The Second Journal of Instruction-Level Parallelism Championship Branch Prediction Competition (CBP-2)

December 9, 2006
Orlando, Florida, U.S.A.

In conjunction with
The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)

Sponsored by
Intel MRL
IEEE TC-uARCH

Steering Committee:

Dan Connors, Univ. of Colorado
Tom Conte, NCSU
Phil Emma, IBM
Konrad Lai, Intel
Yale Patt, Univ. of Texas
Jared Stark, Intel
Mateo Valero, UPC
Chris Wilkerson, Intel

Program Committee:

Chair: **Daniel A. Jiménez**, Rutgers
Veerle Desmet, Ghent University
Phil Emma, IBM
Ayose Falcón, HP Labs
Alan Fern, Oregon State
David Kaeli, Northeastern
Gabriel Loh, Georgia Tech
Soner Önder, Michigan Tech
Alex Ramirez, UPC and BSC
Oliverio Santana, ULPGC
André Seznec, IRISA
Jared Stark, Intel
Chris Wilkerson, Intel
Huiyang Zhou, University of Central Florida

CBP-2 Foreword

The Second Journal of Instruction-Level Parallelism Championship Branch Prediction Competition (CBP-2) brings together researchers in a competition to evaluate the best ideas in branch prediction. Following on the success of the first CBP held at MICRO in 2004, CBP-2 presents a strong program with contestants from diverse backgrounds.

This second CBP divide the competition into two tracks: *idealistic* branch predictors and *realistic* branch predictors. Idealistic branch predictors have only accuracy as a goal; feasibility is not taken into account. Realistic predictors attempt to balance implementability with accuracy.

Contestants submitted proposed branch predictors in the form of simulator code and papers explaining the ideas. The predictors were evaluated by simulating them in a common framework to determine their accuracy. Idealistic entries were selected as finalists solely based on their accuracy on an undistributed set of traces while realistic predictors were selected based on their accuracy as well as their feasibility as determined by the program committee.

In the end, seven entries were selected to compete as finalists out of a total of 13 submissions. There were four realistic and three idealistic predictors selected. One idealistic and one realistic predictor (Gao and Zhou) were combined into a single entry into both tracks, giving a total of six finalists.

Each entry was reviewed by at least three program committee, with an average of 4.1 reviews for realistic and 3.0 for idealistic entries. For the realistic track, reviewers were asked to evaluate each entry in terms of its feasibility as a component of a future microarchitecture as well as in terms of its accuracy. For the idealistic track, reviews focused on the presentation of the material since accuracy was the only concern. The finalists were selected during an email-based PC meeting held on November 13. One winner will be announced from each track at the workshop in Orlando.

I would like to thank the program committee, the steering committee, and above all, the authors who put forth considerable effort in coding and writing up their branch predictors.

Daniel A. Jiménez, Chair

Advance Program

- **Opening Remarks (8:30am)**

- **Session 1: Realistic Track (8:40am)**

  - **A 256 Kbits L-TAGE Branch Predictor**
    André Seznec, *(IRISA/INRIA/HiPEAC)*
  - **Path Traced Perceptron Branch Predictor Using Local History for Weight Selection**
    Yasuyuki Ninomiya *(Department of Computer Science The University of Electro-Communications, Tokyo, Japan)*
    Kôki Abe *(Department of Computer Science The University of Electro-Communications, Tokyo, Japan)*
  - **Fused Two-Level Branch Prediction with Ahead Calculation**
    Yasuo Ishii *(NEC Corporation, Computer Division)*

- **Selection of the Winner of the Realistic Track (9:55am)**

- **Coffee Break (10:00am)**

- **Session 2: Idealistic Track (10:30am)**

  - **PMPM: Prediction by Combining Multiple Partial Matches**
    (This finalist competes in both tracks)
    Hongliang Gao *(School of Electrical Engineering and Computer Science, University of Central Florida)*
    Huiyang Zhou *(School of Electrical Engineering and Computer Science, University of Central Florida)*
  - **Looking for Limits in Branch Prediction with the GTL Predictor**
    André Seznec *(IRISA/INRIA/HiPEAC)*
  - **Neuro-PPM Branch Prediction**
    Ram Srinivasan *(CCS-3 Modeling, Algorithms and Informatics, Los Alamos National Laboratory and Klipsch School of Electrical and Computer Engineering, New Mexico State University)*
    Eitan Frachtenberg *(CCS-3 Modeling, Algorithms and Informatics, Los Alamos National Laboratory)*
    Olaf Lubeck *(CCS-3 Modeling, Algorithms and Informatics, Los Alamos National Laboratory)*
    Scott Pakin *(CCS-3 Modeling, Algorithms and Informatics, Los Alamos National Laboratory)*
    Jeanine Cook *(Klipsch School of Electrical and Computer Engineering, New Mexico State University)*

- **Selection of the Winner of the Idealistic Track and Future Directions (11:45am)**

A 256 Kbits L-TAGE branch predictor *

André Seznec
IRISA/INRIA/HIPEAC

Abstract

The TAGE predictor, TAgged GEometric length predictor, was introduced in [10].

TAGE relies on several predictor tables indexed through independent functions of the global branch/path history and the branch address. The TAGE predictor uses (partially) tagged components as the PPM-like predictor [5]. It relies on (partial) match as the prediction computation function. TAGE also uses GEometric history length as the O-GEHL predictor [6], i.e. , the set of used global history lengths forms a geometric series, i.e., $L(j) = \alpha^{j-1}L(1)$*. This allows to efficiently capture correlation on recent branch outcomes as well as on very old branches.*

For the realistic track of CBP-2, we present a L-TAGE predictor consisting of a 13-component TAGE predictor combined with a 256-entry loop predictor. This predictor achieves 3.314 misp/KI on the set of distributed traces.

Presentation outline

We first recall the TAGE predictor principles [10] and its main characteristics. Then, we describe the L-TAGE configuration submitted to CBP-2 combining a loop predictor and a TAGE predictor. Section 3 discusses implementation issues on the L-TAGE predictor. Section 4 presents simulation results for the submitted L-TAGE predictor and a few other TAGE predictor configurations. Section 5 briefly reviews the related works that had major influences in the L-TAGE predictor proposition and discusses a few tradeoffs that might influence the choice of a TAGE configuration for an effective implementation.

1. The TAGE conditional branch predictor

The TAGE predictor is derived from Michaud's PPM-like tag-based branch predictor [5] and uses geometric history lengths [6]. Figure 1 illustrates a TAGE predictor. The TAGE predictor features a base predictor T0 in charge of providing a basic prediction and a set of (partially) tagged predictor components Ti. These tagged predictor components Ti, $1 \leq i \leq M$ are indexed using different history lengths that form a geometric series, i.e, $L(i) = (int)(\alpha^{i-1} * L(1) + 0.5)$.

Throughout this paper, the base predictor will be a simple PC-indexed 2-bit counter bimodal table; in order to save storage space, the hysteresis bit is shared among several counters as in [7].

An entry in a tagged component consists in a signed counter *ctr* which sign provides the prediction, a (partial) tag and an unsigned useful counter *u*. Throughout this paper, *u* is a 2-bit counter and *ctr* is a 3-bit counter.

A few definitions and notations The *provider component* is the matching component with the longest history. The alternate prediction *altpred* is the prediction that would have occurred if there had been a miss on the provider component.

If there is no hitting component then *altpred* is the default prediction.

1.1. Prediction computation

At prediction time, the base predictor and the tagged components are accessed simultaneously. The base predictor provides a default prediction. The tagged components provide a prediction only on a tag match.

In the general case, the overall prediction is provided by the hitting tagged predictor component that uses the longest history, or in case of no matching tagged predictor component, the default prediction is used.

However, we found that, on several applications, using the alternate prediction for newly allocated entries is more efficient. Our experiments showed this property is essentially global to the application and can be dynamically monitored through a single 4-bit counter (*USE_ALT_ON_NA* in the simulator). On the predictor an entry is classified as "newly allocated" if its prediction counter is weak.

Therefore the prediction computation algorithm is as follows:

1. Find the matching component with the longest history
2. if (the prediction counter is not weak or USE_ALT_ON_NA is negative) then the predic-

* This work was partially supported by an Intel research grant, an Intel research equipment donation and by the European Commission in the context of the SARC integrated project #27648 (FP6).

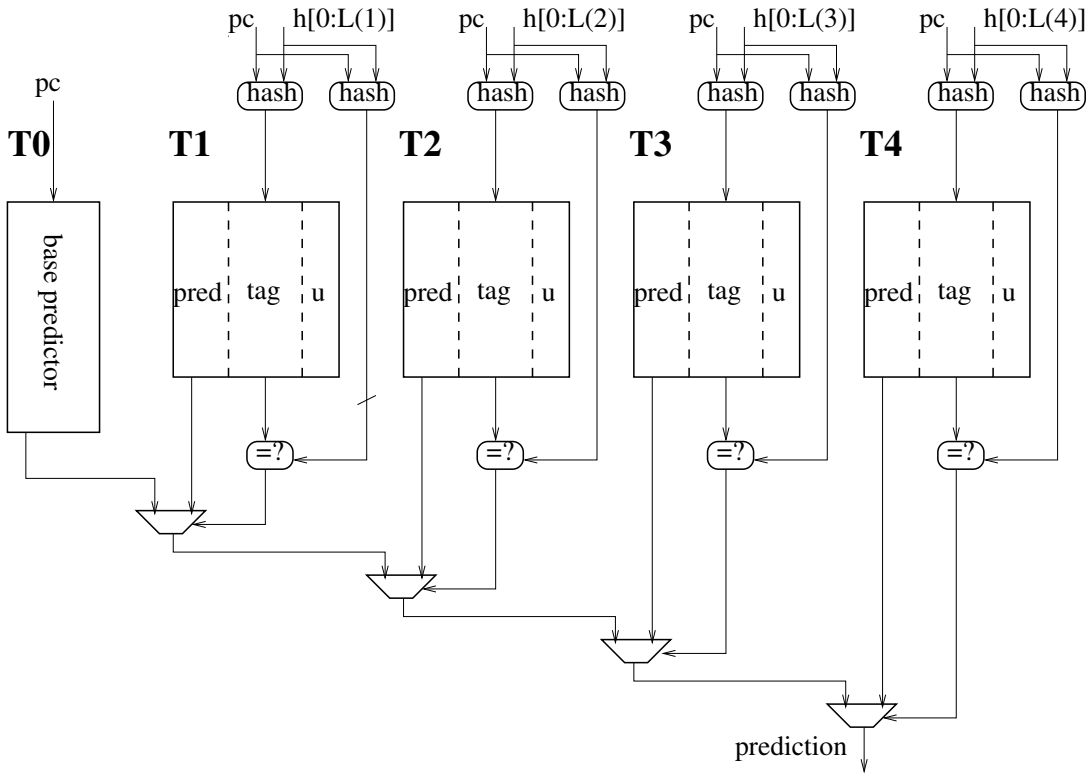

Figure 1. A 5-component TAGE predictor logical synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths. On an effective implementation, predictor selection would be performed thrrough a tree of multiplexors

tion counter sign provides the prediction else the prediction is the alternate prediction

1.2. Updating the TAGE predictor

Updating the useful counter u The useful counter *u* of the provider component is updated when the alternate prediction *altpred* is different from the final prediction *pred*.

The useful *u* counter is also used as an age counter and is gracefully reset as described below. We use an aging algorithm resetting alternatively the bits of the *u* counters.

As a small improvement on the updating presented in [10], after the reset, we flip the signification of the bits u0 and u1 of the useful counter till the next reset:

- reset No 2n-1: u1=0; until reset No 2n: u= 2u1+u0
- reset No 2n: u0 =0; until reset No 2n+1: u= 2u0+u1
- reset No 2n+1: u1=0; until reset No 2n+2: u= 2u1+u0

The period used in the presented predictor for this alternate resetting is 512K branches.

Updating the prediction counters The prediction counter of the provider component is updated. When the useful counter of the provider component is null, the alternate prediction is also updated.

Allocating tagged entries on mispredictions On mispredictions at most one entry is allocated.

If the provider component Ti is not the component using the longest history (i.e., $i \leq M$), we try to allocate an entry on a predictor component Tk with $i < k \leq M$

The allocation process is described below.

The M-i u_j counters are read from predictor components Tj, $i < j \leq M$. Then we apply the following rules.

(A) Avoiding ping-pong phenomenon: in the presented predictor, the search for a free entry begins on table Tb, with b=i+1 with probability 1/2, b=i+2, with probability 1/4 and b=i+3 with probability 1/4.

The pseudo-random generator used in the presented predictor is a simple 2-bit counter.

(B) Initializing the allocated entry: An allocated entry is initialized with the prediction counter set to weak correct. Counter u is initialized to 0 (i.e., *strong not useful*).

2. Characteristics of the submitted L-TAGE predictor

2.1. Information used for indexing the branch predictor

2.1.1. Path and branch history The predictor components are indexed using a hash function of the program counter, the global branch history *ghist* (including non-conditional branches as in [6]) and a (limited) 16-bit path history *phist* consisting of 1 address bit per branch.

2.1.2. Discriminating kernel and user branchs Kernel and user codes appear in the traces. In practice in the traces, we were able to discrimate user code from kernal through the address range. In order to avoid history pollution by kernel code, we use two sets of histories: the user history is updated only on user branches, kernel history is updated on all branches.

2.2. Tag width tradeoff

Using a large tag width leads to waste part of the storage while using a too small tag width leads to false tag match detections. Experiments showed that one can use narrower tags on the tables with smaller history lengths.

2.3. Number of the TAGE predictor components

For a 256 Kbits predictor, the best accuracy we found is achieved by a 13 components TAGE predictor.

2.4. The submitted L-TAGE predictor

2.4.1. The loop predictor component The loop predictor simply tries to identify regular loops with constant number of iterations.

The loop predictor provides the global prediction when the loop has successively been executed 3 times with the same number of iterations. The loop predictor used in the submission features 256 entries and is 4-way associative.

Each entry consists of a past iteration count on 14 bits, a current iteration count on 14 bits, a partial tag on 14 bits, a confidence counter on 2 bits and an age counter on 8 bits, i.e. 52 bits per entry. The loop predictor storage is therefore 13 Kbits.

Replacement policy is based on the age. An entry can be replaced only if its age counter is null. On allocation, age is first set to 255. Age is decremented whenever the entry was a possible replacement target and incremented when the entry is used and has provided a valid prediction. Age is reset to zero whenever the branch is determined as not being a regular loop.

2.4.2. The TAGE predictor component The TAGE predictor features 12 tagged components and a base bimodal predictor. Hysteresis bits are shared on the base predictor. Each entry in predictor table Ti features a Wi bits wide tag, a 3-bit prediction counter and a 2-bit useful counter.

The submitted predictor uses 4 as its minimum history length and 640 as its maximum history length.

The characteristics of the TAGE component are summarized in Table 1. The TAGE predictor features a total of 241.5 Kbits of prediction storage.

2.4.3. Total predictor storage budget Apart the prediction table storage, the predictor uses two 640 bits global history vectors, two 16 bits path history vectors, a 4 bits USE_ALT_ON_NA counter, a 19 bits counter for gracefully resetting the u counters, a 2-bit counter as pseudo-random generator and a 7-bit counter WITHLOOP to determine the usefulness of the loop predictor. That is an extra storage of 1344 bits.

Therefore the predictor uses a total of (241.5+13)*1024 + 1344 = 261,952 storage bits.

3. Implementation issues

3.1. The prediction response time

Since the loop predictor features a small number of entries, the response time of the submitted predictor is dominated by the TAGE response time.

The prediction response time on most global history predictors involves three components: the index computation, the predictor table read and the prediction computation logic.

It was shown in [6] that very simple indexing functions using a single stage of 3-entry exclusive-OR gates can be used for indexing the predictor components without significantly impairing the prediction accuracy. In the simulation results presented in this paper, full hash functions were used. However experiments using the 3-entry exclusive-OR indexing functions described in [6] showed a very similar total misprediction numbers (+0.03 misp/KI).

The predictor table read delay depends on the size of tables. On the TAGE predictor, the (partial) tags are needed for the prediction computation. The tag computation may span during the index computation and table read without impacting the overall prediction computation time. Complex hash functions may then be implemented.

The last stage in the prediction computation on the TAGE predictor consists in the tag match followed by the

	Base	T1,T2	T3,T4	T5	T6	T7	T8,T9	T10	T11	T12
history length		4,6	10,16	25	40	64	101,160	254	403	640
Nb entries	16K pred. 4K hyst.	1K	2K	2K	2K	1K	1K	1K	0.5K	0.5K
Tag width		7	8	9	10	11	12	13	14	15
storage budget (bits)	20K	12K	26K	28K	30K	16K	17K	18K	9.5K	10K

Table 1. Characteristics of the TAGE predictor components

prediction selection. The tag match computations are performed in parallel on the tags flowing out from the tagged components.

