Revisiting Local History to Improve the Fused Two-Level Branch Predictor

Yasuo Ishii
The University of Tokyo, NEC
yishii@is.s.u-tokyo.ac.jp

Keisuke Kuroyanagi
The University of Tokyo
ksk9687@is.s.u-tokyo.ac.jp

Takeo Sawada
The University of Tokyo
tsawada@is.s.u-tokyo.ac.jp

Mary Inaba
The University of Tokyo
mary@is.s.u-tokyo.ac.jp

Kei Hiraki
The University of Tokyo
hiraki@is.s.u-tokyo.ac.jp

ABSTRACT

For a long time, branch predictors that use local history have employed a large table to provide a dedicated branch history series for each branch instruction. This has increased the cost of the local history table and complexity of management of the branch history. We have explored the design space of the local history to reduce the total cost and the complexity. We found that a predictor that uses per-set branch history, which holds branch history in a moderately-sized local history table, outperforms a predictor that uses a conventional, large, local history table.

In this paper, we propose the FTL++ branch predictor, which exploits the benefit of per-set branch history. This predictor reduces the number of local history table entries and extends the length of the local history to improve the prediction accuracy. Furthermore, we combine optimization techniques to the FTL++. The optimized FTL++ branch predictor achieves higher prediction accuracy than existing branch predictors.

1. INTRODUCTION

Dynamic branch prediction uses global history and local history to achieve higher prediction accuracy. Use of local history is an effective way to detect control structures, such as the loop structure. Therefore, local history is used in many existing branch predictors. However, branch predictors using local history have several problems. First, the storage cost of local history is much higher than that of global history because local history requires a large table, which is called a local history table (LHT), for tracking the branch history of each branch instruction, while global history requires only a simple shift register, called a global history register (GHR). Moreover, managing the local history table requires a complex mechanism because it has to keep the information of in-flight branches [1].

In this paper, we explore the best configuration of local history table to improve prediction accuracy and cost efficiency. We disclose that a local history table with moderate number of entries has lower cost and complexity yet produces good prediction accuracy. Such local history table is also called a per-set branch history [3]. We propose the FTL++ branch predictor, which utilizes per-set branch history to improve prediction accuracy.

2. THE FTL BRANCH PREDICTOR

Design of the FTL++ is derived from the Fused Two-Level (FTL) Branch Predictor [2] that extends the Optimized GEometric History Length (O-GEHL) branch predictor [5]. The FTL is our previous work and was proposed in the previous competition. The overall structure of the FTL and the O-GEHL is shown in Figure 1.

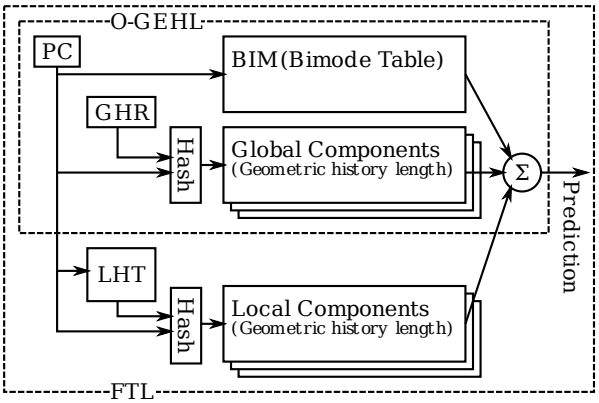

Figure 1: O-GEHL and FTL Branch Predictor.

The O-GHEL contains a bimodal component (labeled as BIM) and global prediction components (labeled as global components) indexed by the branch address and global history. Each entry of the prediction components is a saturating counter that represents a bias of a corresponding branch history pattern. The predictions from these components are reduced through an adder tree and the sign of the sum represents the prediction result. The key feature is the history length policy of the hash functions, which is called the geometric history length. Geometric series (e.g., 2, 4, 8, 16 ...) are used for the history length of hash functions. By using geometric history length, the branch predictor can effectively use very long branch history (longer than 100).

