Journal of Instruction-Level Parallelism 9 (2007) 1-13 Submitted 4/07; published 5/07

An Idealistic Neuro-PPM Branch Predictor

Ram Srinivasan[†*] RSRI@LANL.GOV
Eitan Frachtenberg[†] ETCS@CS.HUJI.AC.IL
Olaf Lubeck[†] OLUBECK@LANL.GOV
Scott Pakin[†] PAKIN@LANL.GOV
Jeanine Cook[*] JCOOK@NMSU.EDU

[†] *CCS-1 Performance and Architecture Lab, Los Alamos National Laboratory.*
[*] *Klipsch School of Electrical and Computer Engineering, New Mexico State University.*

Abstract

Historically, Markovian predictors have been very successful in predicting branch outcomes. In this work we propose a hybrid scheme that employs two Prediction by Partial Matching (PPM) Markovian predictors, one that predicts based on local branch histories and one based on global branch histories. The two independent predictions are combined using a neural network. On the CBP-2 traces the proposed scheme achieves over twice the prediction accuracy of the *gshare* predictor.

1. Introduction

Data compression and branch prediction share many similarities. Given a stream of symbols (e.g., ASCII characters for text compression or past branch outcomes for branch prediction), the goal in both cases is to predict future symbols as accurately as possible. One common way of achieving this goal in data compression is Prediction by Partial Matching (PPM) [1, 5]. PPM is a Markov predictor in which the prediction is a function of the current state. The state information in an mth-order PPM predictor is an ensemble of the m most recent symbols. If the pattern formed by the m recent symbols has occurred earlier, the symbol following the pattern is used as the prediction.

As an example of how PPM works, consider the sample stream of binary symbols presented in Figure 1. To predict what symbol will appear at position 0 we look for clues earlier in the stream. For example, we observe that the previous symbol is a 1. The last time a 1 was observed—at position 2—the following symbol was a 1. Hence, we can predict that position 0 will be a 1. However, we do not have to limit ourselves to examining a single-symbol pattern. The last time 1 1 (positions 2 and 1) appeared was at positions 10 and 9 and the subsequent symbol (position 8) was a 0 so we can predict 0. We find the next longer pattern, 0 1 1 (at positions {3, 2, 1}), at positions {13, 12, 11} with position 10 predicting 1. The longest pattern with a prior match is 0 1 0 1 0 0 1 1 (positions {8, 7, 6, 5, 4, 3, 2, 1}) at positions {24, 23, 22, 21, 20, 19, 18, 17}. The subsequent symbol is 0 so we can choose 0 as our best guess for position 0's value.

Generally, predictors that use longer patterns to make a prediction are more accurate than those that use shorter patterns. However, with longer patterns, the likelihood of the same pattern having occurred earlier diminishes and hence the ability to predict decreases. To address this problem, an mth-order PPM scheme first attempts to predict using the m

24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	0	1	0	0	1	1	1	1	0	1	0	1	0	0	1	1	?

Figure 1: Sample stream.

most recent symbols. Progressively smaller patterns are utilized until a matching pattern is found and a prediction can thereby be made.

In the context of branch prediction, the symbols are branch outcomes. The past outcomes used in prediction can be either *local* or *global*. In a *local* scheme, the prediction for a given branch is based solely on the past outcomes of the (static) branch that we are trying to predict. In contrast, a global scheme uses outcomes from all branches to make the prediction. In this paper we propose a hybrid PPM-based branch predictor that employs pattern matching on both local and global histories. The predictions based on the two histories are combined using a perceptron-based neural network [3] to achieve a high prediction accuracy. Hardware implementability is not our goal. Rather, we determine the best prediction accuracy that can be achieved from PPM-like schemes given virtually unlimited memory and processing time. This approach corresponds to the "idealistic" track of the 2nd JILP Championship Branch Prediction Competition [2].

There are many optimizations and heuristics that improve the speed, accuracy, and memory utilization of the basic PPM method. In Section 2 we present our implementation technique and a set of modifications that prove empirically to be beneficial to performance or resource usage. Finally, we draw some conclusions in Section 3.

2. Implementation

Figure 2 shows the high-level block diagram of the Neuro-PPM predictor. Our scheme consists of two PPM-based predictors [1], one that uses local-history information and the other that uses global-history information to identify patterns and predict branch outcomes. For a given branch, both PPM predictors are invoked to predict the branch outcome. The two predictions are combined using a perceptron-based neural network [3]. The rest of this section describes the PPM predictors and the neural-net mixer.