Therefore, on an aggressively pipelined processor, the response time of the L-TAGE predictor is unlikely to be a single cycle, but may be close to three cycles.

Therefore if the submitted L-TAGE predictor was to be implemented directly, it should be used as an overriding predictor associated with a fast predictor (e.g. a bimodal table).

3.2. How to address TAGE predictor response time: ahead pipelining

In order to provide the prediction in time for next instruction block address generation, ahead pipelining was proposed in [9] and detailed in [8] for global history/path branch predictors. Therefore the access to the TAGE predictor can be ahead pipelined using the same principle as described for the OGEHL predictor [6] and illustrated on Figure 2.

The prediction tables are read using the X-branch ahead program counter, the X-branch ahead global history and the X-branch ahead path history. On each of the tables, 2^{X-1} adjacents entries are read. 2^{X-1} possible predictions are computed in parallel and information on the last X-1 branchs (1 bit per intermediate branch) is used to select the final prediction. Ahead pipelining induces some loss of prediction accuracy on medium size predictors, mostly due to aliasing on the base predictor. It is also necessary to checkpoint 2^{X-1} predictions to be able to resume without delay on a branch misprediction [9].

For transitions from user mode to kernel mode and viceversa, we make the following hypothesis: 1) The first two branches after a trap or an exception are predicted in a special way using 1-block ahead information for the first one and 2-block ahead information for the second one. 2) The first two branches after the return in the user code are predicted with the ahead information available before the exception or the trap.

3.3. Other implementation issues

3.3.1. Number of predictor components The complexity of a design, its silicon area and its power consumption increase with the number of components. For a 256 Kbits predictor, the best accuracy we found is achieved by a 14-component L-TAGE predictor. However, the TAGE predictor is also quite efficient with a more limited number of components [10].

3.3.2. Predictor update implementation issues The predictor update is performed after commit. Therefore the update logic is not on the critical path.

On a correct prediction, at most two prediction counters *ctr* and the useful counter *u* of the matching component must be updated, i.e., at most two predictor components are accessed.

On a misprediction, a new entry is allocated on a tagged component. Therefore, a prediction can potentially induce up to three accesses to the predictor on a misprediction, i.e, read of all predictor tables at prediction time, read of all predictor tables at commit time and write of (at most) two predictor tables at update time. However, the read at commit time can be avoided: a few bits of information available at prediction time (the numbers of the provider component,the alternate component, *ctr* and *u* values for these two components and the nullity of all the *u* counters) can be checkpointed.

The predictor can therefore be implemented using dual-ported predictor components. However, most updates on correct predictions concern already saturated counters and can be avoided through checkpointing the information *saturated ctr* and *saturated u*. Using 2 or 4-bank structure for the predictor tables (as on EV8 predictor [7]) is a cost-effective alternative to the use of dual-ported predictor tables.

3.3.3. Speculative management

Simple speculative history management On predictor relying only on global history and/or path history such as TAGE, the speculative management of histories can be implemented through circular buffers [3]. Restoring the branch history (respectively the path history) consists of restoring the head pointer.

But complex iteration count management In a deeply pipelined processor, the effectivity of a loop predictor depends on an accurate management of the speculative iteration counts since several iterations of the same loop can

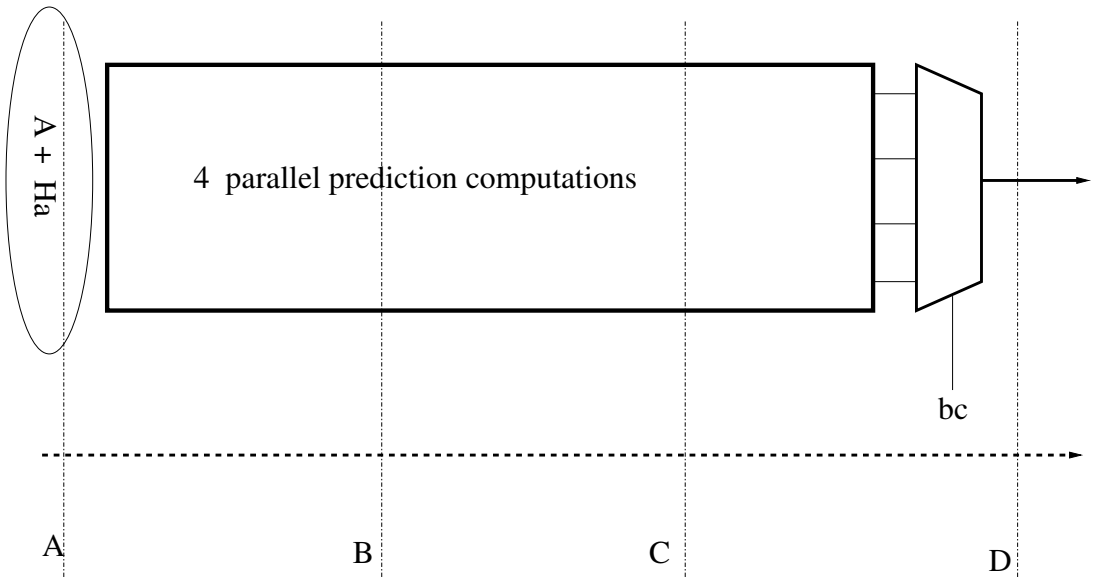

Figure 2. Principle of 3-block ahead branch prediction: information on branch A is used to predict the output of branch C; information on block B and C is used to select the final prediction

be inflight at the same time. A loop predictor is implemented used on the Pentium-M. Therefore this complexity should be considered as manageable.

4. Predictor accuracy

Results per application on the distributed set of traces are displayed in Table 2 for the submitted L-TAGE predictor.

In order to illustrate the potential of the TAGE predictor, we also present simulation results for TAGE predictors featuring simpler hardware complexity: the included 241,5 Kbits TAGE component, a 256 Kbits 13-component TAGE predictor (13C) and a 256 Kbits 8-component TAGE predictor (8C). Finally, as the TAGE predictor can deliver prediction in time provided that ahead pipelining is used, we also illustrate simulations results for a 3-branch ahead 256 Kbits TAGE predictor (8C-Ahead).

The average accuracy of the submitted predictor is **3.314 misp/KI** on the distributed set of traces. When the loop predictor is turned off, the 241,5 Kbits TAGE component achieves 3.368 misp/KI. When the total of the 256 Kbits are affected to the 13 components of TAGE, 3.357 misp/KI is achieved. A 256 Kbits 8-component TAGE predictor achieves 3.446 misp/KI while a 256 Kbits 3-branch ahead TAGE predictor achieves 3.552 misp/KI.

It can be noted that the benefit of the loop predictor is essentially marginal apart on 164.gzip. Simulations on other sets of traces confirm that, only very rare applications effectively benefit from the loop predictor, therefore associating the loop predictor with TAGE is probably not worth the complexity for a real implementation.

Using a medium number of components (8) in TAGE instead of the best number of components (13) impacts the accuracy only slightly: for the final designer the choice of the number of components will be a tradeoff between the extra complexity induced by using more predictor tables and a small accuracy loss. Finally, ahead pipelining does not impair very significantly the predictor accuracy and can therefore be considered for delivering the prediction in time.

5. Conclusion

The use of multiple global history lengths in a single branch predictor was initially introduced in [4], then it was refined by Evers et al. [2] and further appeared in many proposals. Using tagged predictors was suggested for the PPM predictor from Chen et al.[1]. A first PPM-like implementable version was proposed in [5]. TAGE enhances this first proposition by an improved update policy. The TAGE predictor directly inherits the use of geometric history length series from the OGEHL predictor [6], but is more storage-effective. Using only a limited storage, the loop predictor allows to capture some behaviors that are not captured by the TAGE predictor.

The submitted L-TAGE predictor can be directly adapted to hardware implementation as a multi-cycle overriding pre-

	164	175	176	181	186	197	201	202	205	209
L-TAGE	10.074	9.010	3.222	9.049	2.442	5.152	5.712	0.371	0.346	2.339
241.5 Kbits TAGE	10.789	9.015	3.263	9.055	2.445	5.156	5.868	0.372	0.352	2.347
13C	10.781	8.990	3.224	9.006	2.415	5.141	5.853	0.368	0.349	2.345
8C	10.615	9.0.80	3.369	9.446	2.534	5.352	5.914	0.379	0.496	2.368
8C-Ahead	10.854	9.384	3.627	9.688	2.728	5.438	6.024	0.406	0.515	2.439
	213	222	227	228	252	253	254	255	256	300
L-TAGE	1.080	1.068	0.386	0.592	0.219	0.311	1.460	0.139	0.036	13.284
241.5Kbits TAGE	1.121	1.110	0.399	0.594	0.219	0.325	1.464	0.141	0.041	13.288
13C	1.119	1.114	0.397	0.590	0.218	0.325	1.547	0.141	0.041	13.269
8C	1.147	1.146	0.542	0.633	0.240	0.378	1.513	0.145	0.041	13.528
8C-Ahead	1.195	1.188	0.571	0.690	0.246	0.398	1.581	0.169	0.043	13.868

Table 2. Per benchmark accuracy in misp/KI

dictor backing a fast single-cycle predictor (e.g. a bimodal predictor) and achieves very high accuracy.

This configuration was submitted because it achieves very accuracy while it could be implemented in hardware. However for an effective hardware implementation, the hardware complexity of the submitted L-TAGE predictor should be compared with other TAGE-based predictor solutions evaluated in Section 4. These solutions might more cost-effective tradeoffs between hardware complexity and predictor performance; in particular:

- The complexity of a real hardware loop predictor is higher than only reflected by its prediction storage budget; the management of the speculative iteration counts might be a major source of hardware logic complexity. The small accuracy benefit brought by the loop predictor is probably not worth this extra complexity.
- The number of components in the submitted predictor is high: using a smaller number of components, e.g. 8, might be a better design tradeoff.
- The L-TAGE predictor would have a multicycle response time: using a slightly less accurate but ahead pipelined TAGE predictor might allow the design of an overall more efficient instruction fetch front-end [8].

References

[1] I.-C.K. Chen, J.T. Coffey, and T.N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[2] M. Evers, P.Y. Chang, and Y.N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In 23^{rd} *Annual International Symposium on Computer Architecture*, pages 3–11, 1996.

[3] Stephan Jourdan, Tse-Hao Hsing, Jared Stark, and Yale N. Patt. The effects of mispredicted-path execution on branch prediction structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1996.

[4] S. McFarling. Combining branch predictors. TN 36, DEC WRL, June 1993.

[5] Pierre Michaud. A ppm-like, tag-based predictor. *Journal of Instruction Level Parallelism (http://www.jilp.org/vol7)*, April 2005.

[6] A. Seznec. Analysis of the o-gehl branch predictor. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, june 2005.

[7] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeidès. Design tradeoffs for the ev8 branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[8] A. Seznec and A. Fraboulet. Effective ahead pipelining of the instruction addres generator. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[9] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 116–127, 1996.

[10] André Seznec and Pierre Michaud. A case for (partially)-tagged geometric history length predictors. *Journal of Instruction Level Parallelism (http://www.jilp.org/vol7)*, April 2006.

Path Traced Perceptron Branch Predictor Using Local History for Weight Selection

Yasuyuki Ninomiya and Kôki Abe
Department of Computer Science
The University of Electro-Communications
1-5-1 Chofugaoka Chofu-shi
Tokyo 182-8585 Japan
{y-nino, abe}@cacao.cs.uec.ac.jp

Abstract

In this paper, we present a new perceptron branch predictor called Advanced Anti-Aliasing Perceptron Branch Predictor (A^3PBP), which can be reasonably implemented as part of a modern micro-processor. Features of the predictor are twofold: (1) Local history is used as part of the index for weight tables; (2) Execution path history is effectively used. Contrary to global/local perceptron branch predictor where pipelining can not be realized, the scheme enables accumulation of weights to be pipelined, and effectively reduces destructive aliasing, leading to increasing prediction accuracy significantly. Compared with existing perceptron branch predictors such as Piecewise Branch Predictor and Path Trace Branch Predictor, our scheme requires less computational costs than the former and utilizes execution path history more efficiently than the latter. We present a version of A^3PBP where the number of pipeline stages is reduced as much as possible without increasing critical path delay. The resulting predictor realizes a high prediction accuracy. Discussions on implementability of the presented predictor with respect to computational latency, memory requirements, and computational costs are given.

1 Introduction

In recent years, central processing units tend to become more deeply pipelined. The deeper the pipeline becomes, the more the cycle time is reduced, but the more seriously the performance can be degraded due to branch misprediction.

Perceptron branch predictors have extensively been studied these years to reduce the misprediction rate[6]. In this paper, we propose a new perceptron branch predictor called Advanced Anti-Aliasing Perceptron Branch Predictor (A^3PBP), which can be reasonably implemented as part of a modern micro-processor. It is known that the behavior of a branch is well predicted through learning its past behaviors with respect to the execution path history up to the branch[3]. However, there is a possibility that a branch having the same execution path history behaves differently in some cases, and if it occurs the predictors solely based on the path history fails to predict the outcome of the branch. The phenomenon is called a destructive aliasing. In order to resolve the destructive aliasing we propose to use the local history of each branch on the execution path history. The proposed scheme modifies the address of each branch on the execution path history by substituting a part of the address with its local history, and indexes the weight table by the modified execution path history. By doing so, when predicting the behavior of a branch, we can use individual weight table entries on such occurrences that the execution paths are the same but the modified execution paths are different, resulting in resolving the destructive aliasing.

Contrary to the conventional global/local perceptron branch predictor which can not be pipelined, the scheme enables accumulation of weights to be pipelined. Compared with existing perceptron branch predictors such as Piecewise Branch Predictor[5] and Path Trace Branch Predictor[2], our scheme requires less computational costs than the former and utilizes execution history more efficiently than the latter.

In this paper, we first describe recent studies on perceptron branch predictors, focusing on their methods of improvements and problems to be overcome. As a solution to solve the problems A^3PBP is proposed. A version of A^3PBP with reduced pipeline stages without increasing the critical path delay are then described. Lastly, discussions on implementability of the presented predictor with respect to latency, memory requirements, and computational costs are given.

2 Related Work

We refer to the current branch whose outcome is to be predicted as "branch B" hereafter. Idealized Piecewise Linear Branch Predictor[4] uses the address of branch B as well as the path, i.e., a dynamic sequence of branches leading to B, as an index to access weight tables. It uses XOR of the branch B address and the n-th older address within the path as the index to access weights for the n-th global branch history. The scheme reveals high prediction accuracy than Global/Local[6] or Path-Based Neural Branch Predictor[3]. However, using the address of the current branch B as the index to access every weight table does not allow to accumulate the weights before the branch is fetched. The ability of accumulating the weights in advance is essential for reducing the latency by pipelining. Thus Idealized Piecewise Linear Branch Predictor can not be pipelined without exhausting in advance branches which will possibly be fetched.

The problem was tried to be overcome by Ahead Pipelined Piecewise Linear Branch Predictor[5] which uses all possible values of reduced address of the branch B as indices for accessing weight tables in upstream of pipeline structure. However, it requires a large amount of computational cost to exhaust possible cases to occur, even if the number of cases are reduced by limiting the width of the branch B address.

Path Trace Branch Predictor (PTBP)[2] has a pipelined structure with weight tables which are accessed in parallel. A weight table in a stage is accessed by a hashed value of addresses of branches at intervals of pipeline depth within the path leading to the branch B. The scheme is motivated by trying to utilize the path information as much as possible within a pipelined structure. However, learning in PTBP tends too sensitive to path history, resulting in a waste of memory used by weight tables.

3 Advanced Anti-Aliasing Perceptron Branch Predictor

In this section, we propose a new branch predictor, Advanced Anti-Aliasing Perceptron Branch Predictor called A^3PBP, or A3PBP. Figure 1 shows the structure of A^3PBP whose characteristics are explained in the following.

3.1 Utilization of Local History as Part of Index

As described in [3], the path history is naturally reflected to the weights by using the branch address at the most downstream stage (stage 0) as the index to access weight tables at upstream stages of pipeline structure. Here we incorporate the local history of each branch on the execution path to the branch address. This enables to resolve the destructive aliasing without facing the difficulty of pipelining the prediction.

It is expected that using local history improves prediction accuracy as seen from the case of global/local perceptron branch predictor. Implementable perceptron branch predictors such as Path-Based Neural Predictor, however, have not employed local history. It is because the weights for local history of the branch B whose outcome is to be predicted should be accessed and accumulated at the most downstream stage, leading to a large amount of increase in latency. If we want to reduce the latency, the weights for local history need to be accumulated along the pipeline stream. However, the index values for accessing weight tables for local history can not be determined at the upstream of the pipeline if the local history of the current branch B is used as the index.