The FTL extends the O-GEHL by using local history. It employs a local history table (labeled as LHT) and additional prediction components (labeled as local components) that use the local history. These additional components also use the geometric history length. The FTL outperforms the O-GEHL because it can detect several control structures, such as loop structure, effectively.

3. REVISITING LOCAL HISTORY

As a preliminary study, we explore the best configuration of the local history table. For a long time, local history tables have been designed to provide a dedicated branch history series for each branch because many previous

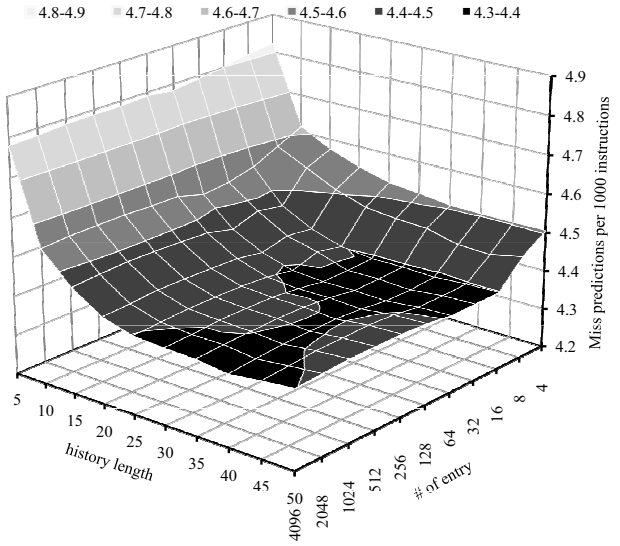

Figure 2: Prediction accuracy in local history design space (several parameters differ from our final predictor).

algorithms have assumed that this type of approach improves accuracy [3]. However, we think the design trade-offs of the local history is changed because the geometric history length in the local components will help to handle a much longer history length than existing predictors.

To confirm the best configuration of the local history, we evaluate the FTL with ten global components and five local components in a CBP3 framework. We use a fixed global history length and vary the number of local history entries and the length of the local history. The cost of the local history table is ignored in this evaluation. Figure 2 shows the result. Surprisingly, this result shows several characteristics that contradict the conventional assumption. (1) All predictors using more than sixteen entries achieve the same accuracy range. (2) A moderate number of local history entries (around 32 entries) show slightly higher prediction accuracy than larger number of entries.

We conclude that the local history table that employs a moderate number of entries is the most cost-efficient. From previous study [3] that categorized branch history, local history that stores a moderate number of entries is called per-set branch history in this paper. This preliminary study also includes several interesting features, but further analysis is intended to be part of our future work.

4. THE FTL++ BRANCH PREDICTOR

The FTL++ improves upon the FTL by using per-set branch history instead of conventional local history. As well as history management, the FTL++ also employs several techniques, which contains minor updates of a prediction algorithm and novel filtering mechanisms, to improve the prediction accuracy.

4.1 Prediction Computation

In the FTL++, branch predictions are only performed in the fetch stage. In the fetch stage, the FTL++ calculates indexes for prediction components. Multiple prediction counters are read in parallel from prediction components. The predictor sums the counters through an adder tree. The sign of the sum is used as a prediction result.

Contrary to the previous study, the prediction result from the adder tree can be overwritten by the sign of a counter from a BIM. This happens only when the absolute value of the sum is much smaller than the updating threshold θ. We call this feature a BIM counter overwriting. This feature is motivated by the knowledge that the absolute value of the sum has been strongly correlated to its confidence level [4]. The details of this feature are described in Section 4.4.

