Yet Another Implementation of Fast Path-Based Neural Branch Predictor

Yukun Zeng
Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77840
Email: yzeng@tamu.edu

***Abstract*—Branch prediction is an important technique used to improve instruction-level parallelism in deeply pipelined processors of modern parallel computer systems. In the last decade, a new approach of neural branch prediction was introduced and received increasing attention. Comparing to static branch prediction and some popular dynamic prediction strategies like two-level branch prediction and gshare, neural predictors generally achieve higher average performance in terms of hit rate but are also more complex in hardware implementation. In this paper, we review the branch prediction literature and present an implementation of Fast Path-Based Neural Branch Prediction, which is the simpliest neural approach that can also achieve considerable performance. The final benchmark show that our predictor outperforms gshare by 22.4% under a 8KB hardware budget.**

I. Introduction

In the instruction set of modern CPU, there are lots of basic instructions that functions differently. Through combining these basic instructions together in certain sequences, we can program the CPU to do complex computations and solve real-life problems. However, for complex applications, it's impossible that they can be solved by computing all the way to end, those applications are always involved with some if/loop logics. Thus, except for computation instructions like ADD, MULTIPLY, etc, there are also certain kinds of instructions exist to control the instruction flow and select next instruction to execute on-the-fly. Though branch instructions enable us to integrate more daily logics into computer programs, it might at the meantime break the smooth flow of instruction fetching and execution in highly parallel computer systems [1]. Such problems result in delay since the result of a branch instruction changes the location of instruction fetches and the execution of next instruction must wait until the conditional branch decisions are made. To reduce such delays, branch predictors are introduced in [1] to determine the next instruction that a branch instruction might lead to before it was executed. With branch prediction, we can start fetching, decoding and even executing the next instruction beforehand, which substantially improved the performance of modern pipelined microprocessors.

II. Related Works

Generally, the former researches conducted on Branch Prediction can be divided into two categories, static branch prediction and dynamic branch prediction. Here we present a brief review of predictors of those two types.

A. Static Branch Prediction

Static branch prediction is the earliest and simplest branch prediction strategy. As the name itself shows, such predictors produce static results, i.e., they made prediction decisions before program runtime. Classic static branch prediction strategies include but not limited to:

- Predict all branches will be taken.
- Predict that all branches with certain operation code will be taken, others not taken.
- Predict all backward branches (toward lower addresses) will be taken, while all forward branches will not be taken.

Despite their simplicity, the actual performance of those strategies is considerably good, the result accuracy ranges from 60% to more than 90% depending on different benchmarks used [1]. Afterwards, more static branch predictors are proposed, one typical example among them is static branch predictor with neural networks. Using this approach, the branch direction is predicted in compile-time by program features, its prediction results reached an 80% correct prediction rate [2].

B. Dynamic Branch Prediction

Dynamic branch prediction makes prediction decisions based on history branch taken data. Since dynamic predictors can be self-tuning in the runtime, they can provide better performance most of the time. Typical dynamic predictors include:

- Branch Target Buffer [3]: use 2-bit saturating up-down counters to collect history information for predictions.
- Two-level adaptive training branch prediction [4]: the most widely-used strategy that has been put in many real computer architecture.
- Dynamic Perceptron Branch Prediction(DPBP) [5]: first successful integration of neural networks into dynamic branch prediction.
- Path-Based Neural Branch Prediction(PBNBP) [6] and Piecewise Linear Branch Prediction(PLBP) [7]: improved versions based on perceptron approach.
- L-TAGE [8]: state-of-the-art branch prediction with high accuracy.

III. The Path-Based Neural Predictor

After careful evaluation of other branch prediction methods, I finally decided to choose PBNBP as my target. In the next two sections, I will state the reason of choosing this approach and present a review of its design.

A. Reason

Reviewed the evolution of branch predictors, I first decided to implement a dynamic predictor instead of static ones, the reasons shall be three fold:

- Static prediction strategies in general perform worse than dynamic techniques.
- Static predictors are unable to tune themselves on the fly, while the prediction strategies always vary greatly between different programs.
- The program logic of static predictors is often simple and unchallenging.

When choosing dynamic branch predictors, I first narrowed down several high-quality research papers, which are perceptron, PBNBP, piecewise linear branch prediction and L-TAGE, among which perceptron is the earliest, L-TAGE performs the best and the other two is improved strategies based on or inspired by perceptron. After consulting Prof. Jimnez, I choose Path-Based Neural Branch Prediction approach for following reasons:

- The paper of L-TAGE is too vague to understand, making it hard to implement in a short time.
- Perceptron as the precursor of dynamic neural network branch prediction is too simple and its performance might not help me win the prize.
- PLBP's requirements on memory space make it hard to satisfy the competition requirements. The size of a version of PLBP (which our professor used in a competition) is 65,789, which exceeds our project requirements [7].
- PBNBP approach has a considerably good accuracy and enjoys a low latency, while occupying a reasonable space. PBNBP is easy to implement both in program and on hardware, which means it's highly applicable.

