H-Pattern: A Hybrid Pattern Based Dynamic Branch Predictor with Performance Based Adaptation

Samir Otiv, Kaushik Garikipati, Milan Patnaik, V. Kamakoti

Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India

ABSTRACT

This paper proposes a novel hybrid pattern based branch predictor (*H-Pattern*) which uses a dynamic learning approach to find patterns in the execution of conditional branches. *H-Pattern* is comprised of two branch predictors - our proposed *nBPAT* (N Bit pattern) predictor and an alternate predictor (henceforth referred to as *AltPred*) that can be any other predictor such as *GShare*, *TAGE* or *ISL-TAGE*. The local *nBPAT* predictor aims to capture patterns in branch behavior. If the pattern predictor is in its learning phase, the *AltPred* predictor is used. A performance based selection is carried out between the *nBPAT* predictor and *AltPred*, when both are available. On implementing *H-Pattern* with *GShare* on the CBP simulator with all 40 traces, we achieved $\mathbf{3.8}$, $\mathbf{4.7}$ and $\mathbf{6.4}$ mispredictions per kilo instructions (MPKI) for the unlimited, 32KB and 4KB storage budgets respectively. On implementing *H-Pattern* with *TAGE*, we achieved $\mathbf{2.134}$, $\mathbf{2.644}$ and $\mathbf{3.712}$ mispredictions per kilo instructions (MPKI) and with *ISL-TAGE*) we achieved $\mathbf{2.058}$, $\mathbf{2.542}$ and $\mathbf{3.691}$ mispredictions per kilo instructions (MPKI) for the unlimited, 32KB and 4KB storage budgets respectively.

Keywords

Branch Prediction, Pattern Matching.

1. INTRODUCTION

Modern day computer architectures heavily rely on speculations to increase instruction-level parallelism (ILP). As a result of effective speculations, data that is likely to be read in the near future can be prefetched, and predicted values can be used before the actual values are available [1]. Among this data, conditional branch instructions form a critical factor for effectiveness of ILP as they can cause instructions to stall resulting in high execution cost. Rather than stall when a branch is encountered, a pipelined processor uses branch prediction to speculatively fetch and execute instructions along the predicted path. Hence, efficient branch prediction forms an important factor for increasing CPU performance which are driven by accurate prediction mechanisms. In this regard, machine learning techniques can further improve performance of the branch predictors by increasing the accuracy of predictions. In this paper, we propose one such learning based technique that can be efficiently implemented in hardware to improve branch prediction.

As pipelines deepen and the number of instructions issued per cycle increases, the penalty for a misprediction increases. Past efforts to improve branch prediction focus primarily on eliminating aliasing in the two-level adaptive predictors [2] [3] [4] [5] which occurs when two unrelated branches destructively interfere by using the same prediction resources. Our technique largely differs to previous work and focuses on improving the accuracy of the prediction mechanism itself.

1.1 Contributions

Most common techniques proposed for branch prediction rely on the effectiveness of using the global and local histories. The proposed predictor effectively finds regular patterns in the local execution history and uses them for future predictions. The key features of our proposal are as follows:

- We attempt to address the problem of branch prediction by incorporating a prediction mechanism which shows good accuracy, and at the same time is very simple to implement.
- We propose a novel pattern based branch predictor (*nBPAT*) using dynamic learning.
- Our predictor can predict branches for many executions ahead in time as compared to the existing branch predictors which predict the next instance only.
- We provide a technique to capture the correlation between events of the same branch over time.
- We implement an effective and efficient technique to learn all patterns with length upto n bits using only $O(n)$ space complexity.

The rest of the paper is organized as follows: The proposed pattern based branch prediction technique is explained in section II. The integration of proposed predictor with alternate predictors is discussed in section III. Section IV highlights the experiments undertaken and results obtained. Section V concludes the paper.