2.1. Global History PPM Predictor

We first describe the global PPM predictor and then detail the differences compared to the local predictor. Figure 3 shows the block diagram for the PPM predictor that uses global history. An m-bit shift register records global history and reflects the outcome of the last m dynamic branches (a bit's value is one for branch taken, zero otherwise). Each time a branch outcome becomes available, the shift register discards the oldest history bit and records the new outcome. When a prediction is made, all m bits of history are compared against previously recorded patterns. If the pattern is not found, we search for a shorter pattern, formed by the most recent $m-1$ history bits. The process of incrementally searching for a smaller pattern continues until a match is found. When a pattern match occurs, the outcome of the branch that succeeded the pattern during its last occurrence is returned as the prediction. The total number of patterns that an m-bit history can form is $\sum_{L=1}^{m} 2^L$.

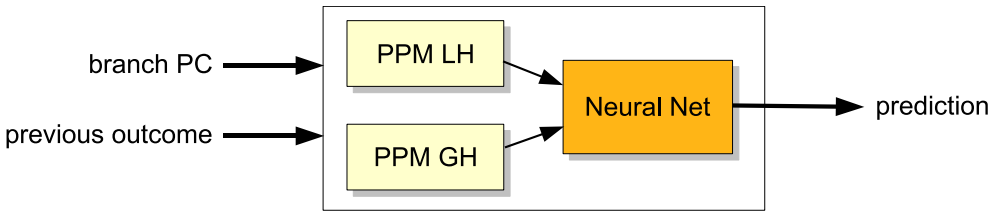

Figure 2: The Neuro-PPM predictor.

To efficiently search the vast pattern space, we group patterns according to their length and associate each group with a table. For example, table t is associated with all patterns of length t and table 1 with all patterns of length 1. When making a prediction we use all m history bits to compute a hash value of the pattern. The n least-significant bits of the computed hash are used to index into one of the 2^n rows of table m. We resolve the collisions caused by different hash values indexing into the same row of the table by searching a linked list associated with this row. Each node in the linked list contains the pattern hash and the predicted outcome. If a hash match is found the prediction is returned. Otherwise, we continue to search for successively smaller patterns using the corresponding tables. During update, when the actual outcome of the branch becomes available, we update all m tables. When a previously unseen pattern of a given length is encountered, a new node is added to the corresponding linked list. While this general principle works well in many scenarios, the accuracy of the prediction can be further improved by the following heuristics:

- program-counter tag match
- efficient history encoding
- capturing pattern bias

To restrict the memory requirement and to decrease the computational time, we apply the following heuristics:

- improved hash function
- periodic memory cleanup
- pattern length skipping
- exploiting temporal pattern reuse

For example, applying these heuristics decreases the MPKI (Mispredicts Per Kilo Instruction) for *twolf* by 30% and improves the simulation time by a factor of 700. We now describe these heuristics in detail.

Program-counter tag match One drawback of the base scheme is that it cannot discriminate among global histories corresponding to different branches. For example, assume that branch b_{21} is positively correlated with branch b_8 while branch b_{32} is negatively correlated with b_8. If the global histories when predicting b_{21} and b_{32} are identical, the patterns destructively interfere and result in 100% wrong predictions. We address this problem by

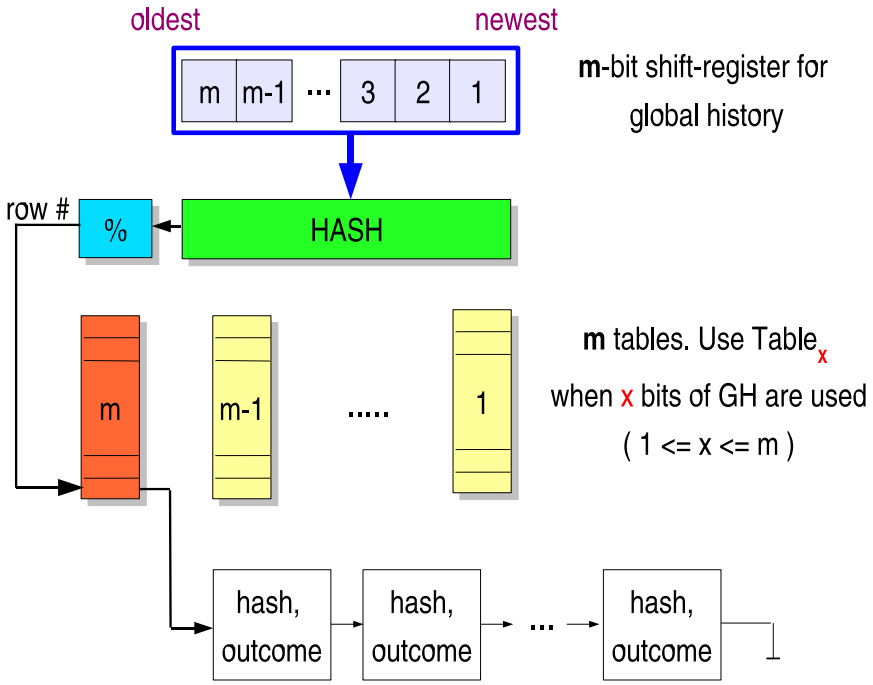

Figure 3: The global PPM predictor.

