, have a surprising correlation with branch outcome. This feature has a single parameter: the depth into the stack to search. The hash value is the position where the address was found, or the maximum table index if the address was not found.

BLURRYPATH Traditional branch path history is a precise record of the sequence of recent branches. "Blurry" path history records larger-granularity regions where branches have been encountered, and only shifts a region into the history when a new region is entered. Regions are computed as branch addresses right shifted by a certain amount given as a parameter. When a branch from a new region is encountered, the previous region is shifted into the history. The feature's parameters are the amount to shift, the depth within the history to hash, and a parameter that controls how much to shift each region number while generating the hash.

ACYCLIC This feature keeps a history register H of length n and records the outcome of a branch with address PC in $H[PC(\mathrm{mod}n)]$. The intuition is that we would like to remove the effect of loops (*i.e.* cycles) in the history and just keep the most recent outcome of any branch. Two kinds of acyclic history are used: one where H is an array of branch outcomes, and another where it is an array of hashed addresses of the corresponding branches. The parameters to the feature are the size of H, the amount by which to shift hashed elements of H, and a mixing style as in PATH.

3.3 Discussion of Features

None of the novel features are particularly good predictors by themselves. However, together with each other and with the traditional features, they provide alternate perspectives on branch history and allow finding new correlations. The features mentioned above are the ones that helped improve accuracy on the CBP2016 traces.

4 Optimizations

This section describes how I optimized my predictors.

Feature Selection Feature selection was done using a genetic algorithm followed by a hill-climbing phase where hundreds of thousands of combinations of features were evaluated on a subset of the CBP2016 traces. The best features I could find are given in the source code as an array of `history_spec` structs.

Coefficients As in previous work [2, 3], I found that multiplying each weight read from each table by a coefficient improved prediction accuracy by allowing tables with more accuracy to have a more important role in the prediction. Once the baseline features were chose, I used a genetic algorithm followed by hill climbing to select coefficients for each feature.

Bit Width Optimization and Table Size My framework allows for optimizing the bit width for the weights in each feature, but I did not find much improvement this way. One of the tables for the small predictor is 5 bits wide; the rest are 6 bits. Similarly, I tune table sizes for some of the features, allowing the rest of the features to split equally the remaining hardware budget for weights.

Shared Magnitudes Previous predictors have weights that share lower-order bits [9, 8]. In my predictor, weights are represented in sign/magnitude format with two signs sharing a single magnitude. The sign bit is the most important bit in the prediction, so it makes sense that there should be more signs than magnitudes.

Branch Filter Filtering branches that have either never or always been taken helps control aliasing in the predictor tables. I implement two Bloom filters that remember whether a branch has ever been taken or ever not been taken. When a branch is present in both filters, it is predicted by the perceptron predictor; otherwise, it is predicted to behave as it always has before. When a branch is first encountered, *i.e.* it is not in either filter, it is predicted using a stack prediction of not taken. Bloom filters become full over time, so the predictor keeps track of the occupancy of the filters and periodically decays random filter entries to provide slightly improved filtering in the presence of program phase behavior.

Transfer Function Multi-layer perceptrons use a non-linear transfer function between layers. It makes sense to try applying a transfer function to the perceptron weights before they are summed, since there may be a non-linear relationship between a weight value and the probability that a branch is taken. After much experimentation, I found that a transfer function of the form $f(x) = (a + bx)/(1 + cx + dx^2)$ gave a significant improvement over the identity function. The function is represented as a ROM table of values indexed by weight magnitude. Figure 1 shows the transfer function used for 6-bit weights (5-bit magnitudes) in the predictor. The function has been tweaked slightly to improve accuracy; in particular, as sign/magnitude representation allows for both positive and negative zero, the function is different for the different representations of zero.

Alternate Prediction on Low Confidence The magnitude of the output of the perceptron gives a confidence in the prediction. When it is very low, the prediction has a high chance of being incorrect. In these cases, the predictor chooses a subset of the features that have demonstrated good accuracy to make an alternate prediction. The predictor keeps track of the individual accuracy of each feature as though that feature alone had made a prediction. Of the 37 features, the predictor chooses the 20 with the fewest mispredictions to compute an alternate sum yielding a prediction.