2. H-PATTERN BRANCH PREDICTOR

This section explains the proposed *H-Pattern* predictor organization. The *H-Pattern* predictor consists of our proposed *n-BPAT* predictor and one alternate predictor which we refer to as *AltPred* in this paper. The *n-BPAT* predictor is combined with *AltPred* using a saturating counter array for selection, like the one used in [3]. In order to obtain accurate predictions, we use a dynamic adaptation scheme to select the branch predictor which has been performing better in the recent time window. This ensures that the best branch predictor is used for prediction at each prediction instance. We now explain the functioning of our proposed novel pattern based *nBPAT* predictor.

2.1 nBPAT Predictor

The foundation of the *nBPAT* predictor rests on the premise that conditional branches in a program often follow a repetitive pattern of decision. In a subsequent execution of a branch, it may follow a pattern observed in previous executions. The *nBPAT* predictor is a local predictor that uses a recent history of execution for a branch, and attempts to find a match for it in a longer history - and can capture all patterns with a period of upto a specified length (n-bits). *nBPAT* predictors used with local branch history are called *l-nBPAT* predictors. In addition, based on the size of the patterns used, an *nBPAT* predictor can be 8BPAT (uses 8-Bit patterns), 16BPAT (uses 16-Bit patterns) etc. The *nBPAT* predictor uses a table whose every entry contains the following, 1) A shift register of length $2n$, to store the local history; and, 2) A saturating counter for selecting the best predictor - *nBPAT* or *AltPred*.

The generic *nBPAT* algorithm can be described as follows:

***nBPAT* Algorithm**:

1. For each branch $2n$ bits (h_{2n-1} to h_0) called *local history* are maintained for storing its behavior (TAKEN as 1/NOT TAKEN as 0) over the last $2n$ executions.
2. For every branch, the sequence of the n most recent executions is called the *current pattern*. The set of all substrings of length n of the *local history* is called the set of *history patterns*.
3. **nBPAT Prediction**: To predict for a given branch:-
   3.1. Match the *current pattern* (h_{n-1} to h_0) with the *history patterns* from (h_n to h_1)(h_{n+1} to h_2)...(h_{2n-1} to h_n).
   3.2. If there is a match, and the most significant bit of the saturating counter for selection is 1, then, return the next bit from the *history pattern* as the prediction.
   3.3. If there is no match, then, use *AltPred*.
4. **Updating nBPAT Entry**: To update the *nBPAT* entry for a given branch after a prediction:-
   4.1. If *AltPred* mispredicted and *nBPAT* correctly predicted, increment the saturating selection counter.
   4.2. If *AltPred* correctly predicted and *nBPAT* mispredicted, decrement the saturating selection counter.
   4.3. Update the *local history* by inserting the outcome of the branch into the *local history* shift register

The above algorithm can be explained by the following example of a 6BPAT. On the execution of a program, let the value of the program counter at an encountered branch be PC. The recent 12 bit history of this branch is stored in a 12 bit *local history* shift register in the nBPAT storage entry as shown in Fig 1. Simultaneously the recent execution history called the *current pattern* is matched with the *history patterns* to find a match as shown in the figure. The next prediction of outcome of the branch is given by the entry just after the matched pattern. On subsequent execution of the branch the *local history* is updated. By storing such a 12-bit *local history* we are able to predict all patterns of lengths upto 6-bits.