storing the program counter (PC) in addition to the pattern hash in each node of the linked list associated with a hash table entry. We return a prediction only when both the pattern hash and the PC match. One might wonder if hashing schemes such as those employed by the *gshare* predictor [4] in which the PC is exclusive or'ed with history bits to index into the table would eliminate the need for PC tag matching. Though such schemes significantly reduce hash collisions they do not eliminate them. We have experimentally determined that even with a more sophisticated hashing scheme than that used by *gshare*, PC tag matching improves prediction accuracy for the CBP-2 traces. Figure 4 shows the percent improvement in prediction accuracy for the CBP-2 traces when PC tagging is used. As that figure indicates, PC tagging improves prediction accuracy by an average by 5.4% across the 20 benchmarks and by as much as 18.3% (for *vortex*).

Efficient history encoding One disadvantage of our m-bit shift register scheme as described thus far is that the history of a long loop displaces other useful information from the history tables. For example, consider the following code fragment:

```
k = 0;
if (i == 0) k=1;        // #1
for (j=0; j<LEN; j++)   // #2
{ c += a[j]; }
if (k != 0) c -= 10;    // #3
```

Branches corresponding to lines #1 and #3 are positively correlated. That is, if the condition `i==0` is *true*, then `k!=0` is guaranteed to be *true*. However, the loop at line #2

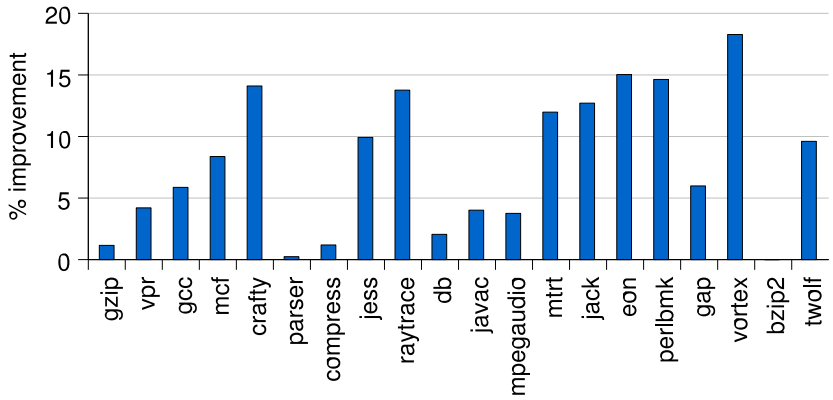

Figure 4: Improvement in prediction accuracy due to PC tag match.

that interleaves the perfectly correlated branches will pollute the global history with `LEN` − 1 *takens* and one *not taken*. If `LEN` is much larger than the global-history length (m), the outcome of the branch at line #1 is lost and therefore the correlation cannot be exploited when predicting the outcome of the branch at line #3. One solution to this problem is to increase the length of the global-history window. Because each additional history bit exponentially increases the memory requirement for storing the patterns, this solution is not very practical. An alternate solution is to compress the global history using simple schemes such as run-length encoding (RLE). With RLE, the m-bit shift register is replaced with n counters. These counters reflect the lengths of the most recent n strings, where a string is defined as a contiguous stream of zeros or ones. For example, a global history of 000011000 has an RLE representation of 4, 2, 3. If $m = 2$, the last two counter values (2 and 3) are stored. We use 8-bit counters in our implementation. To help differentiate a string of zeros from a string of ones, we initialize the counters to 0 or 128, respectively, at the start of a string. The chance of counter overflow is negligible because 99% of the sequences of zeros or ones in the global history are less than 100 elements long for the CBP-2 traces. During pattern search, the hash values are computed from the n RLE counters instead of the m-bit shift-register. Of all the CBP-2 benchmarks, RLE noticeably benefited only *raytrace* and *mtrt*. However, because the reduction in MPKI was significant in both cases—approximately 57%—and did not increase MPKI significantly in the other cases we decided to retain RLE in our implementation.

The reason that some benchmarks observe a significant benefit from RLE while others observe minimal benefit is explained by the string-length distributions of each trace. Figure 5 presents the cumulative distribution function (CDF) of the global-history string lengths observed in *mtrt* and *perlbmk*. It is clear from the CDF that strings of zeros and ones are significantly longer in *mtrt* than in *perlbmk*. Consequently, RLE is more frequently applicable to *mtrt* than *perlbmk* and therefore yields a much greater improvement in MPKI for *mtrt* than for *perlbmk*.