Path-Based Neural Predictor realizes pipelined branch prediction by accessing weight tables using addresses in the path leading to the branch B. But the scheme has no way of accessing the local history without using the branch B address itself. The need to accumulate weights for local history after obtaining the branch B address does not allow the predictor to be pipelined.

Here we propose a novel way of utilizing local history which allows pipelined prediction. Instead of preparing a dedicated weight table for local history, we reflect the local history information to ordinary weight tables. For that purpose we form the index for accessing the weight tables using a function of the address of branch B and its local history. Figure 2 shows an example of the function, where the index of weight tables is formed by reducing branch address by l bits and concatenating it with l-bit local history read from a table called LHT (Local History Table). Overhead for forming the index is negligible. The index formed in this way is referred to as MA (multiplex address) hereafter.

In our scheme, addresses constituting the path leading to the branch B are replaced with MAs which are used for accessing both weights and bias weight. Indexing weight tables by MAs enables the predictor to learn multiple cases (2^l cases for the example given above) depending on the pattern of local history. In conventional predictors, paths constituting the same addresses with different patterns of local history are not discriminated, resulting in increasing negative collisions of weights.

For implementing the scheme, we need to read bias weight in parallel with reading local history. This can be done by reading a set of possible bias weights in parallel with reading local history, followed by selecting one of the bias weights using a selector as shown in Figure 3. This minimizes the increase in latency. It is worth noting that the address of the branch B is used for accessing the bias

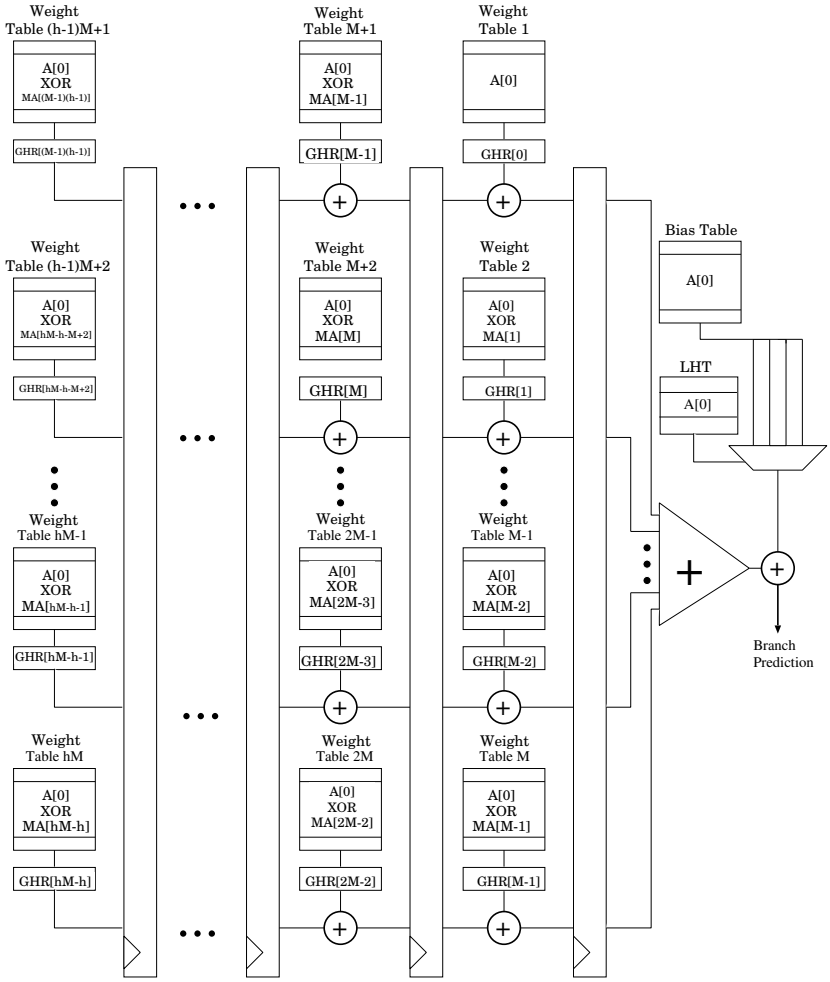

Figure 1. Structure of A^3PBP.

weight table. This implies that the size of bias table is 2^l times larger than that of the other weight tables. It is reasonable because larger bias weight table helps decrease the possibility of negative collisions of weights[7].

3.2 Effective Utilization of Execution Path History

Idealized Piecewise Linear Branch Predictor which uses the path and the address of branch B for accessing weight tables reveals a higher prediction accuracy. However, its pipelined version requires a large amount of cost for computing predictions corresponding to all possible branch B addresses. Reducing the bit length of the branch B address for suppressing the costs as was done by Ahead Pipelined Piecewise Predictor can not fully utilize the information of execution path history.

From the above observations, requirements for the way of indexing weight tables which enables effective utilization of execution path history in pipelined architecture are derived as follows: (1) The index should be closely related to the address of the branch B; (2) The index should be available before the address of the branch B becomes known; (3) The index should have as wide a range of domain as the address of the branch B. From the first and second observations, it is desirable to extract the information of the address of the branch B from the execution path history. Addresses of branches that were executed shortly before the branch B should contain the information of the address of branch B itself as they do in loops. As shown in Figure 1, our archi-

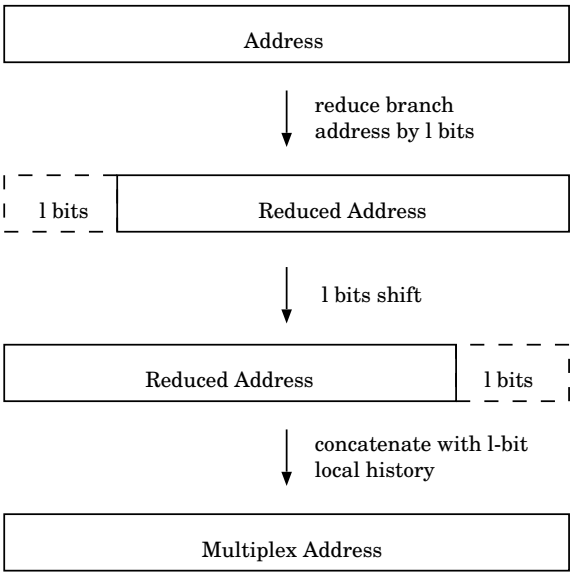

Figure 2. An example of forming the index MA (multiple address) for accessing the weight tables.

tecture uses the address of branch B whose outcome is to be predicted at stage 0 to form the index. The address is denoted by A[0]. For upstream stages of the pipeline, the value of A[0] is regarded as the address of a branch that was executed before the branch whose behavior is going to be predicted at those stages.

The pipeline depth should be shallow because the index for accessing the weights accumulated through pipeline stages should be closely related to the address of branch B. This results in a two-dimensional pipeline structure of h stages. At each stage M accumulations are performed in parallel, hM being the total number of weights to be accumulated which equals to the length of global history. As shown in Figure 1 the index of the weight table at row m and column n of the array structure is given by a hashed value of A[0] and MA[$(n-1)M+m-n$], where $1 \leq m \leq M$ and $1 \leq n \leq h$.

Figure 1 illustrates an example of A^3PBP architecture which employs XOR for the hashing. Thus each weight table in a stage is accessed by an index containing path information that is the one the most possibly related to the branch B available at that stage. (The index of the weight table at row 1 and column 1 is given by A[0] XOR MA[0]. However, MA[0] is not used in the figure. This is because the local history required for composing MA[0] can not be supplied at stage 1 without increasing the latency.)

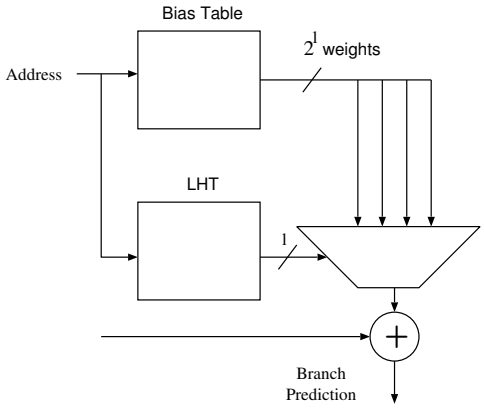

Figure 3. Structure for reading bias weights and local history in parallel.

The usage of path history in our scheme is different from Pipelined PTBP where weight tables in a stage are accessed by hashed values of addresses at intervals of h within the path leading to the branch B. In our scheme the weight tables are accessed by information more closely related to the branch B available at that stage.

The bias weight is multiplied by a constant c before summed with the accumulated weights. This increases the effect of bias weight on prediction. The constant c is set to a power of two and the multiplication is easily realized by bit shifts.

4 Customized Version of A^3PBP for CBP-2

Figure 4 shows a version of A^3PBP customized under the condition of 32KB storage. There are several ways of representing local history of a branch, e.g., using states of an automaton modeling the branch, using its taken/untaken patterns, etc. The customized version uses the states of a bimodal predictor for the local history.

In this version only one pipeline stage (stage 1) precedes the stage 0 where the outcome of the branch B is predicted. That is, h = 1. The length of the global history, hM, was set to 31. The global history is stored in a shift register GHR (global history register).

The size of each weight table is 7-bit 1024 entries except for stage 0 where a composite table consisting of four weight tables of the same size is used. One of the four weights from the composite table is selected by l = 2-bit local history read from LHT of 4096 entries.

The accumulation of weights is performed by a CSA (Carry Save Adder) tree except for addition to obtain final results at stage 0 where a CPA (Carry Propagation Adder)

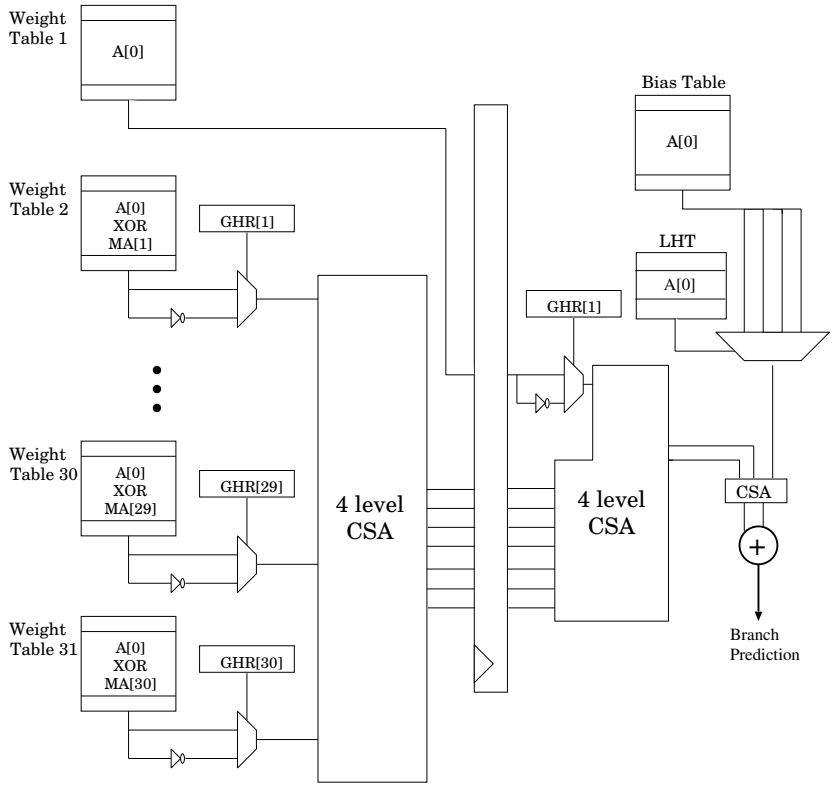

Figure 4. Structure of customized A^3PBP.

is used. The CSA tree of 4 levels at stage 1 reduces 30 addends to 7 intermediate sums which are further reduced to two sums together with another addend by a 4-level CSA tree at stage 0. We replace two's complement subtraction of weights for untaken history of GHR with addition of inverted weights, leaving corrections for two's complement subtraction unapplied, because adding 1 for the collections causes the latency of accumulation to increase. The threshold parameter for learning was experimentally set to 79.

5 Discussions on Hardware Budget Requirements and Latency

We first discuss the latency of prediction by A^3PBP.

The latency for stage 1 is estimated to be less than that for stage 0 from the following discussions. For both stages critical paths contain a weight table access and a selector. A 4-level CSA follows the selector in stage 1 whereas a 1-level CSA and a carry propagation adder (CPA) follow in stage 0. The latency for accessing LHT at stage 0 is larger than that of accessing weight table at stage 1. The latency of a 1-level CSA followed by a CPA is equivalent to that of 4-level CSA in this case of addition, where 12 bit numbers are summed by the CPA. Therefore the critical path of the customized version exists in stage 0.

The critical path in stage 0 passes from the input of LHT to the prediction output through LHT, a 4-to-1 selector, a CSA, and a CPA whose latencies are denoted by $T_{\rm LHT}$, $T_{\rm sel}$, $T_{\rm CSA}$, and $T_{\rm CPA}$, respectively. They are estimated to be $T_{\rm LHT} = 17$, $T_{\rm sel} = 4$, $T_{\rm CSA} = 4$, and $T_{\rm CPA} = 12$ in unit of 2-NAND gate latency. In the estimation we used the known expression $p+\log p+1$ for the latency of a table with 2^p entries[8]. Consequently, the latency of the customized version is estimated to be 37 which is a little larger than 27, an estimated latency of Path-Based Neural Predictor, all in unit of 2-NAND gate latency.

As seen from Figure 4, the structure of our predictor is as simple as that of Path-Based Neural Predictor. So it is reasonable to consider that the complexity, area, and power assumption of our predictor are similar to those of implementable Path-Based Neural Predictor.

The storage requirements of our predictor are calculated as shown in Table 1. The hardware budget of our predictor

Table 1. Storage requirements of our predictor.

Component	Size (bit)
Weight Tables	$7\text{bits} \times 1024 \times 31 = 222208$
Bias Weight Table	$7\text{bits} \times 1024 \times 4 = 28672$
LHT	$2\text{bits} \times 4096 = 8192$
GHR	31
Weight Table Index	$10\text{bits} \times 32 \times 2 = 640$
LHT Index	12
MA	$10\text{bits} \times 31 \times 2 = 620$
Pipeline Registers	$7\text{bits} + 9\text{bits} \times 7 = 70$
Total	260445

is 260445 bits in total that is less than $32\text{KB} + 256\text{bit} = 262400$ bits allowed to use in Realistic Track of the competition.

6 Results

The results of prediction accuracy simulated on the set of distributed benchmarks for CBP-2 competition are reported in a separate table. The misprediction rates are averaged to be 3.999 that was found to be 13% smaller than that of Path-Based Neural Predictor.

7 Conclusions

A novel pipelined perceptron branch predictor was presented. Our scheme uses local history as part of index for weight tables. Another feature of our predictor is to use path history effectively: Weight tables are accessed by the information that is available at that time and is the most possibly related to the branch whose outcome is to be predicted. A version of our proposed predictor which was shown to be reasonably implementable achieves significant improvements on prediction accuracy.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments to improve the quality of the manuscripts. It was partially supported by JSPS Grant-in-Aid for Scientific Research (C)(2) 18500048.

References

[1] M. D. Erecegovac and T. Lang. Digital arithmetic. *Morgan Kaufmann Publishers*, 2004.

[2] Y. Ishii and K. Hiraki. Path trace branch prediction. *IPSJ Trans. Advanced Computing Systems.*, 47(SIG 3(ACS 13)):58–72, 2006.

[3] D. A. Jimenez. Fast path-based neural branch prediction. *Proc. the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, pages 243–252, 2003.

[4] D. A. Jimenez. Idealized piecewise linear branch prediction. *in the First JILP Championship Branch Prediction Competition (CPB-1)*, 2004.

[5] D. A. Jimenez. Piecewise linear branch prediction. *Proc. the 32nd International Synposium on Computer Architecture (ISCA'05)*, pages 382–393, 2005.

[6] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. *Proc. the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, pages 197–206, 2001.

[7] Y. Ninomiya and K. Abe. Path traced perceptron branch predictor using local history for weight selection. *IPSJ SIG Technical Reports*, 2006(88):31–36, 2006.

[8] J. E. Savage. The complexity of computing. *Krieger Publishing Co.*, 1987.

Fused Two-Level Branch Prediction with Ahead Calculation

Yasuo Ishii

NEC Corporation
Computers Division
1-10, Nisshin-cho, Fuchu, Tokyo, Japan
y-ishii@bc.jp.nec.com

Abstract

In this paper, Fused Two-Level (FTL) branch predictor combined with Ahead Calculation method is proposed.

The FTL predictor is derived from fusion hybrid predictor. It achieves high accuracy by adopting PAp-base Geometrical History Length (p-GEHL) prediction which is an effective prediction scheme exploiting local histories. The p-GEHL predictor has several prediction tables indexed from independent functions of the local branch histories and the branch addresses. The prediction is computed through the summation of the values read from the prediction tables. This approach uses limited budget effectively and allows accurate predictions.