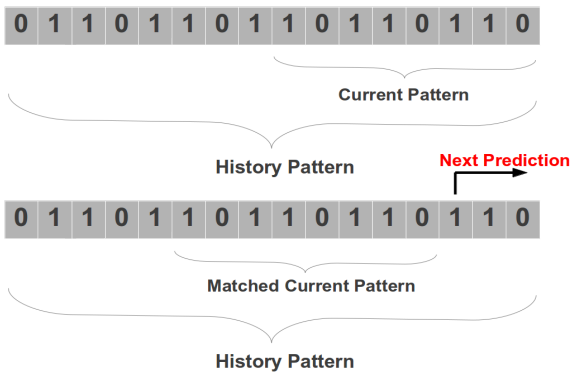

Figure 1: *nBPAT* Example

3. H-PATTERN : INTEGRATING NBPAT WITH ALTPRED

H-Pattern* using *GShare: The storage used by *nBPAT* is a tagless direct mapped table when used with *GShare* as the alternate predictor. The table is indexed with a number of the least significant bits of the program counter. After each prediction instance, *nBPAT* entries are updated as explained in the *nBPAT* algorithm. Half of the space budget is allocated to the *nBPAT* predictor.

H-Pattern* using *TAGE/ISL-TAGE: A 2-way associative partially tagged table storage is implemented for *nBPAT* when using *TAGE* or *ISL-TAGE* as the alternate predictor. In this case every *nBPAT* entry contains an additional *useful counter*, which is the measure of usefulness of a branch entry in the table. All *useful counters* are reset periodically after a fixed number of cycles, with the help of a global reset counter.

After every prediction, if there is a tag match with the *nBPAT* table, the *nBPAT* entry is updated according to the *nBPAT* algorithm. Moreover, after every prediction, the *useful counter* is updated and the *nBPAT* table entry may be reassigned to another branch. The algorithm for updating values when using *H-Pattern* with *TAGE/ISL-TAGE* is as follows.

***Updating H-Pattern with TAGE/ISL-TAGE* Algorithm**:

1. If the *TAGE* predictor MISPREDICTED and there is no tag match in *nBPAT* 2-way associative table, and, either of the 2 potential entry locations have Useful = 0, then, make Tag = [BranchTag] and Useful = [Maxmimum].
2. If the entry ALREADY exists in the *nBPAT* 2-way associative table, then,
   2.1. If *nBPAT* was not ready, OR, *nBPAT* mispredicted and *TAGE* correctly predicted, decrease useful;
   2.2. If *nBPAT* correctly predicted and *TAGE* mispredicted, increase useful;
   2.3. Update the *nBPAT* entry as described in step 4 of *nBPAT* algorithm.
   2.4. Update the TAGE/ISL-TAGE predictor.

The parameters used for the *TAGE* predictor are in accordance with those of the 8-component *TAGE* predictor as specified in [6]. For *ISL-TAGE*, as described in [7], in addition to the *TAGE* predictor, a Statistical Correlator & a Loop Predictor are incorporated. The Immediate Update Mimicker is unnecessary since the CBP4 framework already simulates immediate updates.

4. EXPERIMENTAL RESULTS

We simulated the proposed *H-Pattern* branch predictor on the CBP framework with 40 traces of *CPU SPEC2K* benchmarks. The *nBPAT* predictor was implemented in C++. The simulations were undertaken for the storage budgets of Unlimited, 32K and 4K. *H-Pattern* with *GShare* achieved a $\mathbf{3.8}$, $\mathbf{4.7}$ and $\mathbf{6.4}$ MPKI for the storage budgets of unlimited, 32K and 4K respectively. *H-Pattern* with *TAGE* achieved a $\mathbf{2.134}$, $\mathbf{2.644}$ and $\mathbf{3.712}$ MPKI for the storage budgets of unlimited, 32K and 4K respectively. *H-Pattern* with *TAGE* achieved a $\mathbf{2.058}$, $\mathbf{2.542}$ and $\mathbf{3.691}$ MPKI for the storage budgets of unlimited, 32K and 4K respectively. The accuracy of existing predictors is augmented by our proposed *nBPAT* predictor, whose predictions are highly accurate, as shown by the success rate values in table 1. The success rates given are calculated as shows:

$$SuccessRate = \frac{Number\_of\_correct\_predictions}{Number\_of\_attempted\_predictions} * 100 \quad (1)$$

