Multiperspective Perceptron Predictor with TAGE

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, combining perceptron-based prediction and TAGE. A hashed perceptron predictor uses previous outcomes and addresses of branches organized in ways beyond the traditional global and local history. This *multiperspective perceptron predictor with TAGE* achieves a mean mispredictions per 1000 instruction (MPKI) rate of 5.226 given an 8.25KB hardware budget, 4.048 for a 64.25KB hardware budget, and 2.967 for an unlimited hardware budget.

1 Introduction

Recent TAGE-based predictors use a hashed perceptron predictor [11] as a "statistical corrector." Although the global-history-based TAGE predictor is very good at matching branch histories to find a good prediction for the current branch, the perceptron predictor can find correlations when the matching fails. My predictor significantly augments the perceptron component of TAGE-SC-L to provide superior prediction. The perceptron predictor component described in this paper incorporates several different kinds of branch history to make a prediction.

This entry into the competition includes source code made available by Seznec *et al.* in the IMLI paper [2]. I modify the code to include my perceptron predictor. I tune the TAGE table sizes to accommodate the perceptron predictor and stay within the hardware budget. The TAGE, local, and loop predictors are adequately explained in previous work, so this paper focuses on the novel component: the multiperspective perceptron predictor.

2 Background

2.1 TAGE

The TAGE family of predictors has been quite successful in branch prediction competitions [7, 8, 9]. TAGE makes a prediction by searching tagged tables of geometrically increasing history lengths for hashed histories, then making a prediction with the longest match. A clever allocation strategy for table entries allows TAGE to be very space efficient. Recent TAGE predictors [9, 2] have used a perceptron predictor as a "statistical corrector" to find correlations between branch history and outcomes when the TAGE algorithm fails. This paper demonstrates a significantly improved perceptron predictor used as the statistical corrector for TAGE.

2.2 Hashed Perceptron Predictor

The hashed perceptron predictor is similar to an idea of Loh and Jiménez called *modulo path-history* [5], while O-GEHL [6] 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 [4] 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 in the context of a statistical corrector for TAGE:

3.1 Features

The following 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. Both the backward and forward formulations of IMLI turn out to be useful for the CBP2016 traces.

The code of Seznec *et al.* includes IMLI prediction, but I disable that code in favor of my own implementation that includes the alternate IMLI. The IMLI-OH component is not used as it did not improve accuracy.

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 $H[PC(\text{mod} n)]$. 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.

3.2 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 I used a superset of the features to design a perceptron-only entry to the branch prediction competition (described in a different paper). The features for this predictor were chosen 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. For the perceptron with TAGE predictor, I eliminated the features duplicated by TAGE (e.g. bias, global and local history) and, of the remaining features, use hill-climbing to choose a subset of features that worked well.

Coefficients As in previous work [1, 3], I found that multiplying each weight read from each perceptron 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.

Hashing with IMLI Some of the indices into the tables are additionally hashed with one of the IMLI counters. 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, the TAGE and perceptron components are evaluated. The TAGE prediction proceeds as described in the literature: a large table of global history confidence counters is searched using different hashes of global history, and a matching history yields the TAGE prediction. The loop predictor also makes a prediction, and if the algorithm decides that the loop predictor is likely to be correct, its prediction replaces the TAGE prediction. The perceptron predictor computes a sum of weights selected by indexing a bias table, a local history table (in the case of the 64KB and unlimited predictor, three local history tables are used), a global history table, a callstack table, and the various novel features described above. The sum is weighted by the coefficients for the various features. If the TAGE prediction was made with low confidence, the perceptron prediction is used. Otherwise, a confidence in the perceptron sum is computed and used to decide whether to use the perceptron prediction or the TAGE prediction. This algorithm is the same as that used in TAGE-SC-L; the novel part is the augmentation of the perceptron predictor with the features described above.

4.2 Updating the Predictor

To update the predictor, first the loop and TAGE predictors are updated as described in the literature on TAGE-SC-L. Then the perceptron predictor is updated. The thresholds used to compute the confidence that decides between TAGE and the perceptron predictor are updated on the strength and correctness of the perceptron prediction. Then the thresholds used to decide whether the perceptron predictor needs to be updated on a weak but correct prediction are updated with dynamic threshold training [6]. This threshold is referred to as θ in the original perceptron paper [4], and Seznec's code uses multiple θ values indexed by a hash of the PC. Finally, if the perceptron prediction was incorrect, or if the magnitude of the sum it computed is below the threshold just computed, then each weight used to accumulate the perceptron sum is incremented if the branch was taken, or decremented otherwise.