The Ahead Calculation is an effective implementation scheme for neural predictors exploiting local histories like the p-GEHL predictor. This scheme is so-called pre-calculation method. The prediction result is computed when the previous branch with the same address was predicted and the result is stored in a RAM which is called Local Prediction Cache (LPC). It reduces the prediction latency since the predictor only have to read the RAM by branch address instead of computing the prediction through adder trees.

FTL branch predictor was optimized for the CBP2 realistic track infrastructure. This optimized-FTL branch predictor with Ahead Calculation achieved 3.466 MPKI with a 262,400 bits budget.

1. Overview

This study focused predictors exploiting local histories. The local history correlates with accurate branch prediction as demonstrated by the fact that three finalists of 1st Champion Ship Branch Prediction Competition (CBP) exploited local histories [6, 9, 13]. In this paper, an effective prediction method with local history and a feasible prediction structure to exploit local history is described.

In Section 2, the principles and features of FTL predictor, a derivative of fusion hybrid branch predictor is introduced. In section 3, an Ahead Calculation method to reduce the latency of complex local predictors is proposed. In section 4, optimization tricks for the 2nd CBP are described. In section 5, the optimized-FTL is covered. The simulation result is shown in section 6, and finally, conclusion is described in section 7.

2. Fused Two-Level (FTL) Branch Prediction

The FTL Branch Prediction is derived from fusion hybrid branch predictor which is already being used for accurate predictions in previous studies [8, 13].

In this section, implementation issue such as prediction latency will not be discussed. These issues will be discussed in the following sections.

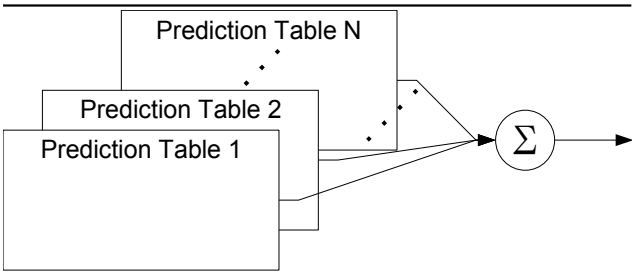

Figure 1:FTL Branch Predictor

Table 1. Two-Level Branch Predictors' Classes.

	Total	**Partial**
Global	GAp[12], Gshare[11], GEHL[2]	Path-Base[5], Piecewise[6, 7]
Local	PAp/PAg[12]	Local Perceptron[4]

2.1. Four Groups of Two-Level Branch Predictor

Overview of the FTL is shown in **Figure 1**. The structure of this predictor is similar to fusion hybrid base predictors. This predictor has many prediction tables each indexed with independent functions. This predictor generates the prediction by making summation of values read from each prediction table. In this scheme, since the prediction result is decided by these results from each table, the indexing function for each sub-predictor is one of the most important factors for accurate prediction. To explore the best indexing function, indexing functions for two-level branch prediction proposed in existing studies were grouped into four groups. These groups are shown in **Table 1**.

In this table, the two lines, **Global** and **Local** mean the predictor exploits either global history or local history. The two columns, **Total** and **Partial** mean whether or not, indexing function involves all newer histories when it exploits old history, the **Partial** means indexing function does not involve newer histories (**Figure 2**). For example, gshare predictor belongs to **Global/Total** since it exploits (1) global history, and (2) H[1:N] when its history length is N. On the other hand, an Nth neuron of simple perceptron predictor is computed by only one bit of global history H[N] which is not the newest history, thus the perceptron predictor is **Global/Partial** predictor.

Each group has both advantage and disadvantage. To cover each disadvantage, FTL employs three groups of predictors

```
0. Boolean Total_Predict (Addr : integer)
1.   sum := 0
2.    for each i
3.      sum := sum + Wi(indexi(Addr, Hist[1:Li]))
4.      /* Each neuron exploits all histories
           which are newer than H[Li] */
5.    end for
6.   return (sum ≧ 0)
7. End Total_Predict

0. Boolean Partial_Predict (Addr : integer)
1.   sum := 0
2.    for each i
3.      sum := sum + Wi(indexi(Addr, Hist[Li:Li+1]))
4.      /* Each neuron does not exploit any
           histories which are newer than H[Li] */
5.    end for
6.   return (sum ≧ 0)
7. End Partial_Predict
```

Figure 2: Examples of Total Prediction and Partial Prediction

```
0. Boolean Predict (A : integer)
1.   sum := 0
2.    for each i
3.      sum := sum + Wi(indexi(Addr, Hist))
4.    end for
5.   return (sum ≧ 0)
6. End Predict

0. Void Update (A : integer, Outcome : boolean)
1.   if ((p!=Outcome) or (|Sum| < θ))
2.    for each i in parallel
3.      if Outcome = Taken then
4.        Wi(index(Addr, Hist)) :=Wi(indexi(Addr, Hist)) + 1
5.      else
6.        Wi(index(Addr, Hist)) :=Wi(indexi(Addr, Hist)) - 1
7.      end if
8.    end for
9.   end if
10.End Update
```

Figure 3:Predicting and Updating Algorithm

from **Global/Total**, **Global/Partial**, and **Local/Total**. In the rest of this section, indexing policy of these three groups is described.

2.1.1.Global/Total predictor

Indexing functions of **Global/Total** group are derived from GEHL predictor since it is one of the most accurate predictor of **Global/Total** predictors. History length for each prediction table is determined by geometrical series such as $L(j) = a^{j-1} \cdot L(1)$. The indexing functions also involve the path information since it has good correlation to branch accuracy.

2.1.2.Global/Partial predictor

Indexing functions of **Global/Partial** group are derived from Gao's method[9] which exploits partial histories effectively. It means that an indexing function generates an address of prediction table by bit vector which is a small part of global histories. The length of the bit vector is one of the parameters. This group also exploits the path information for accurate prediction. This approach also reduces complexity of the predictor, significantly reducing the number of skewing adders compared to simple perceptron predictor.

2.1.3.Local/Total predictor

The 1st CBP finalists did not exploit this group. However, this group has a potential to achieve an effective prediction. The indexing functions of **Local/Total** groups are the same with GEHL predictor except that these indexing functions exploit local histories. We call this approach PAp-base GEHL (p-GEHL) prediction. This prediction approach also reduces the size of adder tree from **Local/Partial** approach since this approach does not have to employ many tables like local perceptron.

2.2.Prediction and Updating Algorithm

The prediction algorithm which was already introduced in 2.1 and updating algorithms are shown in **Figure 3**.

The fusing method is implemented by an adder tree. A prediction result is decided from the sign bit of the summation of values read from prediction tables. This scheme is derived from fusion hybrid predictor [8, 13]. The FTL updating policy is also derived from the fusion hybrid updating policy. The FTL predictor is only updated when miss prediction occurs or when the absolute value of summation is smaller than the threshold θ.

3. Ahead Calculation Method

In this section, an Ahead Calculation method for **Local** predictor to reduce prediction latency is described.

3.1. Implementation Issues of Local Predictor

Modern predictors based on skewing fusion method have a demerit of prediction latency since it is difficult to finish their calculation in sufficiently short time. Especially, it is difficult for **Local** predictor, since the **Local** prediction schemes require two accesses, one for the local history table and the other for pattern history table. The long latency is harmful for the processor performance. To avoid this latency issue, the modern predictors have to employ some tricks such as Ahead Pipelining. However, the Ahead Pipelining method cannot be applied for **Local** predictors because the local history table cannot be read until the predicted branch instruction is decided [14]. To resolve this issue, a new implementation method called Ahead Calculation method is proposed.

3.2. Ahead Calculation Algorithm

In this part, the Ahead Calculation method is explained. This technique reduces the prediction latency of **Local** predictors. The predictor pre-calculates the next prediction result when the predictor makes a prediction. The calculated result for the next prediction is stored in a RAM which is called Local Prediction Cache (LPC). The predictor exploits the stored values when it predicts the branch with the same instruction

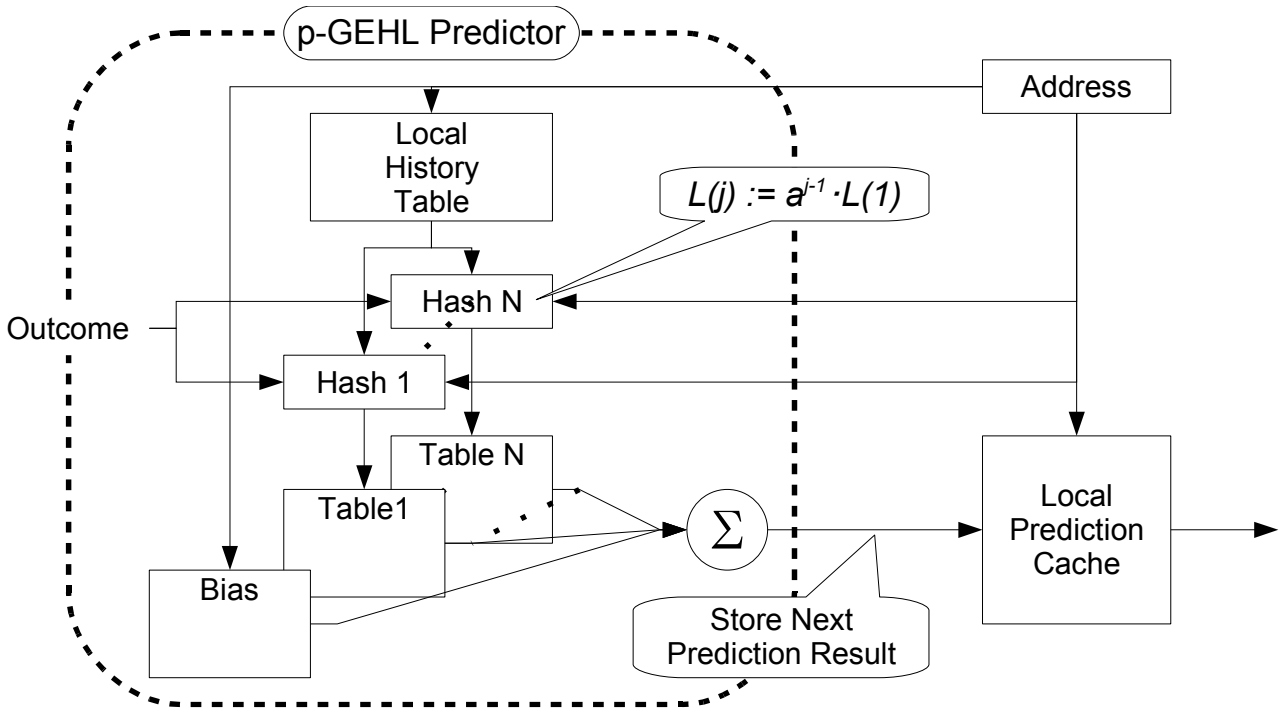

Figure 4:Ahead Calculation

address. The predictor with Ahead Calculation only has to read the LPC when it predicts a target branch. The prediction latency is only the read latency of LPC. This is the same as the simple bias table. This latency is sufficiently implementable. An overview of the Ahead Calculation is shown in **Figure 4**.

3.3. Cost Estimation

External cost for Ahead Calculation is only LPC. In optimized FTL, the LPC has 2048-entry table and each entry is 7 or 8 bits. This implementation does not require any other costs like duplication of computing logic. The LPC, which is a simple RAM, also requires relatively smaller space and lower power consumption than computation logics such as duplicated adder trees. From the above, the LPC is not so expensive for modern processors.

3.4. Read After Write Hazard

The read after write (RAW) hazard causing inaccurate prediction may occur with the Ahead Calculation. However, the effect of this issue is small since the hazard occurs only when the interval of prediction for one branch instruction is within a few cycles. Such tight loops will be unrolled in optimized code or filtered by loop counter. In the rest of this paper, this issue was intentionally ignored.

4. Other Tricks

In this section, several tricks and borrowed techniques for the optimized FTL predictors are described.

4.1. Bias Filtering

FTL employs an 1-bit array to filter strongly biased branches.

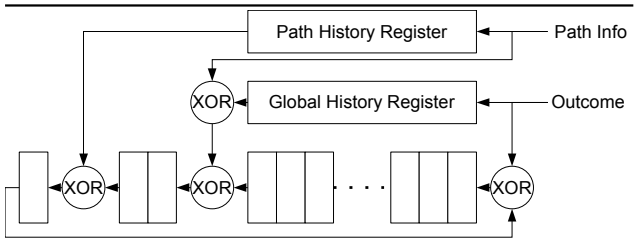

Figure 5:Enhanced Folded Indexing

This table is initialized by '0' and is indexed by the branch address. When a prediction is required, the predictor reads this array with branch address. When the value is '0' then the predictor generates its prediction only by the bias table of the FTL predictor. When the prediction result is wrong, the bias table is required to be updated and the used entry of filtering array is set to '1' except when the read bias value is zero. The predictor exploits other prediction methods only when the value read from the bias filtering array is '1'.

In real processor, each entry of this array should be reset to '0' at appropriate timing. Without reset mechanism, almost all entries of this array will become '1' after some context switches. This array should be reset when the branch target buffer miss occurs. This approach is feasible since almost all modern processors employ set associative structured branch target buffer. However, this resetting method cannot be implemented in CBP infrastructure since it has no branch target buffer instance. Thus, even if the filter does not have any reset mechanisms, this filtering method is feasible on real processors.

4.2. Loop Counter

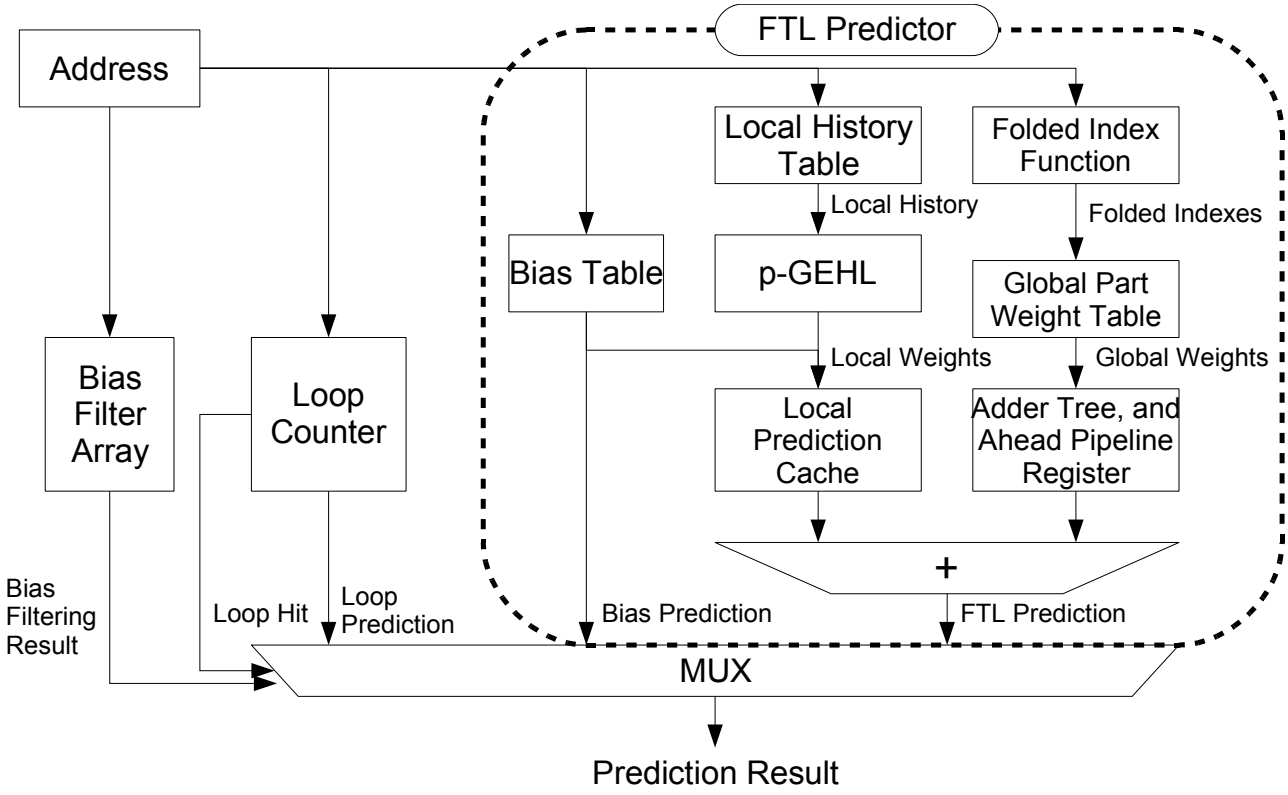

Figure 6:Optimized FTL Branch Predictor

Loop counter employs technique borrowed from Gao's loop counting method [9]. It is set-associative cache structured loop counter. Each entry of this loop counter contains loop count information with confidence value. This loop counter is feasible since it is not as complex as that of set-associative structured branch target buffer. The optimized FTL predictor employs 8-way set-associative loop counter.