Table 1: *H-Pattern* Predictor Strike Rates

Predictor Type	Success Rate Unlimited	Success Rate 32KB	Success Rate 4KB
GShare	$\mathbf{94.5\%}$	$\mathbf{93.4\%}$	$\mathbf{91.3\%}$
TAGE	$\mathbf{97.9\%}$	$\mathbf{97.2\%}$	$\mathbf{96.8\%}$
ISL-TAGE	$\mathbf{98.1\%}$	$\mathbf{97.3\%}$	$\mathbf{96.9\%}$
nBPAT	$\mathbf{99.3\%}$	$\mathbf{99.1\%}$	$\mathbf{98.6\%}$

4.1 Performance Comparison

For each candidate alternate predictor, the independent predictor was compared to the hybrid *H-Pattern* predictor, combined with AltPred. The average MPKI values for various *AltPred* predictors used independently as well as in combination with *H-Pattern* for unlimited, 32KB and 4KB storage sizes are as given in Fig. 2–4.

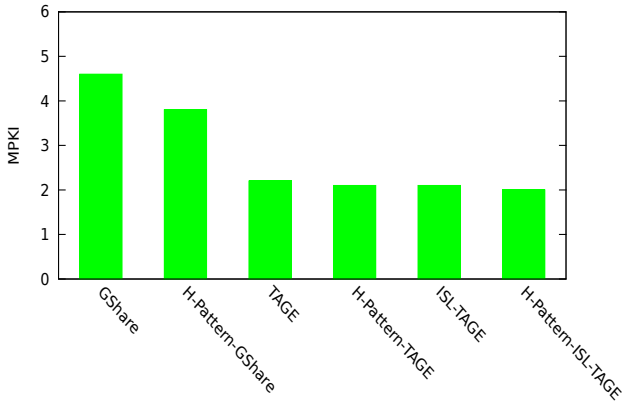

Figure 2: Performance Comparison: Unlimited Storage

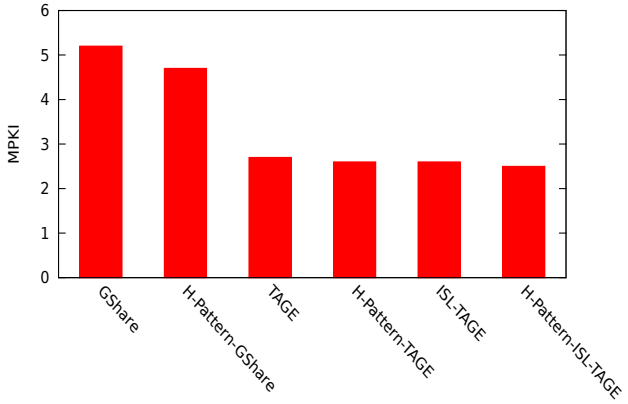

Figure 3: Performance Comparison: 32KB

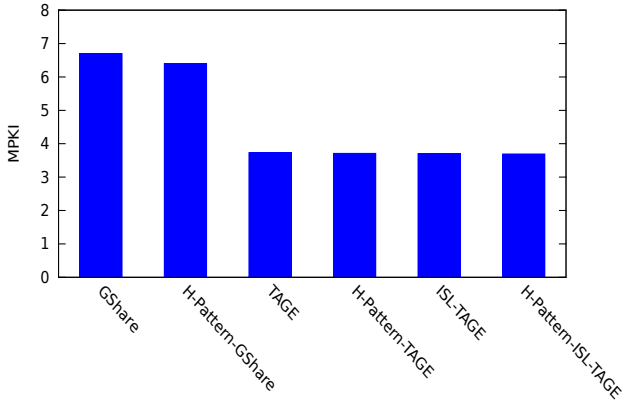

Figure 4: Performance Comparison: 4KB

4.2 Storage Budget Calculation

For each space bracket, the best performing predictors were *H-Pattern* with ISL-TAGE, which we submitted for the competition. The configuration and space budget for these submitted predictors has been shown in the tables 2–5.