5 Size of the Predictors

Table 1 shows the amount of state consumed by the 8.25KB and 64.25KB predictors. 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 can be considered as large run-time constants.

6 Contribution of Features

Figure 1 shows the contribution of the individual features of the multiperspective perceptron predictors for a 64.25KB hardware budget. Not included are the local and global components of the statistical corrector, nor the contribution of TAGE. 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.)

The feature with the greatest contribution was BLURRY-PATH hashing the most recent 10 addresses shifted right by 8. The loss of this feature increases MPKI by 0.025. The

Structure	**# bits, 64.25KB predictor**	**# bits, 8.25KB predictor**
Perceptron weights tables	699 $\times$ 9 tables $\times$ 6 bits = 37746 bits	254 weights $\times$ 5 tables $\times$ 6 bits = 7620 bits
IMLI counters	2 counters $\times$ 32 bits = 64	64 bits
Modulo history	8 bits	8 bits
Recency stack	31 entries $\times$ 16 bits = 496	496 bits
Blurry path	645 bits	405 bits
Acyclic history	14	N/A
TAGE table entries	406016 bits	35712 bits
TAGE bias counters	$2^{14} + 2^{12}$ = 20480 bits	5120 bits
TAGE global history bits	1241	359 bits
TAGE alternate chooser counters	2048 bits	2048 bits
TAGE miscellaneous counters	37 bits	37 bits
SC global history	16 bits	16 bits
SC global history weights	15872 bits	7936 bits
SC update threshold counters	256 bits	256 bits
SC bias weights for	3072 bits	1536 bits
SC call stack weights	7936 bits	1984 bits
SC local history weights	10752 bits	2816 bits
SC local history bits	2816 bits	512 bits
SC second local history weights	7936 bits	N/A
SC third local history weights	5376 bits	N/A
SC local history stack and stack pointer	260 bits	N/A
SC chooser counters	21 bits	21 bits
SC additional local histories	432 bits	N/A
Loop predictor	2753 bits	624 bits
Total Bits	526293 = 64.24KB	67577 bits = 8.25KB
Total Allowed Bits	526336 bits = 64.25KB	67584 bits = 8.25KB

Table 1: Sizes of Prediction Structures

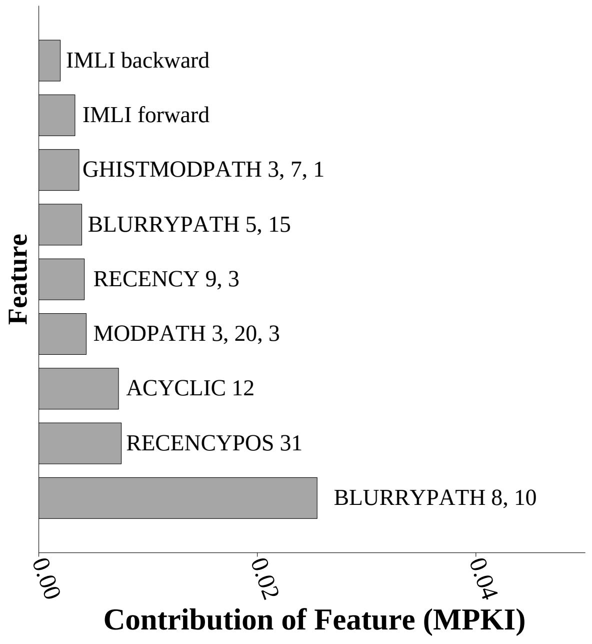

Figure 1: Contribution of individual features to accuracy

other features were roughly an order of magnitude less valuable, with RECENCYPOS contributing the most out of them.

7 Acknowledgments

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] 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.

[2] Jorge Albericio André Seznec, Joshua San Miguel. The inner most loop iteration counter: a new dimension in branch history. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-48, pages 509–520, Washington, DC, USA, 2015. IEEE Computer Society.

[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 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.

[5] 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.

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

[7] André Seznec. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 9, May 2007.

[8] André Seznec. A 64 kbytes isl-tage branch predictor. In *Proceedings of JWAC-2: Championship Branch Prediction*, June 2011.

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

[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] 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.