4.2 Updating Prediction Components

An update of the FTL++ is performed in the retire stage. The prediction components are updated on misprediction or when the absolute value of the sum is smaller than the updating threshold θ. Counters used for the prediction are incremented when a branch is taken. Otherwise, the counters are decremented. In the retire stage, the predictor calculates the sum as performed in the fetch stage in order to compare the absolute value of the sum and the updating threshold θ. Unlike existing perceptron-like predictors, the FTL++ updates a BIM when the counter from the BIM is not equal to the branch outcome.

For this updating policy, the updating threshold θ is one of the most important parameters. The FTL++ adjusts the updating threshold θ dynamically. Section 4.3 describes how to adjust the updating threshold θ.

4.3 Dynamic Threshold Fitting for a Deeply Pipelined Processor

The updating threshold θ significantly affects the accuracy of the predictor using the adder tree [6]. To optimize the threshold value, we employ the dynamic threshold fitting that is derived from the O-GEHL. The threshold counter (TC) is used to decide the best updating threshold θ. The FTL++ modifies the updating algorithm of TC to monitor the ratio of the number of updates on mispredictions and the number of updates on correct predictions in the deeply pipelined processor. The modified algorithm is shown in Algorithm 1.

Algorithm 1 Updating the Threshold Counter (TC)

```
sum_in_fetch ← output of adder tree at the fetch stage
sum_in_retire ← output of adder tree at the retire stage
if (sum_in_fetch ≥ 0) ≠ outcome then
  if (sum_in_retire ≥ 0) = outcome then
    TC ← TC + 2 /* A prediction result is changed
    between the fetch stage and the retire stage */
  else
    TC ← TC + 1 /* Update on a misprediction */
  end if
else
  if |sum_in_fetch| < θ then
    TC ← TC − 1 /* Update on a correct prediction */
  end if
end if
```

In this new updating policy, the prediction result in the fetch stage is required in the retire stage. To support this feature, the FTL++ employs a 128-entry circular buffer for holding previous prediction results. When the predictor makes a prediction, the result is stored in the buffer. When

the predictor updates a threshold counter, the predictor reads the prediction from the buffer.

4.4 BIM Counter Overwriting

The sum of the prediction counters provides not only prediction result but also the confidence level of the prediction. When the absolute value of the sum is small, the confidence level of the prediction becomes low. Such predictions, whose confidence level is low, often become less accurate than predictions from much simpler branch predictors, such as a bimodal predictor. To improve these cases, the FTL++ employs a BIM counter overwriting.

To track the confidence level of the prediction from the adder tree, we add a counter that is called a BIM counter (BC). The BC shows the confidence level of the prediction from the adder tree, when the absolute value of the sum is small. When the BC is positive and the sum of the fetch stage is smaller than half of the updating threshold θ, the FTL++ use a prediction from the BIM as the prediction result. Algorithm 2 shows how the BC is updated. When the sum of the fetch stage is smaller than half of the updating threshold θ, the predictor updates the counter. Until the corresponding branch is retired, the information about whether the sum in the fetch stage is smaller than half of the updating threshold θ is stored in the circular buffer as described in Section 4.3.

Algorithm 2 Updating the BIM Counter (BC)

```
sum_in_fetch ← output of adder tree at the fetch stage
bim_in_retire ← output of the BIM at the retire stage
/* When BC ≥ 0, BIM can be used as a prediction.
Otherwise, sum is always used as a prediction. */
if (|sum_in_fetch| < θ/2) and
((sum_in_fetch ≥ 0) ≠ (bim_in_retire ≥ 0)) then
  if (bim_in_retire ≥ 0) = outcome then
    BC ← BC + 1 /* BIM makes a correct prediction */
  else /* (sum_in_fetch ≥ 0) = outcome */
    BC ← BC − 1 /* sum makes a correct prediction */
  end if
end if
```

4.5 Filter Predictor

We apply two different filtering strategies for the FTL++.

4.5.1 Whitelist Filtering

The first type of filtering strategy is whitelist filtering. The whitelist filtering mechanism is designed to filter easy-to-predict branches. The filtering components overwrite the prediction when the filtering conditions are met. When the overwritten prediction results in a correct prediction, the base predictor cancels the update of the prediction components. We employ a bias filter and a loop predictor as the whitelist filter. These filtering schemes were already proposed in the previous competitions [7].