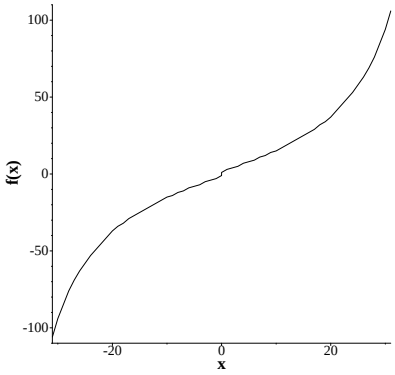

Figure 1: Transfer Function

Adaptive threshold training. Seznec found that good accuracy is achieved when the number of updates to due mispredictions is roughly the same as the number due to low-confidence predictions [7]. He gives an algorithm to adapt the threshold to maintain this property and I use that algorithm.

Hashing with IMLI and RECENCYPOS Some of the indices into the tables are additionally hashed with one of the IMLI counters or the recency position of the branch being predicted. This gives additional context to those indices resulting in a slight improvement in accuracy.

Extra information. Bits from the addresses of other control-flow instructions (e.g. unconditional branches, calls, and returns) are also considered in the branch history.

4.1 Making a Prediction

To make a prediction, first we check the filter. If the branch has only exhibited one behavior, we predict that it will continue to behave that way and prediction stops. Otherwise, we compute the perceptron output. We go through all the tables of weights in sequence. Each feature with its parameters are used to hash the various histories together with the branch address to yield an index into that feature's table that selected a weight. The transfer function is applied to the weight and the results are summed. If the sum is below one, the branch is predicted taken, otherwise it is predicted not taken.

4.2 Updating the Predictor

When the outcome of the branch is known, the algorithm decides whether to update the predictor. It uses the perceptron training rule to decide when to update. It must update the predictor for an incorrect prediction. If the prediction is correct, the magnitude of the perceptron output is compared to a threshold θ. If the magnitude is below θ then predictor is updated. To update the predictor, each weight that was used to make the prediction is incremented if the branch was taken or decremented otherwise. The weights are incremented or decremented with saturating arithmetic. A "fudge factor" is applied to the perceptron output to bias the training algorithm to balance the increased magnitude due to the transfer function.

5 Size of the Predictors

Table 1 shows the amount of state consumed by the 8.25KB and 64.25KB predictors. The 64.25KB predictor uses 37 features indexing 37 tables and the 8.25KB predictor used 16 features for 16 tables. There are many run-time constants (e.g. the sizes of structures etc.) that I do not count against the storage budget since they are an immutable part of the the algorithm just like the statements in the code. I also do not count storage for short-term computations e.g. loop counters or other temporary variables whose values do not persist from one prediction to the next. The features and transfer functions can be considered as large run-time constants.

6 Contribution of Features

Figure 2 shows the contribution of the 16 individual features of the multiperspective perceptron predictor for an 8.25KB hardware budget. Each bar represents the increase in MPKI when the corresponding feature is replaced by a BIAS feature. That is, each bar measures the accuracy lost when replacing the feature with the overall bias of the branch. (See the source code for an in-depth explanation of the parameters.)

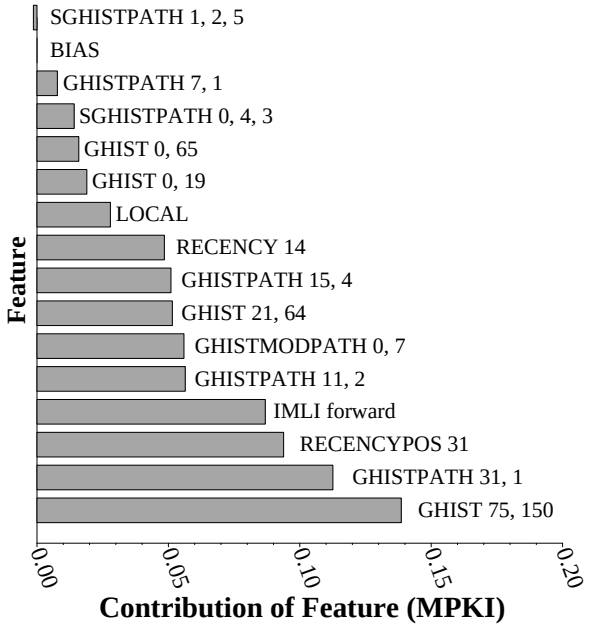

Figure 2: Contribution of individual features to accuracy

The feature with the greatest contribution was GHIST hashing the 75th through 150th most recent global history bits; 0.13 MPKI are lost when this feature is replaced by BIAS. The next most valuable feature is GHISTPATH hashing the 31 most recent branch PCs and outcomes for a contribution of 0.11 MPKI. RECENCYPOS is the third most valuable feature, contributing close to 0.1 MPKI.