4.3. Enhanced Folded Indexing

The FTL exploits long global history length such as GEHL predictor. To achieve complexity-effective indexing circuit for **Global** predictors, an enhanced folded indexing circuit shown in **Figure 5** is proposed. The folded indexing method has been proposed in previous CBP [10]. The enhanced folded indexing circuit is extended to include another pair of 2-bit exclusive-or circuit to fold path information. An implementation cost of the indexing function is shift register and about 10 2-bit exclusive-ors. These circuits can also be used in other predictors, such as GEHL predictor.

4.4. Dynamic Threshold Fitting

Updating threshold θ significantly affects the accuracy of the predictor whose updating policy is derived from perceptron predictor. The best threshold differs among benchmarks. To optimize the threshold value, we employ the dynamic history length fitting. The fitting algorithm is already proposed [2], and it is a cost effective design. We design it as a 14-bit counter.

Table 2: Configuration of Each Predictor

	Systolic	**Simultaneous**	**No Ahead Pipe**
Ahead Pipelining	Systolic-array like hybrid	Simult-reading hybrid	Disable
Ahead Calculation	Enable	Enable	Disable
Bias Filtering	Enable	Enable	Enable
Indexing Functions	Enable	Enable	Enable
Dynamic Thres Fit	Enable	Enable	Enable
Dynamic Adapting	Disable	Enable	Enable
Weight Boosting	Disable	Enable	Enable

4.5. Dynamic Adaptation

The three finalists of previous competition employed dynamic history length adapting. This study also employs similar dynamic adaptation through exploiting some execution information: the hit rate of the bias filter and the loop counter, and the number of static conditional branches.
This trick might not be appropriate for realistic track thus the predictor was also evaluated with this dynamic adaptation disabled.

4.6. Ahead Pipelining for Global Predictor

Table 3: Budget Count for Each Predictor

	Systolic	Simultaneous	No Ahead Pipe
Bias Filter	4096 entry * 1 bit	4096 entry * 1 bit	4096 entry * 1 bit
Bias Table	2048 entry * 6 bit	2048 entry * 6 bit	2048 entry * 6 bit
Prediction Table	17 table 2048 entry * 6 bit	17 table 2048 entry * 6 bit	18 table 2048 entry * 6 bit
LPC	2048 entry * 8 bit	2048 entry * 7 bit	0 bit
Indexing Function	398 bit	760 bit	380 bit
Dynamic Threshold Fitting	14 bit	14 bit	14 bit
Ahead Pipeline Reg	4 duplication * 9 depth * 11 bit	4 duplication * 3 depth * 11 bit	0 bit
Ahead Pipeline Address	9 depth * 11 bit	3 depth * 11 bit	0 bit
Global History	121 bit + 3 * 65 bit	201 bit + 3 * 81bit	201 bit + 3 * 81bit
Local History	1024 entry * 16 bit	1024 entry * 16 bit	1024 entry * 16 bit
Loop Counter	6 entry * 8-way * 58 bit	8 entry * 8-way * 58 bit	16 entry * 8-way * 58 bit
Adaptive Info	0 bit	2 bit	2 bit
Performance Counter	0 bit	160 bit	160 bit
Total Budget Size	262055 bit	261257 bit	262376 bit

The FTL predictor exploits adder tree so complex that the latency of this circuit is not tolerated in real processors. For an implementable latency, the **Local** predictors should exploit Ahead Calculation which is explained in previous section and the **Global** predictor should employ Ahead Pipelining [1].
There are two major policies for implementation of Ahead Pipelining. One is systolic array approach which is used in piecewise linear branch predictor [7] and the other policy is simultaneously reading approach which is used in GEHL predictor [3]. We adopt hybrid approach of these two policies. It means that the predictor reads four values simultaneously from several tables in each step, and computes the intermediate prediction value as systolic array approach. The predictor selects an appropriate intermediate prediction result at the last step of Ahead Pipelining. This Ahead Pipelining policy requires computing circuit duplication. The FTL predictor employs four duplications of computation circuit, but it is still less complex than that of O-GEHL predictor which utilizes 16 duplications[3]. Our predictor requires only 56 adders for its implementation while the Piecewise Linear Branch Predictor requires more than 100 adders[7].

5. Optimized-FTL branch predictor

In this section, an optimized-FTL (o-FTL) is described. This predictor exploits the techniques explained in the previous sections. The overview is shown in **Figure 6**. The o-FTL employs the bias filter and the loop counter for anti-aliasing.
The implementable FTL is divided into two parts (**Global** and **Local**) for effective implementation. The **Global** part is implemented by Ahead Pipelining structure which is explained in previous section. The **Local** part is implemented by Ahead Calculation with LPC. The prediction result of the FTL is computed through the addition of the results of these parts. The prediction of the FTL is filtered by the bias filter and the loop counter. A prediction of the FTL is used only when both filters miss. The FTL predictor is never updated unless all filters miss the prediction.

6. Results

6.1.Predictor Configurations

The predictors were evaluated in three configurations (**Systolic Simultaneous**, and **No Ahead Pipe**), each different in complexity. The feature of each predictor is shown in **Table 2**. The **Systolic** means systolic array-like hybrid Ahead Pipelining. It is the most implementable configuration. The **Simultaneous** adopts simultaneous reading hybrid Ahead Pipelining which is similar to simultaneous pipelining. It adopts some complex structure such as dynamic adapting and weight boosting, but is still implementable. The **No Ahead Pipe** is to show theoretical best care without considering prediction latency.

6.2.Budget Counting

The budget counting is shown in **Table 3**. The LPC entry of **Simultaneous** requires only 7 bits as the LSB is rounded. The total number of bits used for each of the three predictors is less than 262,400 bits.

6.3.Prediction Results

The detailed results for each benchmark are shown in **Table 4**. The overall miss prediction rate is reduced compared to existing predictors including previous CBP's finalists[1]. The Ahead Pipelining FTL predictor increases the miss prediction rate from 0.045 MPKI to 0.091 MPKI. However this prediction accuracy is still better than that of 1st CBP's finalists.
As o-FTL predictor exploits some unrealistic features, such as Dynamic Adaptation and Bias Filtering requiring reset mechanism which o-FTL predictor do not employ, thus impact of disabling these features were evaluated. The result of disabling these features are shown in **Table 5**.

7. Conclusion and Future Works

In this paper, combination of the Fused Two-Level branch prediction and the Ahead Calculation method which is an effective implementation scheme for **Local** predictors like PAp-base GEHL predictor was proposed. The optimized FTL

1 The finalists of CBP-1 have been optimized by the author with new CBP-2 infrastructure. In this optimization, some important tricks such as Gao's dynamic adaptation [9] were disabled to meet CBP-2 requirement.

Table 4: Miss Prediction Rate for Each Traces

	Systolic	Simultaneous	No Ahead Pipe
164.gzip	9.554 MPKI	9.531 MPKI	9.519 MPKI
175.vpr	8.676 MPKI	8.644 MPKI	8.616 MPKI
176.gcc	4.802 MPKI	4.513 MPKI	4.529 MPKI
181.mcf	10.985 MPKI	11.026 MPKI	10.903 MPKI
186.crafty	2.553 MPKI	2.582 MPKI	2.454 MPKI
197.parser	5.793 MPKI	5.790 MPKI	5.784 MPKI
201.compress	5.497 MPKI	5.450 MPKI	5.416 MPKI
202.jess	0.484 MPKI	0.493 MPKI	0.465 MPKI
205.raytrace	0.606 MPKI	0.571 MPKI	0.441 MPKI
209.db	2.395 MPKI	2.378 MPKI	2.333 MPKI
213.javac	1.094 MPKI	1.101 MPKI	1.081 MPKI
222.mpegaudio	1.069 MPKI	1.086 MPKI	1.054 MPKI
227.mtrt	0.747 MPKI	0.681 MPKI	0.524 MPKI
228.jack	0.749 MPKI	0.768 MPKI	0.718 MPKI
252.eon	0.276 MPKI	0.283 MPKI	0.297 MPKI
253.perlbmk	0.333 MPKI	0.319 MPKI	0.307 MPKI
254.gap	1.638 MPKI	1.553 MPKI	1.506 MPKI
255.vortex	0.153 MPKI	0.160 MPKI	0.145 MPKI
256.bzip2	0.046 MPKI	0.045 MPKI	0.046 MPKI
300.twolf	12.798 MPKI	12.349 MPKI	12.292 MPKI
Average	3.512 MPKI	3.466 MPKI	3.421 MPKI

branch predictor achieves accurate prediction in feasible prediction latency for real microprocessor.

However, the FTL and the Ahead Calculation method have several challenges. (1) The FTL exploits about 20 tables for generating its prediction, but this might be too many for modern processors since the number of tables strongly affects the size of adder trees. The impact of the number of tables for branch prediction accuracy needs to be evaluated. (2) The Ahead Calculation method will be inaccurate when the RAW hazard occurs. This impacts needs to be evaluated. (3) The cost of speculative updating of the Ahead Calculation method is not discussed in this paper. The trade-offs for modern processor needs to be discussed. (4) There are rooms for improvements in dynamic adaptation, such as exploiting rich executing information discussed in previous competition [9]. These are our future works.

8. Acknowledgments

This work was supported by members of Kei Hiraki's Laboratory in University of Tokyo. The author appreciates their helpful advice.

9. References

[1] A. Seznec and A. Fraboulet. Effective Ahead Pipelining of the Instruction Address Generation. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[2] A. Seznec. The O-gehl Branch Predictor. In *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[3] A. Seznec. Analysis of the O-GEometric History Length branch predictor. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2004.

Table 5: Impact of Dynamic Adaptation and Bias Filter

	Systolic	Simultaneous	No Ahead Pipe
Disable Bias Filtering	N/A	N/A	3.474 MPKI
Disable Dyn Adaptation	3.512 MPKI	3.514 MPKI	3.473 MPKI
Disable Both Feature	3.542 MPKI	3.542 MPKI	3.490 MPKI

[4] D. A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, 2001.

[5] D. A. Jiménez. Fast Path-based Neural Branch Prediction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2003.

[6] D. A. Jiménez. Idealized Piecewise Linear Branch Prediction. In *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[7] D. A. Jiménez. Piecewise Linear Branch Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2004.

[8] G. Loh. The Frankenpredictor. In *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[9] H. Gao and H. Zhou. Adaptive Information Processing: An Effective Way to Improve Perceptron Predictors. In *The 1st JILP Championship Branch Prediction Competition (CBP- 1)*, 2004.

[10] P. Michaud. A PPM-like, Tag-based Predictor. In *The 1st JILP Championship Branch Prediction Competition (CBP- 1)*, 2004.

[11] S. McFarling. Combining Branch Predictors. In *Tech Rep. TN-36, Digital Western Research Laboratory June 1993.*

[12] T. Yeh and Y. N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *The 19th Annual International Symposium on Computer Architecture pp.124-134, May 19-21, 1992, Gold Coast, Australia.*

[13] V. Desmet, H. Vandierendonck, and K. D. Bosschere. A 2bcgskew Predictor Fused by a Redundant History Skewed Perceptron Predictor. In *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[14] D. Tarjan, K. Skadron, and M. Stan. An Ahead Pipelined Alloyed Perceptron with Single Cycle Access Time. In *the Workshop on Complexity-Effective Design (WCED), June 2006*

PMPM: Prediction by Combining Multiple Partial Matches

Hongliang Gao Huiyang Zhou

School of Electrical Engineering and Computer Science
University of Central Florida
{hgao, zhou}@cs.ucf.edu

***Abstract*—The PPM prediction algorithm has been well known for its high prediction accuracy. Recent proposals of PPM-like predictors confirm its effectiveness on branch prediction. In this paper, we introduce a new branch prediction algorithm, named Prediction by combining Multiple Partial Matches (PMPM). The PMPM algorithm selectively combines multiple matches instead of using the longest match as in PPM. We analyze the PPM and PMPM algorithms and show why PMPM is capable of making more accurate predictions than PPM.**

Based on PMPM, we propose both an idealistic predictor to push the limit of branch prediction accuracy and a realistic predictor for practical implementation. The simulation results show that the proposed PMPM predictors achieve higher prediction accuracy than the existing PPM-like branch predictors such as the TAGE predictor. In addition, we model the effect of ahead pipelining on our implementation and the results show that the accuracy loss is relatively small.

1. INTRODUCTION

Given its importance on high performance microprocessor design, branch prediction has been extensively studied. As analyzed in [1],[9], the prediction by partial matching (PPM) algorithm [2], originally proposed for text compression, can achieve very high prediction accuracy for conditional branches. Recent studies of PPM-like predictors [13], [8] confirm the effectiveness of PPM and show that PPM-like predictors outperform many state-of-art branch predictors.

In this paper, we propose to improve PPM-like branch predictors by combining multiple partial matches rather than the longest partial match as used in the PPM algorithm. We first examine why longest partial match may lead to suboptimal prediction accuracy. We then propose a prediction algorithm to selectively combine multiple partial matches (PMPM) with an adder tree.

Based on the PMPM algorithm, we develop an idealistic predictor which explores extremely long history to push the limit on branch prediction accuracy and a realistic predictor for practical implementation. The realistic design is based on recently developed PPM-like TAGE branch predictors [13] and it has similar hardware complexity compared to the TAGE predictor. Besides exploiting correlation from various global branch histories, our design enables efficient integration of local history information. The experimental results show that the proposed designs can achieve higher accuracy than TAGE predictors. Finally, we implement ahead pipelining to evaluate the impact of the access latency of the proposed PMPM predictor and the results show that with ahead pipelining, our design can provide a prediction every cycle with relatively small accuracy loss.

The rest of this paper is organized as follows. In Section 2 we study the PPM branch prediction algorithm and introduce the PMPM algorithm. An idealistic PMPM predictor is presented in Section 3. Section 4 describes our design of realistic PMPM predictors. Finally, Section 5 concludes the paper.

2. PREDICTION BY COMBINING MULTIPLE PARTIAL MATCHES

2.1. PPM with the (Confident) Longest Match

In the PPM algorithm, the Markov model, which determines the prediction, is updated based on input information. When used for branch prediction, the Markov model keeps track of branch history and uses the longest match to make a prediction. The assumption behind the longest match is that the longer history provides a more accurate context to determine the branch behavior.

However, since branches exhibit non-stationary behavior, partial context matches may not be sufficient to accurately predict branch outcomes. To examine this effect, we implemented a PPM-based branch predictor, in which the global branch history is used as the context for each branch. We assign a signed saturating prediction counter with the range [-4, 4] for each (branch address, history) pair. The prediction counter is incremented if the branch outcome is taken and decremented otherwise. When both of the branch address and history are matched, the corresponding prediction counter is used to make a prediction. When there are multiple history matches for the same branch with different history lengths, the prediction counter associated with the longest history is used.

In order to show that the longest match may not be the best choice for branch prediction, we implemented another scheme, in which the prediction counter with the longest-history match is not always selected to make a prediction. Instead, we use the prediction counter as a confidence measure of the potential prediction. Only when the prediction counter is a non-zero value, it can be selected to make a prediction. We call such a scheme as PPM with the confident longest match. We simulate both schemes, i.e., PPM with the longest match and PPM with the confident longest match, and report the misprediction rate reductions, measured in mispredictions per 1000 instructions (MPKI), achieved by the confident longest match. The results are shown in Figure 1. In this experiment, the maximum history length is set as 40.

From Figure 1, we can see that the confidence-based PPM has lower misprediction rates than the PPM scheme for all the benchmarks except the benchmark *vortex*, which implies that the longest match used in PPM may lead to suboptimal

prediction accuracy.

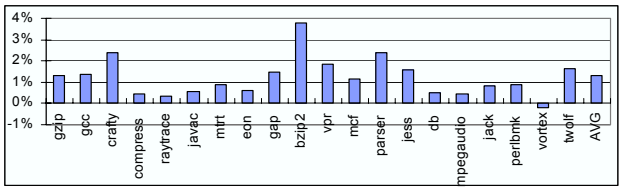

Figure 1. Misprediction rate reductions by the confident longest match in PPM-based branch predictors (Max History Length = 40).

2.2. *Prediction by combining Multiple Partial Matches*

From the experiments with the PPM-based branch predictors, we observed that when there are multiple matches, i.e., matches with various history lengths, in the Markov model, the counters may not agree with each other and different branches may favor different history lengths. Such adaptivity is also reported in [7] and explored in recent works, such as [10]. Based on this observation, we propose to use an adder tree to combine multiple partial matches in a PPM-based predictor (other combination schemes including linear combination have been explored in [3] and the adder tree is selected for its effectiveness and simplicity in implementation) and we call this novel prediction algorithm as Prediction by combining Multiple Partial Matches (PMPM). In a PMPM predictor, we select up to L confident longest matches and sum the counters to make a prediction. Figure 2 shows the average misprediction rates of the proposed PMPM predictors with L varying from 1 to 41 and the original PPM predictor with the longest match. The maximum history length is set as 40.