The bias filter employs a table indexed by the branch address. Each entry has four states (initial, always-taken, always-nottaken, and normal). Other predictors make their prediction when the corresponding state is normal.

The loop predictor employs a set-associative prediction table for tracking the behavior of the loop instruction. Originally, each entry in the prediction table consists of the branch direction of a corresponding branch instruction, the loop length counter, the current iteration counter, the tag, and the replacement information. We use Least Recently Used (LRU) replacement policy for our loop predictor. We duplicate the current iteration counter to update the counter in the fetch stage. The other counters and flags are updated in the retire stage.

4.5.2 Blacklist Filtering

The other type of the filtering strategy is a blacklist filtering. A blacklist filtering mechanism is designed to filter hard-to-predict branches. The role of the blacklist filter is to prevent the predictor from training for hard-to-predict branches. It reduces the meaningless training because the hard-to-predict branch cannot be predicted correctly, even if the predictor gets enough training. Blacklist filtering also reduces destructive aliasing of prediction components. As the blacklist filter, we propose a give-up filter and an exceptional filter for the FTL++.

The give-up filter tracks the branch instructions whose prediction accuracy is lower than that of the bimodal branch predictor. The branch address is registered to the give-up table on misprediction. The filter tracks the accuracy of the bimodal prediction and the prediction of the base predictor. When it detects that the bimodal prediction is more accurate than that of the base prediction, the filter overwrites the final prediction and prevents the base predictor from training for the branch instruction.

The exceptional filter tracks a branch instruction whose prediction sum is far from correct. When the absolute value of the sum is large (in other words, the confidence level of the prediction is high) and the prediction results in a wrong prediction, the exceptional filter starts to track the branch. When such situation occurs repeatedly on the tracked branch, the predictor stops training for the corresponding branch.

4.6 Optimizations for the Competition

We introduce the detailed configurations for the CBP3 in this section. Figure 3 shows the overview of an optimized FTL++. The FTL++ employs four different branch histories and eighteen prediction components for generating the prediction result. The FTL++ collects global history, two per-set histories, and conventional local history. For per-set history, the predictor employs two types of history tables. One contains 32 entries, and the other contains eight entries. The per-set history tables are indexed by a hashed branch address that is folded by the exclusive-OR. The local history table contains 1024 entries.

The global history register and the per-set history table also contain a path history, which is a part of the previous branch address. In this study, 1-bit of the branch address is used as a path history. The predictor updates the global history and the per-set histories on all branch instructions. On unconditional branches and indirect branches, we put a path history into the global history instead of treating them as taken branches. According to [8], the behavior of branches after CALLs and RETs has little correlation to prior branches. To improve prediction accuracy for such branches, we append 3 bits on CALL instructions and 2 bits on RET instructions into global history and per-set history. All resources for the branch histories are duplicated to support speculative updates. We update only speculative branch histories in the fetch stage. The other branch

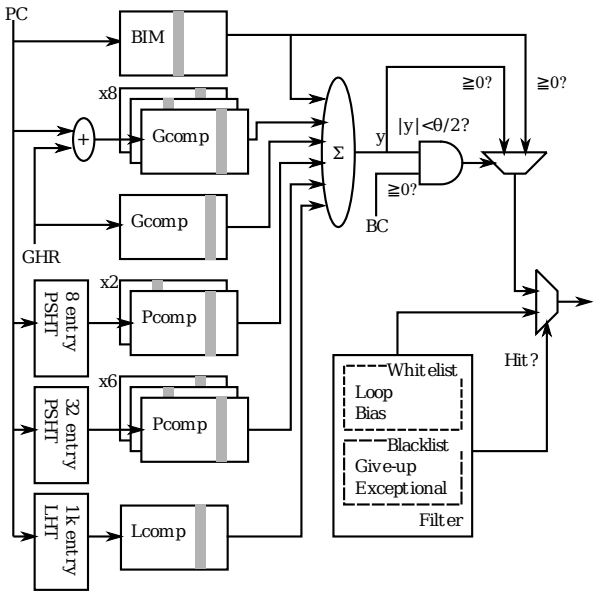

Figure 3: The FTL++ Branch Predictor.

histories are updated in the retire stage.