TABLE I
Algorithm Symbol Table

Symbol	Explanation
w	weight matrix for perceptron
i	index in rows of weight matrix w
n	design parameter
v	addresses of modulo n historical predicted branches
sv	speculative version of v
r	partial sum from last prediction
sr	speculative version of r
g	shift global history register
sg	speculative version of g
y	perceptron output
h	history length
$outcome$	the branch result of real execution
θ	perceptron threshold

Algorithm 1 Branch Predict

function PREDICT(address)
  $i = address \mod n$
  $shift(sv, i)$
  $update.v = sv$
  $update.g = sg$
  $y = w[i, 0] + sr[h]$
  $prediction = (y >= 0)$
  for $j = 1 \rightarrow h$ **do**
    $k = h - j$
    if $prediction = taken$ **then**
      $sr[k + 1] = sr[k] + w[i, j]$
    else
      $sr[k + 1] = sr[k] - w[i, j]$
    end if
  end for
  $sr[0] = 0$
  $shift(sg, prediction)$
  $update.i = i$
  $update.y = y$
  $update.prediction = prediction$
  return $update$
end function

B. Design

Before introducing the design of PBNBP, we first review the concept of perceptron learning [9], symbols used in the following paper is explained in Table I. The perceptron is a binary classifier based on supervised machine learning, it can classify a series of vector input (such as $< x_1, x_2, \ldots, x_n >$) into two categories, e.g., taken or not taken. To adapt it to our branch prediction application, we use x_i in those vectors to represent the bits of a global branch history shift register. The mechanism of a perceptron is shown in Fig. 1.

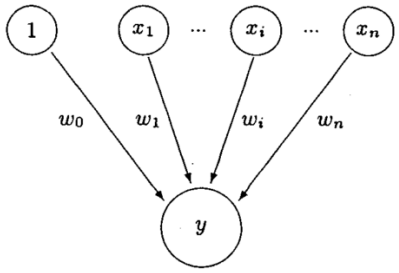

Fig. 1. Mechanism of a Perceptron

The output of a perceptron is computed as:

$$y = w_0 + \sum_{i=1}^{n} x_i w_i \tag{1}$$

Similar to Perceptron Branch Predictor(PBP), PBNBP is also based on perceptron, the simplest neural networks approach. Since dynamic branch predictors need to make decisions based

on branch taken history, PBNBP keeps a global history shift register that records the outcomes of branches as they are executed or speculatively as they are predicted. In order to do this, it keeps an $n \times (h+1)$ matrix $w_{n\times(h+1)}$, in which every element is an 8-bit bytes integer weights, while n is a design parameter. Each row of the matrix is an (h+1)-length weights vector, each vector stores the weights of one perceptron. The first weight elements of any row is known as the biased weight. The Boolean vector $g[1\ldots h] \in \{1\ldots h\} \times \{taken, not\_taken\}$ represents the global history shift register [5]. Different from PBP, only the bias weight in PBNBP is used to predict the current branch. The bias weight is added to a taken total of last h branches, with each summand added during the processing of a previous branch [6].

C. Algorithm

Different from the original paper, we propose our detailed algorithm for both prediction and perceptron updating. These two algorithms are proposed in Algo 1 and Algo 2, respectively. Symbols in our algorithm are explained in Table I. Since the algorithm listed below is in detail and self-explanatory, we skipped the description part here.

Algorithm 2 PREDICTOR UPDATE

function UPDATE(update, outcome)
  $prediction = update.prediction$
  $y = update.y$
  $i = update.i$
  for $j = 1 \rightarrow h$ **do**
    $k = h - j$
    if $outcome == taken$ **then**
      $r[k+1] = r[k] + w[i,j]$
    else
      $r[k+1] = r[k] - w[i,j]$
    end if
  end for
  $r[0] = 0$
  $shift(g, outcome)$
  $shift(v, i)$
  if $outcome \neq prediction$ **then**
    $sr = r$
    $sg = g$
    $sv = v$
  end if
  if $outcome \neq prediction \| abs(y) < \theta$ **then**
    $p\_update(w[i,0], outcome)$
    for $j = 1 \rightarrow h$ **do**
      $k = update.v[j]$
      $flag = (outcome == update.h[j])$
      $w[k,j] = p\_update(w[k,j], flag)$
    end for
  end if
end function

IV. EVALUATION

A. Misprediction Recovery

Similar to the original paper, our implementation also includes a misprediction recovery mechanism to prevent critical errors caused by misprediction. As mentioned in Table, there are several symbols with prefix "s", meaning speculative, those are the speculative version of historical predicted branch address array, historical prediction partial sum array, etc. What we do in the algorithms using those speculative arrays is as follows:

- After the branch prediction, the speculative branch outcome, global history register and partial sums are recorded in speculative arrays.
- When doing branch predictions before historical outcomes are available, i.e., historical branches executed, we use speculative arrays as historical records as input.
- If certain branches finished execution, we check if we do branch prediction wrong, and update speculative arrays to non-speculative results accordingly.