Pattern bias Instead of using only the last outcome as prediction for a given pattern, tracking a pattern's bias towards *taken* or *not taken* can significantly improve the prediction accuracy. $Bias_{taken}$ is given by P($taken|pattern$). The prediction is *taken* if $Bias_{taken} > 0.5$,

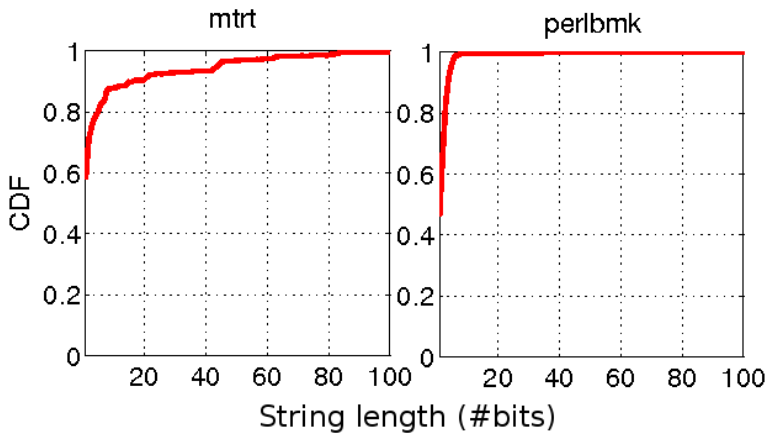

Figure 5: Cumulative distribution function of string length in *mtrt* and *perlbmk*.

suggesting that the pattern is biased towards *taken*. Pattern bias can be captured easily by associating each pattern with an up-down counter. Each time a given history pattern is seen the associated counter is incremented when the branch outcome following the pattern is *taken* and decremented when the outcome is *not taken*. The prediction is simply the sign of the counter. For the sample stream shown in Figure 1, the counter value associated with patterns of length one are: $\text{counter}_{\{1\}} = -2$ and $\text{counter}_{\{0\}} = +5$. This suggests that pattern $\{1\}$ is biased towards *not taken* and pattern $\{0\}$ towards *taken*.

It is well known that workloads exhibit phases of execution [7], with relatively homogeneous behavior within each phase. We believe that pattern bias exhibits phases. During certain phases, a pattern may be biased towards *taken* and in other phases the same pattern may be biased towards *not taken*. A non-saturating counter—or saturating counter with an excessively large saturation value—exhibits lags in tracking the bias and is therefore unable to track rapid phase changes. Conversely, a counter that saturates too quickly will fail to capture pattern bias. Figure 6 shows the outcome of the static branch at PC `0x8048ba0` in *crafty* when the 3-bit global history is 1 0 1. The Figure also shows the predicted outcome when the pattern bias is tracked using a 1-bit counter, 2-bit saturating counter and a non-saturating counter. Note that the 1-bit counter keeps track of only the last outcome. In each plot, we represent correct predictions with green bars and wrong predictions with red bars. The non-saturating scheme is influenced by the global bias of the pattern towards *taken* and is unable to track rapid phase changes. This scheme has a misprediction rate of 33%. The 1-bit counter is affected by short transients and mispredicts twice at every transition in the branch outcome. This scheme mispredicts 39% of the time. A 2-bit counter strikes the optimal balance between phase sensitivity and noise immunity, decreasing the misprediction rate to 21%.

Figure 7 quantifies the impact of counter size of prediction accuracy for *crafty*. The figure plots the percent improvement in prediction accuracy as a function of saturation value and indicates a maximum improvement of 12.7% relative to a non-saturating counter. For the CBP-2 traces we determined empirically that a counter that saturates at ± 8 delivers the best performance overall.

