VPC-based Indirect Branch Predictor

Ankur Roy Chowdhury
Department of Computer Science
Texas A&M University
ankurrc@tamu.edu

***Abstract*— With the advent of deeply pipelined processors, branch prediction has become an important aspect towards maintaining high performance in computer architecture. Moreover, modern Object-Oriented languages continually rely on concepts like *polymorphism*, employing the use of dynamically-dispatched functions: which are implemented as indirect branch/calls. Thus, it is evident that indirect-branch prediction is a relevant area for improvement. In this paper, we implement the *VPC*[1] algorithm for indirect branch prediction. The key idea behind VPC is to treat a single indirect branch as a set of *"virtual" conditional branches*. To that extent, it reuses the existing conditional branch prediction algorithm and re-purposes it for indirect branch prediction.**

***Index Terms*— VPC, indirect branch prediction, perceptron branch predictor**

I. INTRODUCTION

Object-oriented programs are becoming more common nowadays. OOP languages support polymorphism, which helps develop scalable and maintainable software. To support polymorphism, modern languages include dynamically-dispatched function calls (i.e. virtual functions) whose targets are not known until run-time because they depend on the dynamic type of the object on which the function is called. Virtual function calls are usually implemented using indirect branch/call instructions in the instruction set architecture.

Unfortunately, however, an indirect branch instruction is more costly on processor performance because predicting an indirect branch is an N-ary decison, requiring the prediction of the target address instead of the prediction of the branch direction. Direction prediction is inherently simpler because it is a binary decision as the branch direction can take only two values (taken or not-taken).

Thus, it is evident that indirect branches could pose as a limiter to performance. VPC is an indirect branch prediction algorithm that re-purposes the existing branch predictor of a processor to predict indirect branches. VPC eliminates the need for complex branch prediction hardware, thereby increasing efficiency while keeping costs low.

The rest of the paper is organized as follows:

- **Section 2** discusses the conditional branch predictor used in this paper and why it was chosen
- **Section 3** talks about the VPC algorithm and how it was adapted for the chosen conditional branch predictor
- **Section 4** details the infrastructure and talks about the results
- **Section 5** concludes the paper

II. MERGED PATH AND GSHARE INDEXED PERCEPTRON PREDICTOR

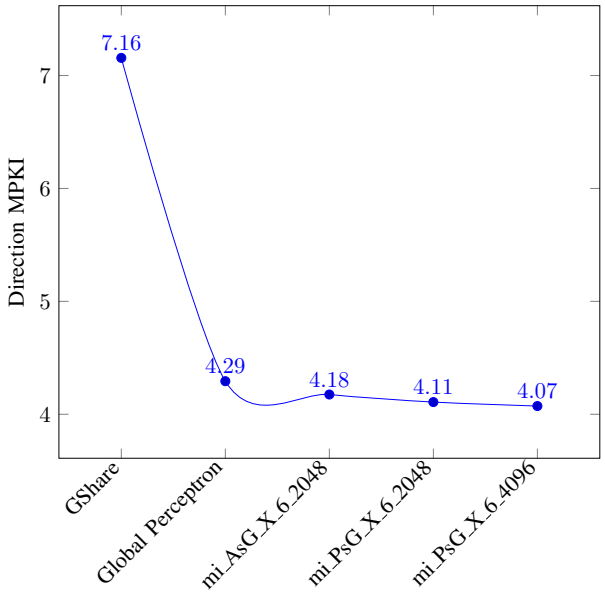

Fig. 1. Comparison of DMPKI for different prediction algorithms

As the VPC algorithm relies on a conditional branch predictor, it is crucial to choose an efficient conditional branch predictor.

A good way to predict branch direction is to correlate it with branch history. The GShare predictor employing 2-bit counters, is a good example. However, even though GShare achieves modest prediction rates, the size of branch history is severely limited due to it's exponential relation with the 2-bit counters.

Neural branch prediction, first implemented as Perceptron predictors[2], improves upon this correlation by employing a linear relation between branch history and the weights required for branch direction prediction. However, even though the Perceptron predictor improves prediction rates by exploiting longer branch histories, it is computationally expensive to train. Moreover, Perceptrons are a binary classifier that are only able to classify linearly-separable quantities. Thus, another drawback of the Perceptron predictor is it's inability to classify/predict inseparable branch history patterns.