The configuration of the reference 32K ISL-TAGE was obtained by halving the optimal 64K configuration specified in [7]. The 32KB *H-Pattern* with ISL-TAGE predictor was obtained by freeing some space from the reference predictor by halving the last shared table, and reducing the size of the statistical correlator and loop predictor, and accomodating a 4-BPAT predictor in that space.

The earlier paper on TAGE [6] specified an optimized set of parameters for 5-component 8-component TAGE predictors. The reference 4KB 32KB TAGE predictors used for the comparison were based on these parameters. The 4KB ISL-TAGE predictor was obtained by freeing some space from the 4KB TAGE predictor, by sharing hysteresis bits. The 4KB H-Pattern predictors based on TAGE ISL-TAGE were obtained by freeing one tag bit on every alternate table, starting from T2, and using this space to accomodate a 4-BPAT predictor. The 32KB H-Pattern with TAGE predictor was obtained by halving T6 in the reference 32K predictor, and accomodating an 8-BPAT predictor in this space.

The H-Pattern with *Gshare* predictors were obtained by halving the number of rows of the reference *Gshare*, and using half of the storage budget for a 6-BPAT predictor (for 4KB 32KB), and an 8-BPAT predictor (for unlimited).

5. CONCLUSION

We propose a novel hybrid pattern based branch predictor (*H-Pattern*) which adapts between two predictors viz *l-nBPAT* and an alternate predictor *AltPred* based on performance. *H-Pattern* with Gshare achieves a $\mathbf{3.8}$, $\mathbf{4.7}$ and $\mathbf{6.4}$ MPKI as compared to $\mathbf{4.6}$, $\mathbf{5.2}$ and $\mathbf{6.7}$ MPKI by the *GShare* predictor for unlimited, 32KB and 4KB storage budgets respectively. Implementing *H-Pattern* with *TAGE* we achieved $\mathbf{2.134}$, $\mathbf{2.644}$ and $\mathbf{3.712}$ mispredictions per kilo instructions (MPKI), compared to $\mathbf{2.177}$, $\mathbf{2.678}$ and $\mathbf{3.735}$ by *TAGE*, for the same respective budgets. *H-Pattern* with *ISL-TAGE*) achieved $\mathbf{2.058}$, $\mathbf{2.542}$ and $\mathbf{3.691}$ MPKI, as opposed to $\mathbf{2.076}$, $\mathbf{2.549}$ and $\mathbf{3.706}$ by *ISL-TAGE* for the same respective budgets. Our proposed *l-nBPAT* predictor can capture all patterns upto n bits by using a history of just $2n$ bits. The proposed predictor can be practically implemented using a low storage budget with good accuracy and low latency of calculation. Our results show that integrating *l-nBPAT* with other predictors (*AltPred*) like *GShare*, *TAGE* and *ISL-TAGE* can result in better accuracy than being used independently.

6. REFERENCES

[1] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In Proceedings of the 30th Annual International Symposium on Microarchitecture, December 1997.

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

[3] S. McFarling. Combining branch predictors. Technical Report TN-36m. Digital Westem Research Laboratory, June 1993.

[4] E. Sprangle, R. S. Chappell, M. Alsup, and Yale N. Pan. The Agree predictor: A mechanism for reducing negative branch history interference. In Proceedings of the 24th International Conference on Computer Architecture. June 1997.

[5] A. N. Eden and T. N. Mudge. The YAGS branch prediction scheme. In Proceedings of the 31st Annual ACMLEEE International Symposium on Microarchitecture, November 1998.