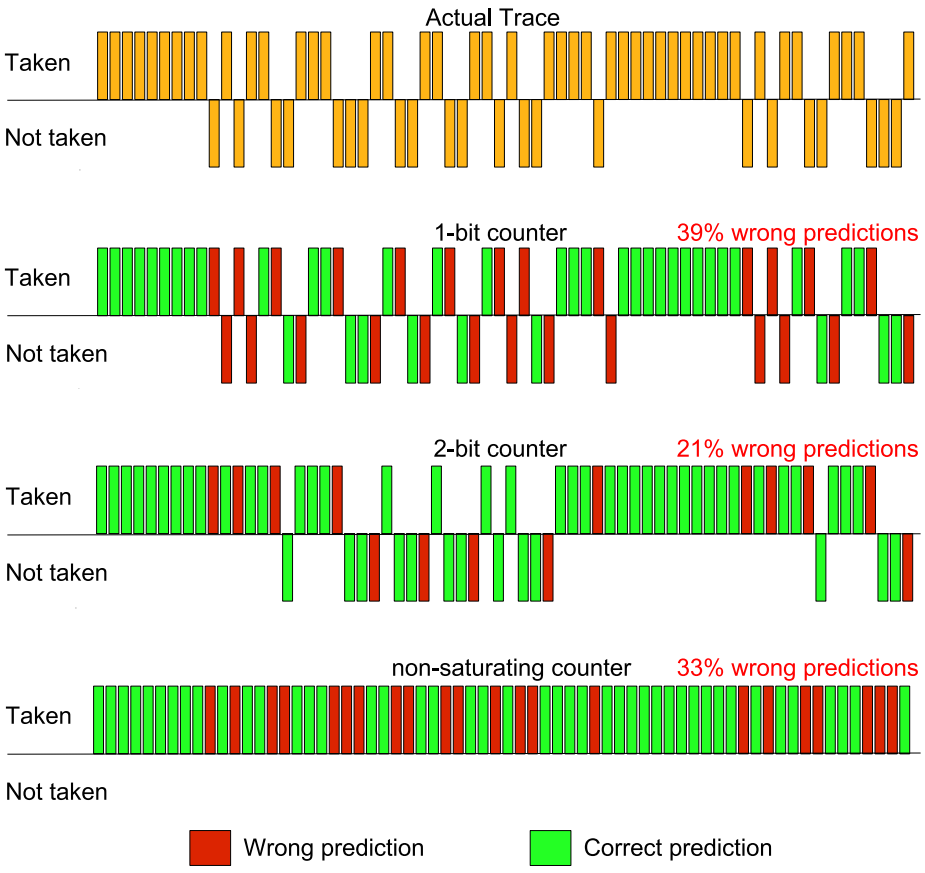

Figure 6: Tracking pattern bias in *crafty* using counters that saturate at different values.

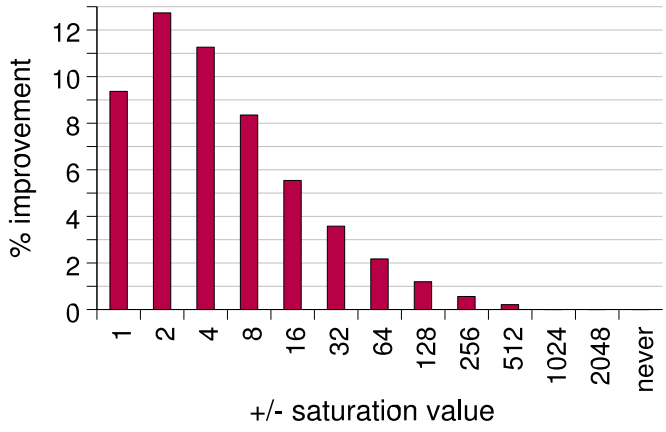

Figure 7: Percent improvement in *crafty*'s prediction accuracy when saturating bias counters are used in lieu of non-saturating counters.

Dynamic pattern length selection The baseline algorithm uses the longest pattern to predict a branch outcome. The implicit assumption is that longer patterns result in higher confidence in the prediction and are therefore more accurate. Although this is generally true, in some benchmarks such as *gzip* and *compress*, using a shorter pattern actually results in higher accuracy than matching longer patterns. To help dynamically select the best pattern length for a given branch, we track the prediction accuracy along with the PC and pattern hash in each node of the linked list. Rather than predicting based on the longest pattern match, we predict using the pattern that results in the highest accuracy. For *javac*, the misprediction rate decreased by 16% due to dynamic pattern length selection. The average improvement in prediction accuracy across the CBP-2 traces is 3%.

To help reduce simulation run time we made the following optimizations.

Hash function We experimented with various hash functions and empirically identified that the AP hash function [6] results in fewer collisions than other schemes. The lower number of collisions in turn improved the linked-list search time and resulted in 10X faster code execution than that achieved by using other hashing schemes. The AP hash is computed as follows:

```
inputs:  rle_cou[], n_cou, PC
output:  (h) pattern hash for the
         n_cou counters of rle_cou[]

for (h=i=0; i<n_cou; ++i)
{
  h = h ^ (i&1 == 0)?
      (h<<7 ^ rle_cou[i] ^ h>>3):
      ~(h<<11 ^ rle_cou[i] ^ h>>5);
}

h = h ^ PC;
```

The index into the pattern tables is obtained by considering the n least significant bits of the computed hash. Like the *gshare* predictor, the above scheme uses the PC in the hash computation. Although the AP hash significantly lowers hash collisions it does not eliminate them. We therefore resolve hash collisions by tag matching both the PC and the pattern hash in the linked list associated with the indexed row of a given table. Note that the primary function of the hash function is to speed up the pattern search process. Comparable hash functions have little effect on prediction accuracy.