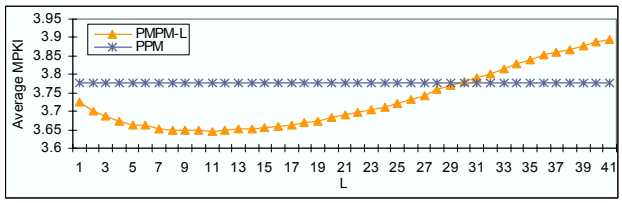

Figure 2. Misprediction rates of PMPM-L predictors and the original PPM predictor

From Figure 2, it can be seen that on average, combining multiple partial matches can provide higher prediction accuracy than utilizing a single partial match. Combining too many partial matches, on the other hand, can be harmful since many low-order Markov chains are included, which are susceptible to noises.

3. IDEALISTIC PMPM PREDICTOR

Similar to PPM predictors, PMPM predictors require extensive history information and the number of different (branch address, history) pairs increase exponentially with the history length. Therefore, modeling an idealistic PMPM predictor to push the limit of branch prediction accuracy is still a challenge. In this competition, we developed an idealistic PMPM predictor which explores extremely long global history (651) information with acceptable requirements on memory storage and simulation time.

3.1. *Predictor Structure*

The overall predictor structure is shown in Figure 3. We use a per-branch tracking table (PBTT) to record some basic information of each branch. PBTT is a 32k-set 4-way cache structure. Each PBTT entry includes the branch address, LRU bits for replacement, a branch tag, a 32-bit local history register, a meta counter used to select either GHR-based or LHR-based predictions, a bias counter, and a simple (bias) branch predictor. A unique tag is assigned to each static branch and it is used to replace the branch address in index and tag hashing functions. Since there are a large number of highly biased branches, we use a simple branch predictor to filter them out. The simple branch predictor detects fully biased (always taken or always not taken) branches. A branch only accesses the main PMPM predictor when it is not fully biased. For remaining branches, PBTT sends the branch tag, LHR, meta counter, and bias counter to the PMPM predictor.

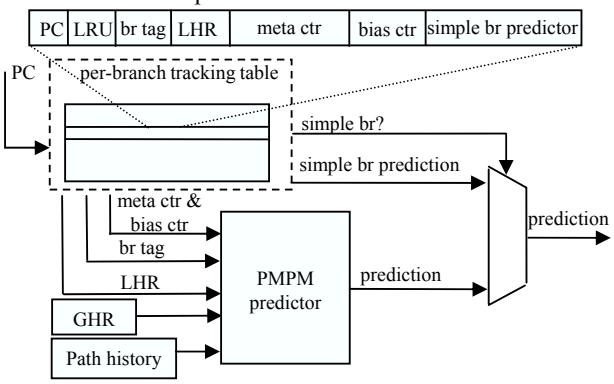

Figure 3. The overall idealistic PMPM branch prediction scheme.

The PMPM predictor has three prediction tables, as shown in Figure 4. A short-GHR table is used to store information corresponding to the most recent 32-bit global history. A long-GHR table is used to store information corresponding to longer global histories. We also use a LHR table to make local-history-based predictions and the LHR table uses 32-bit local histories. Those three prediction tables are 4-way set-associative structures and each entry has a tag, an LRU field, a prediction counter, a usefulness counter, and a benefit counter. In order to capture very long global histories, we use geometric history lengths [10] in the long GHR table.

3.2. *Prediction Policy*

We hash the branch tag, global history, path history, history length to get the index and tag to access each table in the PMPM predictor. We use similar hash functions to those in [13] and [6] with empirically selected prime numbers. The indexes and tags use different primary numbers in their hash functions.

We use the short GHR table to store global histories with lengths 1 to 32. For longer histories, we use 21 different lengths in order to reduce the storage requirement. Both short and long GHR tables use 32-bit path history. M of the hit counters (i.e., the prediction counters with a tag match) are summed up with the bias counter to generate the GHR-based prediction. The LHR prediction table works the same way as the short GHR

table with the selection of *N* hit counters. The final prediction is selected by the meta counter.

3.3. Update Policy

The prediction counters in the prediction tables are updated if they have a tag match with the current branch. The tag contains both branch address and context information as described in Section 3.2.

In the case of a miss, i.e., the branch history has not been retained in the tables; new entries are allocated in the following way:

- For the short-GHR table, we assign new entries when the meta counter indicates that LHR prediction table is not sufficient to make correct predictions.
- For the long-GHR table, we assign new entries when the overall prediction is wrong.
- For the LHR table, we always assign new entries.

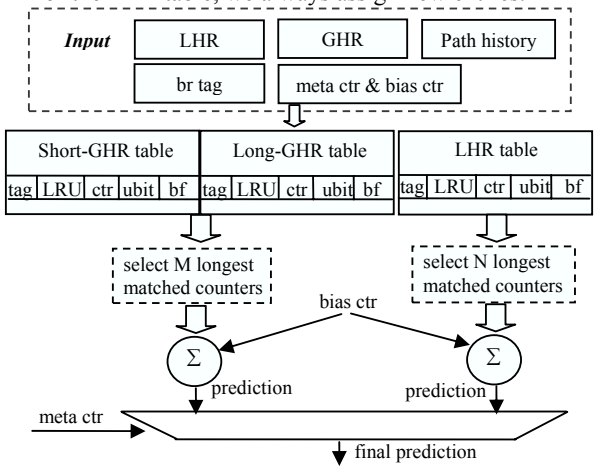

Figure 4. An optimized idealistic PMPM predictor.

3.4. Optimizations

In order to further improve the accuracy of the proposed PMPM predictor, we track the usefulness of each prediction counter using the usefulness (*ubit*) and benefit (*bf*) fields in the same entry. The *ubit* shows whether the prediction counter agrees with the branch outcome and the benefit counter shows whether the inclusion of the prediction counter is beneficial. The reason is that for some prediction counters, the inclusion has no impact on the final prediction since other prediction counters may be sufficient to make a correct prediction.

With the usefulness and benefit counters, we make two predictions: a prediction (*ubit_pred*) obtained by selecting the useful (*ubit* >= 0) counters and a prediction (*final_pred*) by selecting useful and beneficial (*bf >= 0*) counters. The *final_pred* is used as the final prediction. The *ubit_pred* is used to update the benefit counter as follows:

If the prediction counter is not included, the *ubit_pred* would change from correct to wrong. Then, we increase the corresponding benefit counter.

If the prediction counter is not included, the *ubit_pred* would change from wrong to correct. Then, we decrease the corresponding benefit counter.

3.5. Results

We simulate the proposed idealistic PMPM predictor with the configuration shown in Table 1.

Table 1. Idealistic Predictor Configuration.

Short-GHR Prediction Table	1M sets, 4-way
Long-GHR Prediction Table	2M sets, 4-way
LHR Prediction Table	512K sets, 4-way
Max # of selected counters for GHR matches	7
Max # of selected counters for LHR matches	16
Minimum GHR length of the geometric history lengths used by the Long-GHR Table	38
Maxmum GHR length of the geometric history lengths used by the Long-GHR Table	651
Range of a prediction counter	[-6, +6]
Range of a ubit	[-1, +1]
Range of a benefit counter (bf)	[-31, +31]

With the configuration shown in Table 1, the simulator consumes around 226M bytes of memory and takes 1.65 hours to finish all the benchmarks on a Xeron 2.8Ghz computer. The final misprediction rates (measured in MPKI) for the distributed traces of CBP2 are shown in Table 2. For comparison, Table 2 also includes the misprediction rates of the PPM predictor, i.e., in the idealistic predictor presented in Figure 4, the longest match is used instead of combining multiple matches. From Table 2, we can see that with the same predictor structure, the PMPM algorithm achieves significantly higher prediction accuracy than PPM for all traces except *vortex*. The misprediction rates reduction is up to 20.9% (*gzip*) and 15.2% on average.

Table 2. Misprediction rates of the idealistic predictors using the PPM and PMPM algorithm

Trace	PPM	PMPM	Trace	PPM	PMPM
Gzip	11.977	9.470	vpr	10.096	8.273
Gcc	2.748	2.594	mcf	9.258	7.688
crafty	2.083	1.794	parser	4.404	3.928
compress	6.526	5.167	jess	0.302	0.290
raytrace	0.274	0.272	db	2.721	2.199
javac	1.054	0.930	mpegaudio	1.051	0.922
mtrt	0.326	0.318	jack	0.484	0.472
eon	0.246	0.239	perlbmk	0.177	0.192
gap	1.164	1.081	vortex	0.090	0.087
bzip2	0.036	0.032	twolf	11.591	10.529
Average	PPM: 3.330			PMPM: 2.824	

4. REALISTIC PMPM PREDICTOR

In this section, we present our PMPM branch predictor design for practical implementation. As discussed in Section 2, the PMPM algorithm is built upon PPM. Therefore, we choose to develop our design based on the recently proposed PPM-like branch predictors [8], [13], the TAGE branch predictor [13] in particular.

4.1. Predictor Structure

The overall structure of the proposed PMPM predictor is shown in Figure 5. The predictor structure is very similar to a TAGE predictor, except that a local history prediction table is incorporated. The prediction and update policies, however, are completely redesigned to implement the PMPM algorithm.

As shown in Figure 5, the PMPM predictor contains a

bimodal prediction table [15], seven global prediction tables (labeled as "*gtable0*" to "*gtable6*") indexed by the branch address, global history and path history, and a local prediction table (labeled as "*ltable*") indexed by the branch address and local history. Geometrical history lengths [10] are used for the global prediction tables: *gtable6* is associated with the shortest global history and *gtable0* is associated with the longest global history. Each entry of the global and prediction tables has three fields: a *tag*, a signed saturated prediction counter (labeled as "*ctr*") and an unsigned saturated counter (labeled as "*ubit*") to track the usefulness of the prediction counter. Index and tag hashing functions for the global prediction tables are same as those used in TAGE predictors. The local prediction table uses hashed branch address and local history as the index and the XOR-folded (i.e., PC ^ (PC >> M)) branch address as the tag.

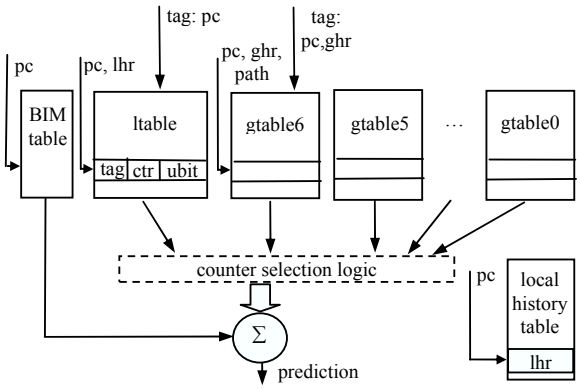

Figure 5. The practical PMPM predictor.

4.2. Prediction Policy

The first phase of prediction is to calculate indexes and tags for each table. Among the entries that have tag matches (i.e, the hit entries), we select out up to 4 prediction counters from global prediction tables and sum those counters with the prediction counter from the local prediction table, if there is a hit, and the counter from the bimodal table. If the sum is zero, we use the prediction from the bimodal table. Otherwise we use the sign of the sum as the prediction. In order to reduce the latency of counter selection, we devise a simple policy to select up to 4 counters from the global prediction tables rather than selecting several prediction counters with longest matches as used in the idealistic PMPM predictor. We divide the global prediction tables into 4 groups, (*gtable6*, *gtable5*), (*gtable4*, *gtable3*), (*gtable2*, *gtable1*) and (*gtable0*), and select out the longer match from each group.

The prediction counter from the local prediction table (*ltable*) is used only if its usefulness (*ubit*) is larger than 1.

The (tag-less) bimodal counter is always used in the summation process.

4.3. Update Policy

The prediction counter in the bimodal table is always updated. The update policies of the global prediction tables and the local prediction table are described as follows.

Similar to the perceptron predictor [4], [5], and the O-GEHL predictor [10], the prediction counters of the global prediction tables are updated only when the overall prediction is wrong or the absolute value of the summation is less than a threshold. We also adopt the threshold adaptation scheme proposed in the O-GEHL predictor to fit different applications. We only update those counters that have been included in the summation. At the same time, for each of these prediction counters, the associated *ubit* counter is incremented when the prediction counter makes a correct prediction. In the case of a misprediction, we also try to allocate a new entry. The new entry will be selected from tables where the branch misses. We select one entry that has a zero *ubit* from those tables as a new entry allocated for the current branch. If there are multiple entries with zero *ubits*, we select the one with the shortest history length. If there is no entry with a zero *ubit*, we don't allocate a new entry. At last, for each entry corresponding to a longer history than the longest match, its *ubit* counter is decremented.

If current branch hits in the local prediction table, we always update the prediction counter. The *ubit* counter is decremented if the corresponding prediction counter makes a wrong prediction. If the prediction counter makes a correct prediction, we increment *ubit* only when the overall prediction is wrong. If the current branch doesn't hit in the local prediction table and the overall prediction is wrong, we will try to allocate a new entry in the local prediction table. If the indexed entry has a zero *ubit*, a new entry is allocated. Otherwise, its *ubit* counter is decremented.

The base update policy described above is also improved by two optimizations. We modify the update policy so that in each group of two global prediction tables, a new entry will not be allocated in the table with shorter history length if the branch hits in the table with longer history length. For applications with a large number of hard-to-predict branches, some otherwise useful entries could be evicted due to frequent mispredictions using the base update policy. To address this issue, we use a misprediction counter to periodically detect those applications/program phases with a large number of hard-to-predict branches. For those application/phases, we slightly vary the update policy: on a misprediction, we don't decrement the *ubit* counters in those prediction tables that have tag misses if we already allocate a new entry; and we will decrement *ubit* of the longest match if its prediction counter is wrong. In this way, we limit the chance of useful entries to be victimized.

4.4. Hardware Complexity and Response Time

Similar to TAGE and O-GEHL predictors, the response time of the proposed PMPM predictor has three components: index generation, prediction table access, and prediction computation logic. In the proposed PMPM predictor, we use similar index functions to those in TAGE. The index and tag generation for the local prediction table has two sequential parts, the local branch history (LHR) table access and simple bitwise XOR of the LHR and the branch address for index. Since we use a small

1K-entry tagless LHR table with short LHRs, we assume the overall latency for generating index for the local prediction table is comparable to the complex index functions for global prediction tables, i.e., one full cycle. The prediction table access is also similar to the TAGE predictor. Rather than selecting the longest two matches (for the optimization related to the newly allocated entries) as in the TAGE predictor, we use an adder tree with up to 6 5-bit values to calculate the prediction, which should have similar latency to the prediction computation of the O-GEHL predictor. Therefore, we expect the response time of the proposed PMPM predictor should be very close to the TAGE or O-GEHL predictors.

4.5. Ahead Pipelining

As the prediction latency is more than one cycle, we use ahead pipelining [11] to generate one prediction per cycle. Assuming 3-cycle prediction latency, our 3-block ahead pipelining scheme for the PMPM predictor is shown in Figure 6. The impact of ahead pipelining on the proposed PMPM predictors will be examined in Section 4.7

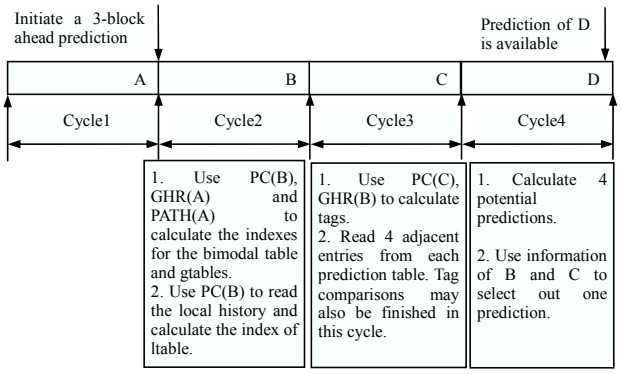

Figure 6. A 3-block ahead scheme for the PMPM predictor. Four basic blocks ending with branches A, B, C and D are fetched in cycle1 to cycle4.

It has been shown in [14] that speculative updates of branch history are important for prediction accuracy and an outstanding branch queue (OBQ) is proposed to manage speculative updates of both global and local branch histories. With ahead pipelining, the global history can be managed in the same way as proposed in [14]. However, for the update of local history, the indexes of the local history table and the OBQ need to use the n-block ahead branch address instead of the current branch address. In this paper, we assume that the OBQ is available to manage speculative updates for local branch history. Considering the extra hardware complexity of the OBQ, we also report the results of PMPM predictors without using local histories in Sections 4.6 and 4.7.