To alleviate these problems, one can combine multiple ways to index the perceptron weights, whilst maintaining

a non-linear relation with it's branch history[3]. As such, the various schemes to index the weights can be grouped into 'single' indexed and 'multi' indexed, wherein a single index perceptron uses the same index to choose it's weights from the weight table. A multi index perceptron has it's weights skewed in the weight table and the indices are picked based on the correlation between histories, and/or addresses, and/or paths. To introduce a uniform way of naming the way these perceptrons are implemented, there can be three parts to it:

- whether they use one or multiple indices to fetch all their weights
- what type of information they use to fetch their weights
- what type of information they use for their input vector

A_B_C:
A = *si*: single index, *mi*: multi index
B and C = A: address, G: global history, P: path history, L: local history, X: nothing
B and C = YsZ: Y $\bigoplus$ Z

Therefore, in accordance to the above notation, **si_A_G** would mean the perceptron is single indexed, uses the address to find the index and uses it's global history as the input vector to the perceptron. Essentially, this is the Global Perceptron discussed a few paragraphs before. The **mi_AsG_X** predictor uses the correlation between the address and the global history to find the indices to it's weights and simply uses them without any input vector. The **mi_PsG_X** predictor uses the correlation between the path and the global history to find it's weights. Further more, **mi_PsG_X_6_4096** predictor is the same as above, but has 6 weights (excluding bias) per perceptron and has 4096 weights per weight table.

Algorithm 1: mi_PsG_X Prediction Algorithm

```
Input: past_pc[], history
Output: prediction:{-1,1}
index[0] := past_pc[0];
out := W[0, index[0]];
for j ← 1 to h do
    segment := (history ⊕ path) & mask;
    index[j] := (segment ⊕ address);
    out := out + W[j, index[j]];
end
if out < 0 then
    prediction := -1;
else
    prediction := 1;
end
```

The comparison between all the predictors discussed above, in terms of direction MPKI, is given in figure 1.

As evidenced, the **mi_PsG_X_6_4096** has the best conditional MPKI numbers. Thus, it was chosen to be the conditional branch predictor in our VPC-implementation. It's advantages are:

Algorithm 2: mi_PsG_X Training Algorithm

```
Input: index[], out, prediction, outcome
if prediction ≠ outcome or |out| ≤ θ then
    for j ← 0 to h do
        W[j, index[j]] := W[j, index[j]] + outcome;
    end
else
end
```

- less than linear relation between global history and weights, enabling the use of longer history lengths for prediction
- correlating path history with global history, allowing the predictor to somewhat overcome it's limitation in terms of linear-inseparability of branch history patterns
- fewer weights, making it computationally less expensive

Algorithms 1 and 2 detail the prediction and training algorithms for mi_PsG_X_6_4096, respectively.

III. Virtual Program Counter Prediction

A. Algorithms

Prediction: A Virtual Program Counter (VPC) predictor treats a single indirect branch as multiple conditional branches (virtual branches) in hardware for prediction purposes. Conceptually, each virtual branch has its own unique target address, and the target address is stored in the BTB with a unique virtual PC. The processor uses the outcome of the existing conditional branch predictor, in our case *mi_PsG_X_6_4096*, to predict each virtual branch direction.

- The processor accesses the conditional branch predictor with the virtual PC address, Virtual GHR & the Virtual Path History, and the BTB with the virtual PC address of a virtual branch.
- If the prediction for the virtual branch is "taken," the target address provided by the BTB is predicted as the next fetch address (i.e. the predicted target of the indirect branch).
- If the prediction of the virtual branch is "not-taken," the processor moves on to the next virtual branch: it tries a conditional branch prediction again with a different virtual PC, VGHR and VPath.
- The processor repeats this process until the conditional branch predictor predicts a virtual branch as taken.
- VPC prediction stops if none of the virtual branches is predicted as taken after a limited number of virtual branch predictions.

Algorithm 3 details the above.

Training: The training algorithm has 2 variants, when the prediction was correct and when the prediction was incorrect. Algorithms 4 and 5 detail them respectively. The key functions of the training algorithm are:

- to update the direction predictor as not-taken for the virtual branches that have the wrong target and to update

Algorithm 3: VPC Training Algorithm (adapted for mi_PsG_X)

```
Input: past_pc[], history, path
Output: target
iter := 1;
VPCA := past_pc[0];
done := false;
while !done do
    predicted_target := access_btb(VPCA);
    predicted_direction := access_conditional_BP(VPCA,
      VGHR, VPATH);
    if predicted_target & predicted_direction := TAKEN
      then
        target := predicted_target;
        done := true;
    else if !predicted_target or iter ≥ MAX_ITER then
        btb_miss := true;
        done := true;
    VPCA := HASH(PC, iter);
    VGHR := LeftShift(VGHR);
    VPATH := (VPATH << 4) | (VPCA & 0xF);
    iter++ ;
end
```

it as taken for the virtual branch, if any, that has the correct target.