Memory cleanup For the *twolf* benchmark, if 60 RLE counters are used for encoding the global history more than 4 GB of memory is required to store all the patterns. This leads to frequent page swaps and causes the simulation to take about 2 days to complete on the test system (a Pentium 4 machine with 1 GB of RAM). Because the CBP-2 rules allow only 2 hours to process all 20 traces we perform periodic memory cleanups to speed up the simulation. Specifically, we scan all the linked lists at regular intervals and free the nodes that have remained unused since the last cleanup operation. The frequency of the

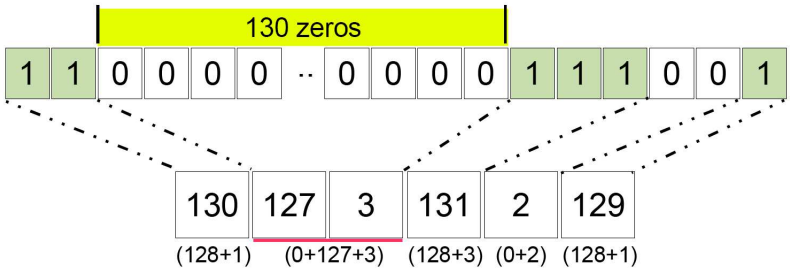

Figure 8: Run-length encoding of local history.

cleanup operation is dynamically adjusted to restrict the memory usage to a preset limit of 900 MB. This results in an almost 50X increase in simulation speed for the CBP-2 traces. However, the main disadvantage of memory cleanup is the loss in prediction accuracy. We observed a loss in prediction accuracy of 10% for *twolf* and 4% for *crafty*, for example.

Pattern length skipping In the original algorithm, when a pattern of length m is not found, we search the history for the pattern of length $m-1$. This process of searching for incrementally smaller patterns continues until a match is found. To lower memory usage and computation time requirements we modified our implementation to skip many pattern lengths. Using 60 RLE counters for global history encoding we found that searching patterns of length $\{m, m-5, m-10, \ldots, 1\}$ for the *gzip* benchmark produced a fivefold faster simulation than searching patterns of length $\{m, m-1, m-2, \ldots, 1\}$. Also, because the memory usage of an $m-5$ search granularity is considerably smaller than an $m-1$ search, memory cleanup is performed less frequently, which leads to a slight improvement in prediction accuracy.

Temporal reuse To exploit the temporal reuse of patterns, nodes matching a given hash value and PC are moved to the head of the linked list. Doing so decreases the pattern search time and produces an almost 3X improvement in simulation time across the test suite.

2.2. Local-History PPM Predictor

The local-history PPM predictor uses the same algorithm and optimizations as those used in the global predictor. However, it uses different history information for the pattern match. Instead of using a single set of RLE counters, the local PPM predictor uses one set of counters for each static branch in the trace. As in the global-history PPM predictor, patterns from all branches are grouped according to length and stored in up to m tables. During pattern search, both the pattern hash and the PC of the branch being predicted are matched. Because consecutive strings of zeros and ones are significantly longer in local history than in global history, 8-bit RLE counters are insufficient for the run-length encoding. One solution to this problem is to increase the counter size (e.g., 32 bits). This increase, however, can result in long predictor warmup time and in certain cases will perform no better than an *always taken* or *always not taken* prediction scheme. Therefore, we restrict the counters to 8 bits and handle counter saturation by pushing a new counter that represents the same bit as the saturated counter and dequeuing the oldest counter in the RLE list. Figure 8 illustrates an example of how sequences are encoded. A pattern of *not takens* (shown

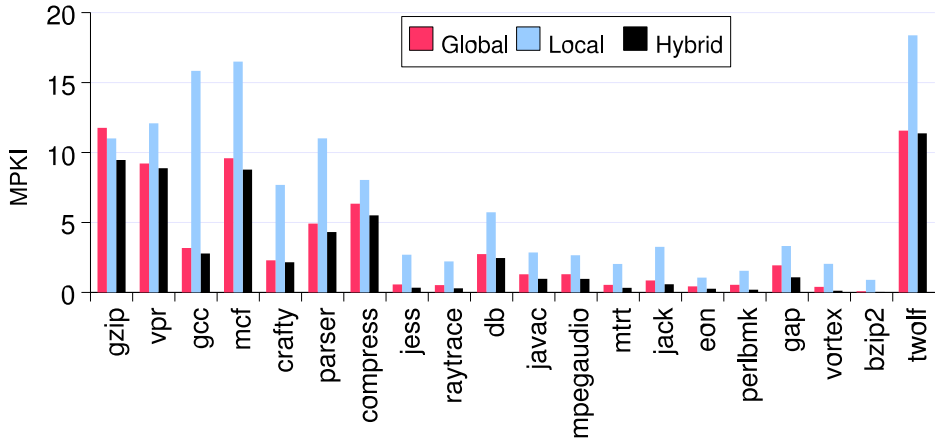

Figure 9: MPKI for the local-PPM, global-PPM, and hybrid predictor.