Through those approaches, the resilience of our branch predictor is guaranteed and the delay of waiting historical branches to finish is greatly reduced.

B. Latency

Comparing to the Java implementation [10] of the path-based neural predictor, we exploit several features of C++ to reduce the prediction latency:

- **memset:** The initialization process of the neural predictor is time-consuming, especially when we are using a long history length for high prediction accuracy. Different from the loop-assign process in the Java implementation, we exploit memset in C++ which is much faster in setting the init value of weight matrix and other arrays.
- **memcpy:** Noticing a large portion of shift and array copy operations conducted in the predictor, we use memcpy to perform those so that the performance is optimized.

Hardware Budget	fixed length path	2Bc-*gskew*	global/ local	path-based neural
1 KB	10	10	25/9	13
2 KB	10	10	31/11	18
4 KB	12	10	34/12	20
8 KB	15	11	34/12	32
16 KB	20	14	38/14	34
32 KB	20	15	40/14	34
64 KB	20	16	50/18	37

Fig. 2. Tuned History Length

C. Hardware Budget

The major memory consumption of path based neural predictor comes from its weight matrix, which contains in total $n \times (h + 1)$ entries. As a common practice, we use 8 bits for one entry in the matrix, which means the weight range in our implementation is $[-128, 127]$. Under this setting, we referred to Fig. 2 in [6] and discover that under 8KB budget the best choice of history length is 32(which also informs us that $n = 256$). Therefore, the hardware budget is set according to the original paper and certified to meet the project requirements.

D. Misprediction Rate

Fig. shows the misprediction rates for our branch predictor ranging over hardware budgets from 1KB to 64 KB based on the provided microarchitectural instrastructure. Clearly, our predictor performs very well since even the least 1KB budget (history length 13) using our predictor can produce a much better result than 32,768-entry *gshare* with history length of 15.

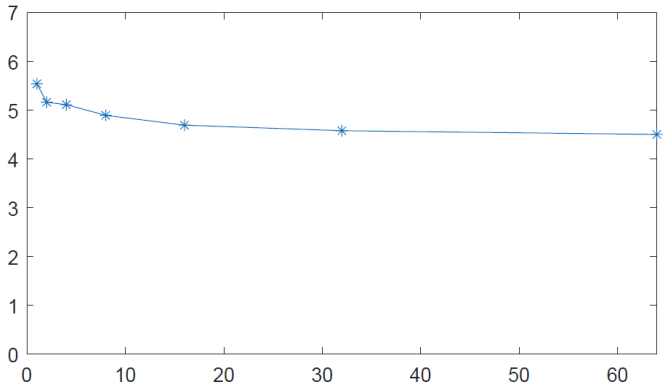

Fig. 3. Misprediction Rate Per Hardware Budget

V. Conclusion

We have presented a new implementation of path-based neural branch predictor. The predictor is originally proposed in [6] as an improvement of initial neural branch predictor in [5], its Java implementation can be found at [10]. The programming language is C++ and we exploited several C++ features to improve the predictor's latency. We also presented a comprehensive test of the branch predictor built on different hardware budget, experimental results showed that it achieves a miss rate of 4.893% under the project specified 8KB budget, which is 22.4% lower than *gshare*'s 6.305%. Moreover, with the increase of hardware budget, it can achieve even better performance, e.g., 4.578% miss rate under 32KB budget.

References

[1] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press, 1981, pp. 135–148.

[2] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 1, pp. 188–222, 1997.

[3] J. K. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," in *Instruction-level parallel processors*. IEEE Computer Society Press, 1995, pp. 83–99.

[4] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th annual international symposium on Microarchitecture*. ACM, 1991, pp. 51–61.

[5] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE, 2001, pp. 197–206.

[6] D. A. Jiménez, "Fast path-based neural branch prediction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 243–252.

[7] D. Jiménez, "Idealized piecewise linear branch prediction," *Journal of Instruction-Level Parallelism*, vol. 7, pp. 1–11, 2005.

[8] A. Seznec, "The l-tage predictor," *Journal of Instruction Level Parallelism*, vol. 9, 2007.

[9] H.-D. Block, "The perceptron: A model for brain functioning. i," *Reviews of Modern Physics*, vol. 34, no. 1, p. 123, 1962.

[10] D. A. Jiménez, "Java Fast Path-Based Neural Branch Predictor," http://hpca23.cse.tamu.edu/taco/public-bp/index.html.