4.6. Prediction Accuracy

As the PMPM predictor is built upon the TAGE predictor, we compare their prediction accuracies. We simulated two 32kB PMPM predictors, a PMPM predictor with both global history and local history (labeled as "PMPM-GL") and a PMPM predictor with only global history (labeled as "PMPM-G"), and one 32kB TAGE predictor using the base configuration presented in [13]. The configurations are shown in Table 3. Those three predictors have the same global history input and similar table structures. Compared to the TAGE predictor, the PMPM-G predictor has larger prediction counters but smaller tags. A larger prediction counter is necessary for the PMPM predictors to suppress noises in the summation process. In order to save some space for local history related tables, the PMPM-GL predictor has a smaller bimodal table and smaller tags for three gtables compared to the PMPM-G predictor. To simplify the configurations, all bimodal tables in Table 3 have the same number of prediction bits and hysteresis bits. Table 4 shows the misprediction rates of these three predictors. Compared to the TAGE predictor, the average misprediction reductions are 6% and 2% respectively achieved by the PMPM-GL and the PMPM-G predictors.

Table 3. Configurations of the PMPM-GL, PMPM-G and TAGE predictors with 32kB storage budget.

PMPM-GL	History	10 bits local history; Geometrical global histories from 5 to 131; 16 bits path history.
	Table sizes	bimodal table: 8k entries; gtables: 2k entries; ltable: 1k entries; LHR table: 1k entries.
	Counter widths	Prediction ctr: 5 bits; ubit counter: 2 bits.
	Tag widths	gtable4 to gtable6: 8 bits; gtable0 to gtable3: 9 bits; ltable: 5 bits.
PMPM-G	History	Geometrical global histories from 5 to 131; 16 bits path history.
	Table sizes	bimodal table: 16k entries; gtables: 2k entries.
	Counter widths	Prediction ctr: 5 bits; ubit counter: 2 bits.
	Tag widths	9 bits.
TAGE	History	Geometrical global histories from 5 to 131; 16 bits path history.
	Table sizes	bimodal table: 16k entries; gtables: 2k entries.
	Counter widths	Prediction ctr: 3 bits; ubit counter: 2 bits.
	Tag widths	11 btis

Table 4. Misprediction rates (MPKI) of the PMPM-GL, PMPM-G and TAGE predictors with 32kB storage budget.

Trace	PMPM-GL	PMPM-G	TAGE	Trace	PMPM-GL	PMPM-G	TAGE
gzip	9.685	10.300	10.899	vpr	8.926	9.003	9.361
gcc	3.826	3.794	3.536	mcf	10.182	10.128	10.254
crafty	2.555	2.558	2.682	parser	5.332	5.437	5.422
compress	5.571	5.827	5.934	jess	0.413	0.464	0.456
raytrace	0.652	1.112	1.099	db	2.343	2.408	2.527
javac	1.105	1.144	1.160	mpegaudio	1.093	1.137	1.163
mtrt	0.734	1.188	1.139	jack	0.724	0.845	0.831
eon	0.305	0.470	0.487	perlbmk	0.319	0.497	0.480
gap	1.436	1.776	1.783	vortex	0.141	0.336	0.312
bzip2	0.037	0.043	0.041	twolf	13.447	13.466	13.758
Average	PMPM-GL: 3.441		PMPM-G: 3.597		TAGE: 3.666		

4.7. Realistic PMPM predictors for CBP2

In order to further improve the prediction accuracy of the PMPM predictors, we empirically tuned the configurations and used several optimizations for our realistic PMPM predictors for CBP2. The optimizations are listed as follows:

1) In the bimodal prediction table, four prediction bits will share one hysteresis bit as proposed in [12].
2) We use the number of branches that miss in all global prediction tables as a metric to detect traces with large branch footprints. For those traces, we periodically

reset the *ubits* as used in the TAGE predictor.

3) If the predictions from global prediction tables are same, we will limit the update of *ubits*.

Considering the extra hardware to support local history management, we submitted two versions of the PMPM predictor for CBP2. One predictor (labeled as "PMPM-CBP2-GL") uses both global and local histories. The other (labeled as "PMPM-CBP2-L") only uses global history. The configurations and hardware costs of the predictors are shown in Table 5. The prediction accuracies of these PMPM predictors are shown in Table 6. From the table, we can see that the local history is still important for some benchmarks (e.g., *raytrace, mtrt and vortex*) although we already use a very long global history.

Table 5. Predictor Configurations and Hardware Costs

	PMPM-CBP2-GL	PMPM-CBP2-G
Each prediction counter	5 (bits)	5 (bits)
Each ubit	2 (bits)	2 (bits)
History lengths for gtable 6 to gtable 0	4, 7, 15, 25, 48, 78, 203	4, 7, 15, 25, 48, 78, 203
Number of entries for each gtable	2k	2k
Tag widths for gtable 6 to gtable 0	6, 7, 7, 8, 9, 10, 10 (bits)	7, 8, 8, 10, 11, 12, 12 (bits)
Total cost of gtables	217088 (bits)	239616 (bits)
Ltable	1k entries; 5 bits tags	
Cost of the ltable	12288 (bits)	
Local history table	1k entries; His Len: 11	
Cost of the local history table	11264 (bits)	
Cost of the bimodal table	20480 (bits)	20480 (bits)
Global history register	203 (bits)	203 (bits)
Path history register	16 (bits)	16 (bits)
Cyclic shift registers for gtable tags and indexes	184 (bits)	206 (bits)
Cost of adaptation counters and flags	71 (bits)	71 (bits)
Total Hardware Cost	261594 (bits)	260592 (bits)

Table 6. Misprediction rates (MPKI) of the PMPM-CBP2-GL and PMPM-CBP2-G predictors

Trace	CBP2-GL	CBP2-G	Trace	CBP2-GL	CBP2-G
gzip	9.712	10.346	vpr	8.945	9.063
gcc	3.690	3.637	mcf	10.092	10.033
crafty	2.581	2.565	parser	5.215	5.244
compress	5.537	5.819	jess	0.393	0.433
raytrace	0.542	0.963	db	2.319	2.380
javac	1.107	1.159	mpegaudio	1.102	1.159
mtrt	0.657	1.009	jack	0.688	0.763
eon	0.276	0.359	perlbmk	0.314	0.484
gap	1.431	1.745	vortex	0.137	0.331
bzip2	0.037	0.042	twolf	13.551	13.616
Average	PMPM-CBP2-GL: 3.416		PMPM-CBP2-G: 3.557		

With the ahead pipelining scheme described in Section 4.5, we simulated the proposed PMPM predictions with ideal 1-block ahead and 2 to 4-block ahead pipelining schemes. The prediction accuracies of them are shown in Figure 7. As shown in the figure, with the 3-block ahead pipelining, the average accuracy loss is less than 0.16 MPKI compared to 1-block ahead pipelining.

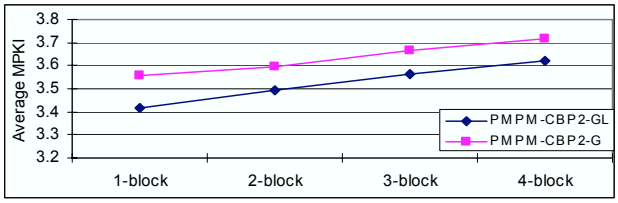

Figure 7. Misprediction rates of the ahead pipelined PMPM predictors for CBP2.

5. CONCLUSIONS

In this paper, we show that the PPM algorithm with the longest match may lead to suboptimal results. By combining multiple partial matches, the proposed PMPM algorithm can better adapt to various history length requirements. An idealistic implementation of the PMPM predictor is presented and it is used to evaluate how high the prediction accuracy can be achieved. We also proposed a realistic design based on the PMPM algorithm. Our results show that the realistic PMPM predictors can achieve higher accuracy than the TAGE predictors with the same storage cost.

REFERENCES

[1] I.-C. Chen, J. Coffey, and T. Mudge, Analysis of branch prediction via data compression, in Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1996.

[2] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications, vol. 32, pp. 396-402*, Apr. 1984.

[3] H. Gao and H. Zhou, "Branch Prediction by Combining Multiple Partial Matches", Technical report, School of EECS, Univ. of Central Florida, Oct. 2006.

[4] D. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons", In Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7), 2001.

[5] D. Jiménez and C. Lin, "Neural methods for dynamic branch prediction", *ACM Trans. on Computer Systems*, 2002.

[6] D. Jiménez. Idealized piecewise linear branch prediction. In *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[7] T. Juan, S. Sanjeevan, and J. J. Navarro. A third level of adaptivity for branch prediction. *In Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[8] P. Michaud. A ppm-like, tag-based predictor. In The 1st JILP Championship Branch Prediction Competition (CBP-1), 2004.

[9] P. Michaud and A. Seznec, A comprehensive study of dynamic global-history branch prediction. *Research report PI-1406, IRISA*, June 2001.

[10] A. Seznec. The O-GEHL branch predictor. In The 1st JILP Championship Branch Prediction Competition (CBP-1), 2004.

[11] A. Seznec and A. Fraboulet. Effective ahead pipelining of the instruction addres generator. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[12] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeidés. Design tradeoffs for the ev8 branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[13] A. Seznec, P. Michaud. A case for (partially) tagged Geometric History Length Branch Prediction. *Journal of Instruction Level Parallelism, vol. 8,* February 2006.

[14] K. Skadron, M. Martonosi, and D. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction Level Parallelism, vol. 2,* January 2000.

[15] J. E. Smith. A study of branch prediction strategies. In Proceedings of the 8th Annual International Symposium on Computer Architecture, 1981.

Looking for limits in branch prediction with the GTL predictor *

André Seznec
IRISA/INRIA/HIPEAC

Abstract

The geometric history length predictors, GEHL [7] and TAGE [8], are among the most storage effective conditional branch predictors. They rely on several predictor tables indexed through independent functions of the global branch/path history and branch address. The set of used global history lengths forms a geometric series, i.e., $L(j) = \alpha^{j-1}L(1)$. This allows to efficiently capture correlation on recent branch outcomes as well as on very old branches.

GEHL and TAGE differ through their final prediction computation functions. GEHL uses a tree adder as prediction computation function. TAGE uses (partial) tag match. For realistic storage budgets (e.g., 64Kbits or 256Kbits), TAGE was shown to be more accurate than GEHL. However, for very large storage budgets and very large number of predictor components, GEHL is more accurate than TAGE on most benchmarks. Moreover a loop predictor can capture part of the remaining mispredictions.

We submit the GTL predictor, combining a large GEHL predictor, a TAGE predictor and a loop predictor, to the idealistic track at CBP2.

Presentation outline

In the two past years, we have introduced the geometric history length predictors, GEHL [7] and TAGE [8]. These predictors are among the most storage effective branch predictors, TAGE being more efficient than OGEHL for limited storage budgets. These predictors are natural candidates to serve as basis for the design of a limit branch predictor.

In a a preliminary study, we found that, for very large storage budget and very large number of components, the GEHL predictor, i.e. OGEHL without dynamic history length fitting [7], achieves higher accuracy than the TAGE predictor: we got respective accuracy limits of 2.842 misp/KI and 3.054 misp/KI on the distributed set of traces.

* This work was partially supported by an Intel research grant, an Intel research equipment donation and by the European Commission in the context of the SARC integrated project #27648 (FP6).

Therefore the GTL predictor presented for the idealist track at CBP-2 is an hybrid predictor, using a very large GEHL predictor as its main component. It is combined with a TAGE predictor and a loop predictor.

In Section 1, we first briefly present the principles of the predictor components and the configurations used.

Section 2 presents the accuracy limits we found for the GTL predictor and its components.

1. The GTL predictor

The GTL predictor is illustrated on Figure 1. The GTL predictor features a GEHL predictor [7], a TAGE predictor [8] and a loop predictor as components.

The GEHL predictor is the main predictor, since it is the most accurate component. The GEHL predictor is used as the base predictor component in TAGE, i.e. when there is no partial hit on any of the TAGE components, the TAGE prediction is the GEHL prediction. Moreover, the GEHL prediction is used as part of the indices for the tagged components of the TAGE predictor as well as for the components of the metapredictor. Using a (partial) prediction as part of the index of another table was already used on the YAGS predictor [1] and the bimode predictor [4].

GTL uses a metapredictor to discriminate between the TAGE and the GEHL predictions. We use a metapredictor derived from the skewed predictor [5].

Finally, the prediction features a loop predictor. The loop predictor provides the prediction when a loop has been successively executed with 8 times the same number of iterations as in [2].

1.1. The loop predictor

The loop predictor simply tries to identify regular loops with constant number of iterations.

As in [2], the loop predictor provides the global prediction when the loop has succesively been executed 8 times with the same number of iterations. The loop predictor used in the submission features 512K entries.

1.2. Common features of TAGE and GEHL

The GEHL predictor [7] and the TAGE predictor [8] rely on several predictor tables indexed through indepen-

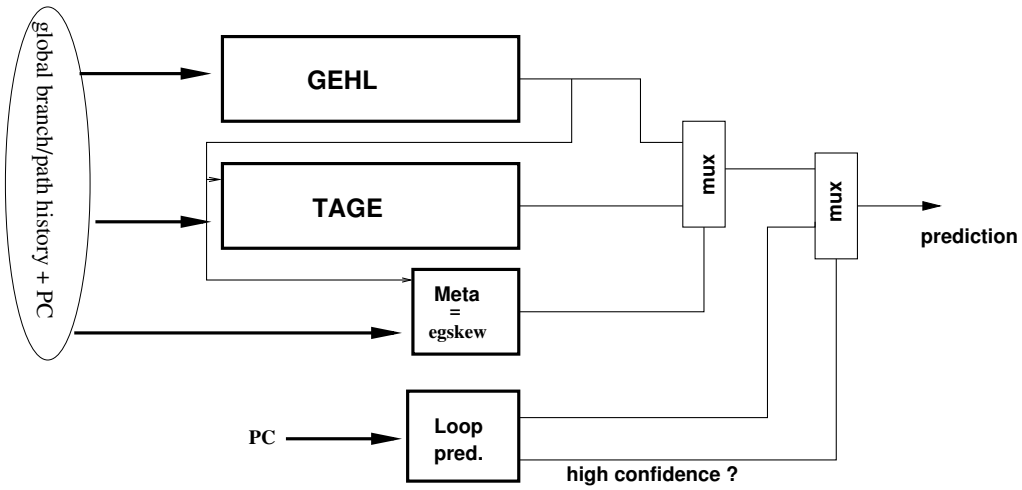

Figure 1. The GTL predictor

dent functions of the global branch/path history and branch address. On both predictors, the set of used global history lengths forms a geometric series, i.e, $L(i) = (int)(\alpha^{i-1} * L(1) + 0.5)$; table T0 is indexed with the PC. The use of a geometric series allows to use very long branch histories in the hundreds range for realistic size predictors (e.g., 64Kbits or 256Kbits). As pointed out in [7, 8], the exact shape of the series is not important. The TAGE predictor and the GEHL predictor essentially differs through their final prediction computation function.

1.3. The GEHL branch predictor [7]

1.3.1. General principle The GEometric History Length (GEHL) branch predictor features M distinct predictor tables Ti, $0 \leq i < M$ indexed with hash functions of the branch address and the global branch history. The predictor tables store predictions as signed counters. To compute a prediction, a single counter C(i) is read on each predictor table Ti. The prediction is computed as the sign of the sum S of the M counters C(i), $S = \frac{M}{2} + \sum_{0 \leq i < M} C(i)$ [1]. The prediction is taken if S is positive and not-taken if S is negative.

Distinct history lengths are used for computing the index of the distinct tables. Table T0 is indexed using the branch address. The history lengths used in the indexing functions for tables Ti, $1 \leq i < M$ form a geometric series.

1.3.2. Updating the GEHL predictor The GEHL predictor update policy is derived from the perceptron predictor update policy [3]. The GEHL predictor is only updated on mispredictions or when the absolute value of the computed sum S is smaller than a threshold θ. Saturated arithmetic is used. More formally, the GEHL predictor is updated as follows, *Out* being the branch outcome:

if ((p !=*Out*) or ($|S| \leq \theta$))
for each i in parallel
if *Out* then $C(i) = C(i)+1$ else $C(i) = C(i)-1$

1.3.3. Dynamic threshold fitting for the GEHL predictor Experiments showed that the optimal threshold θ for the GEHL predictor varies for the different applications. In [7], dynamic threshold fitting was introduced to accomodate this difficulty.

For most benchmarks there is a strong correlation between the quality of a threshold θ and the relative ratio of the number of updates on mispredictions NU_{miss} and the number of updates on correct predictions $NU_{correct}$: experimentally, in most cases , for a given benchmark, when NU_{miss} and $NU_{correct}$ are in the same range, θ is among the best possible thresholds for the benchmark.