as zeros) has an RLE starting at zero, while a string of *takens* (shown as ones) is encoded starting at 127. Therefore, the sequence 1 1 is represented as 129 (127+2), and the sequence 0 0 as 2 (0+2). Since each counter can represent a string of length of up to 127, longer patterns require multiple RLE counters. In our example, the pattern of 130 zeros requires two RLE counters with the first counter saturated at 127 and the second counter indicating the remaining 3 zeros in the sequence.

Figure 9 contrasts the accuracy of the local and global PPM predictors on the 20 CBP-2 traces. In all cases except *gzip*, the global PPM predictor is more accurate overall than the local PPM predictor (by an average of 1.8X across all of the traces). However, for certain branches of any given benchmark, local PPM is more accurate. We therefore designed a hybrid predictor that uses a neural network to combine the local and global PPM predictions into a final prediction. This hybrid predictor is the subject of Section 2.3..

2.3. The Neural Network

Typically, tournament (or hybrid) predictors use simple voting schemes to generate the final prediction from the constituent predictors. For example, the Alpha 21264 employs a 4K $\times$ 2-bit table (i.e., a 2-bit saturating counter for each of 4K branches) to track which of two predictors is more accurate for a given branch. Predictions are always made using the more accurate predictor. We experimented with different selection techniques and found that a perceptron-based neural network outperforms traditional approaches such as the 21264's voting scheme. This is because, unlike traditional approaches, a perceptron can learn linearly-separable boolean functions of its inputs.

Figure 10 illustrates the perceptron-based neural network mixer used in our hybrid predictor. The output of the perceptron is given by $y = w_0 + w_1 P_L + w_2 P_G$. The prediction is *taken* if y is positive and *not taken* otherwise. The inputs P_L and P_G correspond to the predictions from the local and global predictor, respectively, and is -1 if *not taken* and +1 if *taken*. 1×10^6 weights of the form $\{w_0, w_1, w_2\}$ are stored in a table. The lower 20 bits of the branch PC are used to index into the table to select the weights. Training the neural

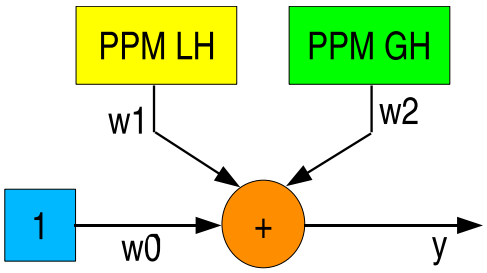

Figure 10: The neural-network mixer.

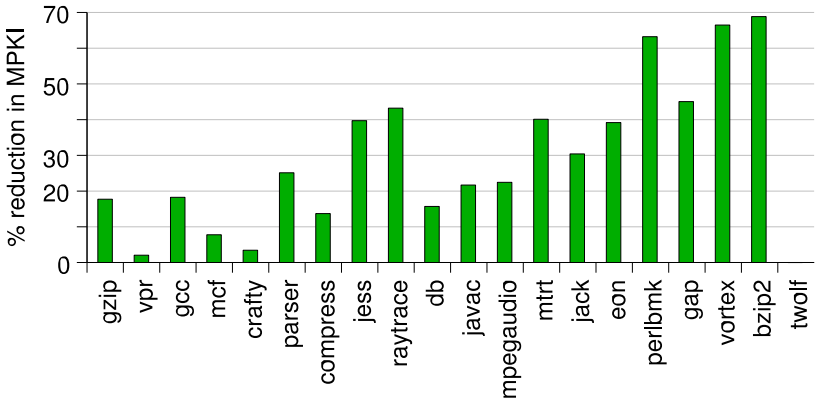

Figure 11: Percent reduction in MPKI when a perceptron instead of voting is used in the selector.

network involves incrementing those weights whose inputs match the branch outcome and decrementing those with a mismatch [3].

Figure 11 shows the percent reduction in MPKI by using a perceptron mixer instead of a traditional voting scheme. The average reduction in MPKI is 14% across all of the CBP-2 traces and is as high as 66% in *vortex* and *bzip*. *twolf* is the only application that shows no improvement.