This method of assigning value to features is not perfect; the BIAS feature seems to contribute nothing because replacing it with itself makes no difference. In designing the predictor, I found that BIAS does contribute to reducing mispredictions. Interestingly, one of the SGHISTPATH features seems to contribute negative value; replacing it with BIAS actually improves MPKI. That feature seemed to contribute value when I was exploring the design space with the genetic algorithm and hill-climbing, but because of time concerns I had to use truncated traces (basically simpoints [11]). When run with the full traces, it turns out that feature seems to hurt performance slightly. Oops.

Structure	# bits, 64.25KB predictor	# bits, 8.25KB predictor
Global history	337 bits	151 bits
IMLI counters	2 counters × 32 bits = 64	32 bits
Global path	106 × 16 = 1696	32 × 16 = 512 bits
Misprediction monitoring bits	37 counters × 24 bits = 888	16 × 24 = 384 bits
Modulo history bits	31 bits	8 bits
Modulo path bits	27 entries × 16 bits = 432	8 × 16 = 128 bits
Local history bits	510 histories × 11 bits = 5610	48 × 11 = 528 bits
Recency stack bits	31 entries × 16 bits = 496	31 × 16 = 496 bits
Blurry path bits	154	0 bits
Acyclic history bits	12	0
Branch filter bits	2 × 18025 = 36050	0
Sized weight table bits	10704 total weights among 7 tables × (5+2) bits = 74928	10409 bits
Rest of table bits	1931 weights × 30 tables × (5+2) bits = 405510	54876 bits
Total Bits	526208 = 64.23KB	67524
Total Allowed Bits	526336 = 64.25KB	67584 = 8.25KB

Table 1: Sizes of Prediction Structures

7 Acknowledgment

This research is supported in part by NSF grants CCF-1332598 and CCF-1162215. Special thanks to the organizers of CBP2016. The infrastructure and workloads they have contributed raise the bar for branch prediction research methodology.

References

[1] Jorge Albericio, Joshua San Miguel, Natalie Enright Jerger, and Andreas Moshovos. Wormhole: Wisely predicting multidimensional branches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 509–520, Washington, DC, USA, 2014. IEEE Computer Society.

[2] Renée St. Amant, Daniel A. Jiménez, and Doug Burger. Low-power, high-performance analog neural branch prediction. In *Proceedings of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*. IEEE Computer Society, November 2008.

[3] Daniel A. Jiménez. An optimized scaled neural branch predictor. In *In Proceedings of the 29th IEEE International Conference on Computer Design (ICCD-2011)*, October 2011.

[4] Daniel A. Jiménez. Strided sampling hashed perceptron predictor. In *Proceedings of JWAC-4: Championship Branch Prediction*, June 2014.

[5] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pages 197–206, January 2001.

[6] Gabriel H. Loh and Daniel A. Jiménez. Reducing the power and complexity of path-based neural branch prediction. In *Proceedings of the 2005 Workshop on Complexity-Effective Design (WCED'05)*, pages 28–35, June 2005.

[7] André Seznec. Genesis of the o-gehl branch predictor. *Journal of Instruction-Level Parallelism (JILP)*, 7, April 2005.

[8] André Seznec. Tage-sc-l branch predictors. In *Proceedings of JWAC-4: Championship Branch Prediction*, June 2014.

[9] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakakis Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[10] André Seznec, Joshua San Miguel, and Jorge Albericio. The inner most loop iteration counter: A new dimension in branch history. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 347–357, New York, NY, USA, 2015. ACM.

[11] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[12] David Tarjan and Kevin Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM Trans. Archit. Code Optim.*, 2(3):280–300, September 2005.