[6] A. Seznec and P. Michaud. A Case for Tagged Geometric History Length Branch Prediction Journal for Instruction Level Parallelism, 2008

[7] A. Seznec. A 64 Kbytes ISL-TAGE branch predictor Journal for Instruction Level Parallelism, 2011

Table 2: *H-Pattern* with *ISL-TAGE* Storage Budget Calculation

Component	**Unlimited size in bits**	**32KB size in bits**	**4KB size in bits**
Data table for *nBPAT*:			
History bits (per row)	**16**	**8**	**8**
Selector bits (per row)	**4**	**4**	**4**
Tag width (per row)	**16**	**6**	**7**
Useful bits (per row)	**4**	**3**	**3**
Total bits (per row)	**40**	**21**	**22**
Number of rows	**2097152**	**512**	**64**
A: nBPAT Tables (in bytes)	**10485760**	**1344**	**176**
B: TAGE Tables	**1288000**	**30272**	**3872**
C: Statistical Co-relator	**393216**	**640**	**80**
D: Loop predictor	**96256**	**360**	**47**
E: Global History	**256**	**256**	**16**
F: 2 12-bit buffer pointers + 2 16-bit path history counters	$\mathbf{24 + 32(bits)}$	$\mathbf{24 + 32(bits)}$	$\mathbf{24 + 32(bits)}$
G: Usefulness counter + Reset counter	$\mathbf{7 + 8(bits)}$	$\mathbf{7 + 8(bits)}$	$\mathbf{7 + 8(bits)}$
H: Random No. Generator + 2 6-bit Stat.Cor. Threshold counters	$\mathbf{32 + 12(bits)}$	$\mathbf{32 + 12(bits)}$	$\mathbf{32 + 12(bits)}$
I: nBPAT Reset counter	**2**	**2.25**	**2**
Total Storage Budget (A+B+C+D+E+F+G+H+I)	**23263504.38bytes**	**32888.625bytes**	**4209bytes**

Table 3: *ISL-TAGE* Parameters Storage Budget Calculation : Unlimited Storage

Characteristics	**T0**	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7–T9 (for each table)**	**T10–T15 (for each table)**
Predictor bits	**2**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**
Useful bits (per row)	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Tag width	**0**	**16**	**16**	**17**	**18**	**19**	**20**	**20**	**20**
No of entries	**4194304**	**262144**	**262144**	**262144**	**524288**	**524288**	**524288**	**262144**	**131072**
Total Storage Space (in bytes)	**1048576**	**655360**	**655360**	**688128**	**1441792**	**1507328**	**1572864**	**786432**	**393216**
History lengths are in a geometric series from 5 to 2000									

Table 4: *ISL-TAGE* Parameters Storage Budget Calculation : 32KB

Characteristics	**T0**	**T1–T2 (shared tables)**	**T3–T7 (shared tables)**	**T8–T13 (shared tables)**	**T14–T15 (shared tables)**
History bits	**0**	$\mathbf{3, 8}$	$\mathbf{12, 17, 33, 35, 67}$	$\mathbf{97, 138, 197, 330, 517, 1193}$	$\mathbf{1741, 1930}$
Predictor bits	**2**	**3**	**3**	**3**	**3**
Useful bits (per row)	**0**	**1**	**1**	**1**	**1**
Tag width	**0**	**8**	**11**	**13**	**14**
No of entries	$\mathbf{16384 + 4096}$	**2048**	**8192**	**4096**	**256**
Total Storage Space (in bytes)	**2560**	**3072**	**15360**	**8704**	**576**

Table 5: *ISL-TAGE* Parameters Storage Budget Calculation : 4KB

Characteristics	**T0**	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**
Predictor bits	**2**	**3**	**3**	**3**	**3**	**3**	**3**	**3**
Useful bits (per row)	**0**	**2**	**2**	**2**	**2**	**2**	**2**	**2**
Tag width	**0**	**11**	**10**	**11**	**10**	**11**	**10**	**11**
No of entries	$\mathbf{2048 + 1024(hysterisis)}$	**256**	**256**	**256**	**256**	**256**	**256**	**256**
Total Storage Space (in bytes)	**384**	**512**	**480**	**512**	**480**	**512**	**480**	**512**
History lengths are in a geometric series from 1 to 128								