2.4. Comparison to More Realistic PPM Schemes

The hybrid PPM predictor proposed in this work uses more on-chip memory to store the patterns than is available on current CPUs. This extra storage leads our predictor closer to the upper limit on achievable prediction accuracy. We now compare the prediction accuracy of our predictor against that of a more implementable PPM predictor. For this "realistic" predictor we use the PPM predictor from CBP-1 [5], which uses purely global history and accommodates all of the state information in 64 Kbits. This PPM predictor was ranked 5th in the contest and had only a 7% higher MPKI than the best predictor overall. Figure 12 shows the percentage reduction in MPKI obtained by our PPM predictor relative to the best PPM predictor in the CBP-1 contest. It is surprising to note that the average improvement possible with the idealistic PPM predictor is only 30%. Though applications like *raytrace*,

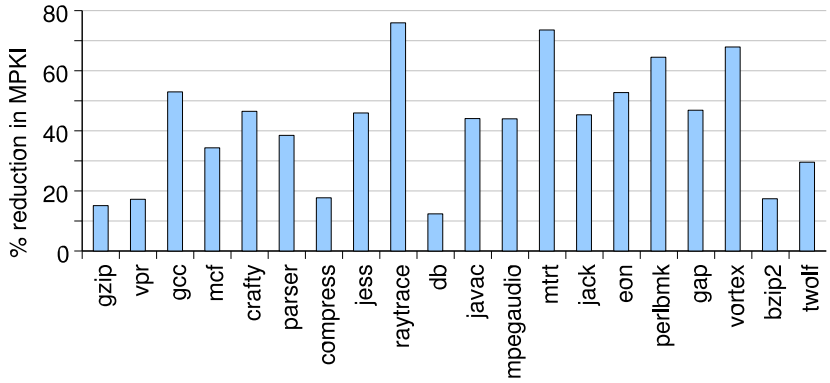

Figure 12: Percent reduction in MPKI for the idealistic scheme over the a realistic PPM predictor.

mtrt, *perlbmk* and *vortex* present a greater opportunity for improvement, these applications generally exhibit small absolute MPKIs.

3. Conclusion

In this paper we presented a branch prediction scheme for the "idealistic" track of the CBP-2 contest. The predictor is based on PPM, a popular algorithm used in data compression. The three main components of our predictor are (1) a local history PPM predictor, (2) a global history PPM predictor, and (3) a neural network. We present many heuristics that help improve the prediction accuracy and simulation time. From the way that these heuristics decrease the number of mispredictions we have gained some interesting insights about branch prediction. These insights are summarized below.

First, it is well known that branch outcomes are highly correlated to global branch history. A fundamental assumption made in many PPM-like (or Markovian) branch-prediction schemes is that identical patterns of global history imply the same static branch and therefore a high likelihood that the prediction will be accurate. Our results, in contrast, suggest not only that identical history patterns often correspond to different branches but also that these identical history patterns often lead to different predictions. By qualifying each pattern in the history with the PC of the associated branch we are able to disambiguate conflicting patterns and reduce *vortex*'s MPKI by 18%, for example.

Our second observation is that the same history pattern at the same branch PC can result in different branch outcomes during different stages of the program's execution. This strongly suggests that branch-prediction techniques need to monitor a pattern's changing bias towards *taken* or *not taken* and predict accordingly. The challenge is in selecting an appropriate sensitivity to changes in bias: excessively rapid adaptivity causes the predictor to be misled by short bursts of atypical behavior; excessively slow adaptivity delays the predictor's identification of a phase change. We found that measuring bias using a 4-bit saturating counter delivers the best prediction accuracy for the CBP-2 traces.

Finally, most branch-prediction schemes use a fixed-length shift register to encode history. In benchmarks such as *raytrace* useful history is often displaced by long loops. The lesson to be learned is that branch predictors need to treat repeated loop iterations as a single entity to preserve more useful data in the history buffer. By run-length encoding the history we reduced *raytrace*'s MPKI by 57%, for example.

Our proposed predictor achieves an average MPKI of 3.00 across the 20 traces provided as part of CBP-2. This represents a 2.1X improvement over the baseline *gshare* predictor distributed with the CBP-2 simulation infrastructure.

4. Acknowledgments

This work is supported by the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC52-06NA25396 with Los Alamos National Security, LLC.

References

[1] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[2] Daniel A. Jiménez et al. The 2nd JILP championship branch prediction competition (CBP-2) call for predictors. http://camino.rutgers.edu/cbp2/.

[3] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[4] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Laboratory, June 1993. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf.

[5] Pierre Michaud. A PPM-like, tag-based branch predictor. In *Proceedings of the First Workshop on Championship Branch Prediction (in conjunction with MICRO-37)*, December 2004.

[6] Arash Partow. General purpose hash function algorithms. http://www.partow.net/programming/hashfunctions/.

[7] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, New York, NY, USA, 2003. ACM Press.