The prediction components are divided into five groups by correlated branch history. Eight components are indexed by the branch address and global history. Six components are indexed by the branch address and per-set history that classifies branch history into 32 groups. Two components are indexed by the branch address and per-set history that classifies branch history into eight groups. One component is indexed by the branch address and conventional local history whose table has 1024 entries. One component is indexed by only global history. We found that the FTL++ using only the global history register and one per-set history table with 32-entry generates accurate predictions, but combining multiple local histories leads to further improvement in prediction accuracy.

4.7 Budget Count

The detailed configuration and the budget counting for the predictor are shown in Table 1. The total budget size is less than 532480 bit (65KB).

5. CONCLUSION

To improve prediction accuracy, we explore the design space of local history. We have found that the local history table with small number of entries requires low hardware resources and still produces good accuracy. This kind of local history is called per-set branch history. Based on this consideration, we propose the FTL++ branch predictor. We improve the prediction accuracy by using cost-efficient per-set history. We also propose an enhancement of the dynamic threshold fitting and the BIM counter overwriting. As well as the branch prediction algorithm, we also propose two filtering mechanisms to prevent hard-to-predict branches from polluting history tables uselessly. In the future, we will explore a more cost-efficient branch prediction algorithm using local history to improve prediction accuracy.

Table 1: Configuration and Budget Count.

	Configuration	Budget
BIM	4096-entry × 6-bit	24576
Global history	(293-bit + 33-bit path) × 4	1304
	9-components × 4096-entry × 6-bit	221184
	12-bit folded history × 8 × 4	384
Per-set history 32-entry	(72-bit + 33-bit path) × 32 × 2	6720
	6-components × 4096-entry × 6-bit	147456
	12-bit folded history × 32 × 6 × 2	4608
Per-set history 8-entry	(16-bit + 16-bit path) × 8 × 2	512
	2-components × 4096-entry × 6-bit	49152
	12-bit folded history × 8 × 2 × 2	384
Local history	1024-entry × 3-bit history × 2	6144
	1-component × 4096-entry × 6-bit	24576
Bias	16384-entry × 2-bit	32768
Loop	16-entry × 8-way × 64-bit	8192
Giveup	8-entry × 8-way × 23-bit	1792
Except	8-entry × 8-way × 28-bit	1472
Others	1-bit flag × 2	2
	32-bit pc buffer × 2	64
	128-entry × 3-bit circular buffer	384
	7-bit buffer pointer × 2	14
	12-bit threshold counter (TC)	12
	10-bit BIM counter (BC)	10
Total		531710

6. REFERENCES

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

[2] Y. Ishii, "Fused two-level branch prediction with ahead calculation," *The Journal of Instruction Level Parallelism*, vol. 9, May 2007.

[3] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pp. 257–266, 1993.

[4] V. Desmet, L. Eeckhout, and K. De Bosschere, "Improved composite confidence mechanisms for a perceptron branch predictor," *J. Syst. Archit.*, vol. 52, pp. 143–151, March 2006.

[5] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pp. 394–405, 2005.

[6] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pp. 197–206, 2001.

[7] H. Gao and H. Zhou, "Adaptive information processing: An effective way to improve perceptron predictors," *The Journal of Instruction Level Parallelism*, vol. 7, April 2005.

[8] L. Porter and D. M. Tullsen, "Creating artificial global history to improve branch prediction accuracy," in *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pp. 266–275, 2009.