A simple algorithm adjusts the update threshold while maintaining the ratio $\frac{NU_{miss}}{NU_{correct}}$ close to 1. This algorithm is based on a single saturated counter TC (for threshold counter).

if ((p !=*Out*) {TC= TC + 1; if (θ is saturated positive){$\theta = \theta + 1$; TC=0;} }
if ((p == *Out*) & ($|S| \leq \theta$)) {TC= TC - 1; if (θ is saturated negative){$\theta = \theta - 1$; TC=0;}}

Using a 7-bit counter for TC was found to be a good tradeoff.

1 For p-bit signed counters, predictions vary between -2^{p-1} and $2^{p-1} - 1$ and are centered on $-\frac{1}{2}$

1.3.4. GEHL configuration in the submitted predictor We leveraged the different degrees of freedom in the design of the GEHL predictor to get the best predictor that we could simulate with a memory footprint in the range of 768 Mbytes.

Experiments showed that, for a GEHL simulator with a memory footprint in the range of 768 Mbytes, using 97 tables provides a high level of accuracy [2] Experiments showed that using 8-bit counters leads to good prediction accuracy.

We also slightly improve the update policy by incrementing/decrementing twice the counters when they are between -8 and 7. This results in a gain of 0.009 misp/KI on the overall GTL predictor.

1.4. The TAGE predictor [8]

The TAGE predictor [8] is derived from Michaud's PPM-like tag-based branch predictor [6]. The TAGE predictor features a base predictor T0 in charge of providing a basic prediction and a set of (partially) tagged predictor components Ti. These tagged predictor components Ti, $1 \leq i \leq M$ are indexed using different history lengths that form a geometric series. For the submitted GTL predictor, the GEHL predictor component provides the base prediction.

An entry in a tagged component consists in a signed counter *ctr* which sign provides the prediction, a (partial) tag and an unsigned useful counter *u*. In this submission *u* is a 2-bit counter and *ctr* is a 5-bit counter.

1.4.1. Prediction computation on TAGE At prediction time, the base predictor and the tagged components are accessed simultaneously. The base predictor provides a default prediction. The tagged components provide a prediction only on a tag match.

In the general case, the overall prediction is provided by the hitting tagged predictor component that uses the longest history, or in case of no matching tagged predictor component, the default prediction is used.

However,on several applications, newly allocated entries provide poor prediction accuracy and that on these entries, using the alternate prediction is more efficient. This property is essentially global to the application and can be dynamically monitored through a single 4-bit counter (*USE_ALT_ON_NA* in the simulator).

Moreover, no special memorization is needed for recording the "newly allocated entry": we consider that an entry is newly allocated if its prediction counter is weak (i.e. equal to 0 or -1). This approximation was found to provide results equivalent to effectively recording the "newly allocated entry" information.

Therefore the prediction computation algorithm is as follows:

1. Find the longest matching component
2. if (the prediction counter is not weak or USE_ALT_ON_NA is negative) then the prediction counter sign provides the prediction else the prediction is the alternate prediction.

1.4.2. Updating the TAGE predictor We present here the various scenarios of the update policy that we implement on the TAGE predictor.

Updating the useful counter u The useful counter *u* of the provider component is updated when the alternate prediction *altpred* is different from the final prediction *pred*.

Updating the prediction counters The prediction counter of the provider component is updated. For large predictors also updating the alternate prediction when the useful counter of the provider component is null results in a small accuracy benefit.

Allocating tagged entries on mispredictions For a 64 Kbits 8-component predictor [8] or 256 Kbits TAGE predictors (submissions to realistic tracks), allocating a single entry is the best tradeoff. For the very large predictor considered here, allocating all the available entries is more effective [3].

The allocation process is described below.

The M-i u_j counters are read from predictor components Tj, $i < j \leq M$. Then we apply the following rules.

(A) Avoiding ping-pong phenomenon: in the presented predictor, the search for free entries begins on table Tb, with b=i+1 with probability 1/2, b=i+2, with probability 1/4 and b=i+3 with probability 1/4.

(B) Priority for allocation For all components Tk, $k > b$, if $u_k = 0$ then the entry is allocated.

(C) Initializing the allocated entry: An allocated entry is initialized with the prediction counter set to weak correct. Counter *u* is initialized to 0 (i.e., *strong not useful*).

1.4.3. TAGE configuration in the submitted predictor We leveraged the different degrees of freedom in the design of the TAGE predictor to get the best predictor that we could simulate while maintaining the total memory footprint of the simulator smaller than 1 gigabyte.

The TAGE component in the submitted predictor feature a total of 19 tagged components, the GEHL predictor is used as the base predictor. Each table feature 1M entries. The tag width is 16 bits on the tagged tables.

2 257 would have been a marginally better choice but the simulation time was too long for CBP2 execution time constraints.

3 An entry is considered as free if its useful counter *u* is null.

1.5. Selecting between TAGE and GEHL predictions

As a meta-predictor to discriminate between TAGE and GEHL predictors, we found that a skewed predictor [5] works slightly better than a bimodal table (by 0.004 misp/KI). The respective history lengths of the three tables are 0, 4 and 22. As for the TAGE predictor component, the output of the GEHL predictor is used to index the meta-predictor.

For one benchmark, the GEHL+TAGE predictor exhibited a slightly higher misprediction rate than the GEHL predictor alone. Therefore to avoid this situation, we use a single safety counter that monitors that GEHL+TAGE against GEHL alone.

1.6. Information for indexing the branch predictor

1.6.1. Path and branch history The predictor components are indexed using a hash function of the program counter, the global history combining branch direction and path (7 address bits). Non-conditional branches are included in these histories.

1.6.2. Discriminating kernel and user branches Kernel and user codes appear in the traces. In practice in the traces, we were able to user code from kernal through the address range. In order to avoid history pollution by kernel code, on jumps in kernel code, we save the global branch history and path histories and restore them when jumping back in user code. Compared with using a single history, 0.074 misp/KI improvement is achieved.

1.6.3. Indexing the TAGE predictor component and the metapredictor The TAGE predictor is used to try to correct predictions of the GEHL predictor. Therefore the output of the GEHL predictor is used as part of the indices of the TAGE predictor components and the metapredictor.

1.6.4. History lengths Geometric length allows to use very long history lengths. Experiments showed that using 2,000 as the maximum history length for both TAGE and GEHL predictors is a good choice, but that using respectively 400 for GEHL and 100,000 for TAGE is marginally better (by 0.014 misp/KI)[4].

For the GEHL predictor, we force the use of distinct history lengths by enforcing the property $L(i) > L(i+1)+1$.

1.7. Static prediction

On the first occurence of a branch, a static prediction associated with the branch opcode is used: this allows to reduce the overall misprediction rate by 0.005 misp/KI.

4 100,000 is 0.002 misp/KI better than 10,000

2. Simulation results

The average predictor accuracy of GTL is **2.717 misp/KI** on the distributed set of traces. Removing the loop predictor and the static prediction, i.e., GEHL+TAGE, one will still obtain 2.774 misp/KI, the accuracy of the GEHL predictor component alone being 2.891 misp/KI. A GEHL predictor alone using a 2,000 branch history length achieves 2.842 misp/KI.

Results for the GTL and GEHL+TAGE are displayed per application in Table 1. Results for GEHL predictor using a 2,000 branch history length are also displayed as a reference. One can note that the benefit of loop prediction is marginal apart on *164.gzip* and to a less extent on *201.compress*.

3. Conclusion

In a preliminary study, we found that, at a a very large storage budget and for very large number of components, the GEHL predictor outperforms the TAGE predictor which accuracy reaches a plateau with medium number of components (around 16) and medium storage budget (around 64 Mbytes): a 256 Kbits 13-component TAGE predictor exhibits only approximately 10% more mispredictions than the best no-storage limit TAGE predictor we found. Therefore, the GTL predictor is built around the GEHL predictor.

To grab the last pieces of predictability contained in global branch/path history, we combine GEHL and TAGE, using the GEHL prediction as the base predictor and as a partial index for TAGE.

A loop predictor can improve a little bit the prediction accuracy of the GEHL+TAGE predictor. On the set of benchmarks for CBP2, the loop predictor has only small return apart on 164.gzip.

Apart the loop predictor, and despites our best efforts, we have not found any way to integrate a local history predictor bringing any benefit to the GTL predictor.

References

[1] A. N. Eden and T.N. Mudge. The YAGS branch predictor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Dec 1998.

[2] Hongliang Gao and Huiyang Zhou. Adaptive information processing: An effective way to improve perceptron predictors. *Journal of Instruction Level Parallelism (http://www.jilp.org/vol7)*, April 2005.

[3] D. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High Perform ance Computer Architecture*, 2001.

[4] C-C. Lee, I-C.K. Chen, and T.N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Dec 1997.

	164	175	176	181	186	197	201	202	205	209
GTL	9.393	7.783	2.434	7.312	1.591	3.891	5.260	0.293	0.234	2.168
GEHL+TAGE	9.992	7.785	2.472	7.350	1.602	3.901	5.397	0.302	0.247	2.175
GEHL	10.166	7.919	2.490	7.663	1.630	3.969	5.480	0.321	0.241	2.234
	213	222	227	228	252	253	254	255	256	300
GTL	0.946	0.943	0.271	0.425	0.177	0.128	1.129	0.087	0.033	9.858
GEHL+TAGE	0.991	0.990	0.287	0.442	0.180	0.137	1.211	0.093	0.041	9.890
GEHL	1.043	1.035	0.291	0.485	0.179	0.183	1.257	0.108	0.044	10.113

Table 1. Per benchmark accuracy in misp/KI

[5] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, June 1997.

[6] Pierre Michaud. A ppm-like, tag-based predictor. *Journal of Instruction Level Parallelism (http://www.jilp.org/vol7)*, April 2005.

[7] A. Seznec. Analysis of the o-gehl branch predictor. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, june 2005.

[8] André Seznec and Pierre Michaud. A case for (partially)-tagged geometric history length predictors. *Journal of Instruction Level Parallelism (http://www.jilp.org/vol7)*, April 2006.

Neuro-PPM Branch Prediction

Ram Srinivasan[†] [*], Eitan Frachtenberg[†], Olaf Lubeck[†], Scott Pakin[†], Jeanine Cook[*]

[†]CCS-3 Modeling, Algorithms and Informatics. Los Alamos National Laboratory.
[*]Klipsch School of Electrical and Computer Engineering. New Mexico State University.
{rsri@lanl.gov, etcs@cs.huji.ac.il, olubeck@lanl.gov, pakin@lanl.gov, jcook@nmsu.edu}

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 *m*th-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 *m*th-order PPM scheme first attempts to predict using the *m* 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 neu-

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

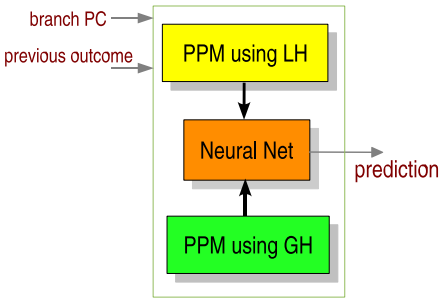

Figure 2. The Neuro-PPM predictor

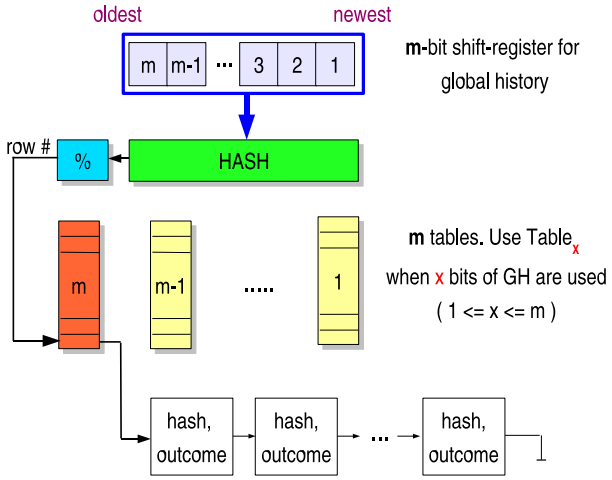

Figure 3. The global PPM predictor

ral 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 $\sum_{L=1}^{m} 2^L$. 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 decreased the MPKI (Mispredicts Per Kilo Instruction) for *twolf* by 30% and improved 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 storing the program counter (PC)

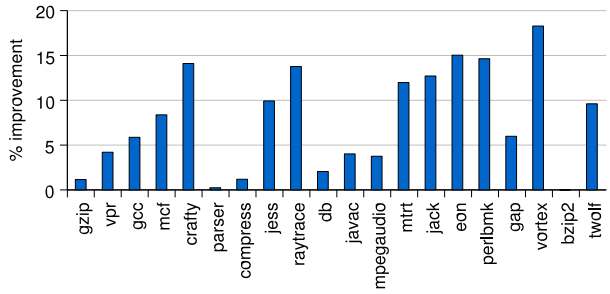

Figure 4. Improvement in prediction accuracy due to PC tag match

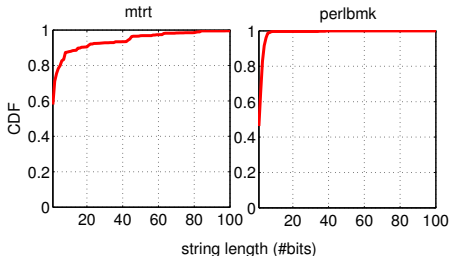

Figure 5. Cumulative distribution function of string length in *mtrt* and *perlbmk*

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 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 0 0 0 0 1 1 0 0 0 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 increased MPKI in none of 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 $Bias_{taken} > 0.5$, 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: $counter_{\{1\}} = -2$ and $counter_{\{0\}} = +5$. This suggests that pattern $\{1\}$ is biased

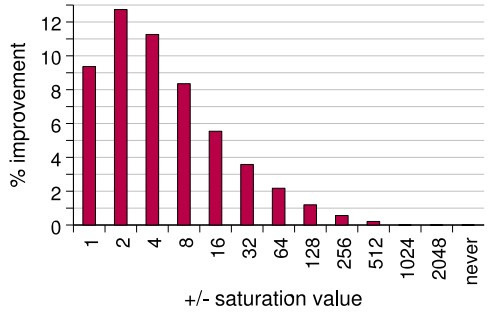

Figure 6. Percent improvement in *crafty*'s prediction accuracy when saturating bias counters are used in lieu of non-saturating counters

towards *not taken* and pattern $\{0\}$ towards *taken*.

Pattern bias appears to exhibit 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 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.

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 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 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

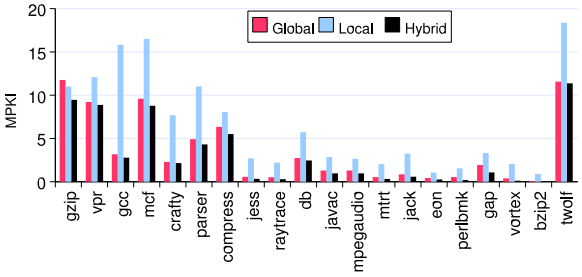

Figure 7. MPKI for the local-PPM, global-PPM, and hybrid predictor

$m-1$ 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 7 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.

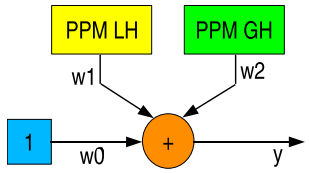

Figure 8. The neural-network mixer

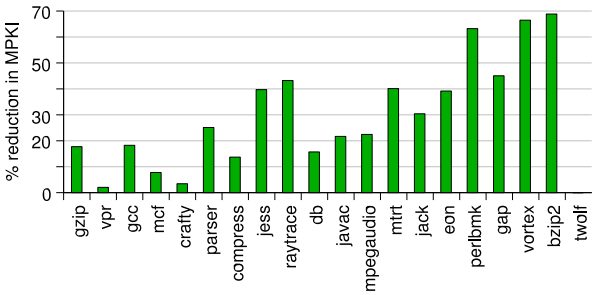

Figure 9. Percent reduction in MPKI when a perceptron instead of voting is used in the selector

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

Figure 9 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.

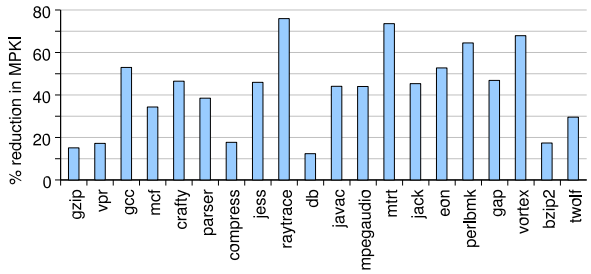

Figure 10. Percent reduction in MPKI for the idealistic scheme over the a realistic PPM predictor

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 10 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*, *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 RLE-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/`.