- to update the replacement policy bits of the correct target in the BTB (if the correct target exists in the BTB)
- to insert the correct target address into the BTB (if the correct target does not exist in the BTB)

When the prediction is correct, the algorithm loops through all the iteration till the *predicted_iter*. The conditional predictor is trained on 'not taken' for all the iterations expect for the last one. On the last iteration, i.e. the iteration that predicted the outcome, the conditional predictor is trained on 'taken'. The training algorithm for *'update_conditional_BP'* is the same as Algorithm 2.

Algorithm 4 details the above.

When the predictor is incorrect about it's target prediction it can lead to 2 cases:

- *wrong target case:* The correct target exists corresponding to one of the VPCAs, but the wrong one was predicted. The algorithm loops through all the iterations, for each VPCA. In each iteration it compares the corresponding target with the correct target. If a match is found, that particular iteration trains the conditional BP on 'taken'. Also, it's replacement policy bits are updated. For all other iterations, the conditional BP is trained on 'not taken'.
- *no target case:* The correct target does not exist in the BTB. This means the correct target needs to be inserted in the BTB for a particular VPCA. The choice of the VPCA depends on whether or not there was a BTB miss. If there was a BTB miss, the VPCA that caused the miss is chosen to be the iteration for training the conditional BP. If there was no BTB miss, the VPCA is chosen by picking the lowest LFU value of all the iterations corresponding to a VPCA. The iteration is then trained and the replacement policy bis are updated.

Algorithm 4: VPC Training Algorithm when prediction is correct (adapted for mi_PsG_X)

```
Input: past_pc[]
iter := 1;
VPCA := past_pc[0];
done := false;
while iter < predicted_iter do
    if iter = predicted_iter then
        update_conditional_BP(TAKEN);
        update_replacement_policy(VPCA);
    else if !predicted_target or iter ≥ MAX_ITER then
        update_conditional_BP(NOT-TAKEN);
    VPCA := HASH(PC, iter);
    iter++ ;
end
```

Algorithm 5 details the above.

The thing to note in case of the VPCA algorithms is that while being adapted for the chosen conditional BP, the use of VPCA, VGHR and VPath differs in the prediction and training algorithms. In the prediction algorithm, the prediction is performed using VPCA, VGHR and VPath. However, when we train the conditional predictor, we only need the VPCA, because the rest of the information correlating the VGHR and the VPath is already 'embedded' into the weight indices.

IV. Results

A branch prediction simulator was implemented on the CBP2 (version 3) infrastructure. The infrastructure includes the reading of trace files, line by line. Each line details the kind of instruction, if the instruction is a conditional branch, it gives the it's direction. If the branch is a indirect branch, it gives it's target.

The infrastructure has two interfaces, one to *predict* and one to *update* the predictor. The *predict* interface tries to predict the target for a given indirect branch instruction. Whereas, the *update* interface works on the previous prediction by training the predictor on correct and incorrect predictions.

Running the simulator on a given set of 71 traces, the predictor yielded the result of **0.711 IMPKI**. The IMPKI for the baseline BTB-based indirect predictor was 3.210. Thus, there was an improvement of 77.85% over the baseline.

V. Conclusion

In this paper an indirect branch predictor was implemented based on the VPC algorithm. For it's implementation, different conditional branch predictors were considered based

Algorithm 5: VPC Training Algorithm when prediction is incorrect (adapted for mi_PsG_X)

```
Input: past_pc[]
iter := 1;
VPCA := past_pc[0];
found_correct_target := false;
while iter < MAX_ITER and found_correct_target =
 false do
    predicted_target := access_BTB(VPCA);
    if predicted_target = CORRECT_TARGET then
        update_conditional_BP(TAKEN);
        update_replacement_policy(VPCA);
        found_correct_target := true;
    else if !predicted_target then
        update_conditional_BP(NOT-TAKEN);
    VPCA := HASH(PC, iter);
    iter++ ;
end
/* no-target case                          */
if found_correct_target = false then
    if btb_miss then
        VPCA := HASH(past_pc[0], predicted_iter);
    else
        VPCA := HASH(past_pc[0],
          min(LFU_counter_index));
    insert_BTB(VPCA, CORRECT_TARGET);
    update_conditional_BP(TAKEN);
    update_replacement_policy(VPCA);
```

on their resource consumption and prediction rates. The VPC algorithm was then adapted to work using the chosen conditional BP. The final implementation was then run as a simulator on a set of given traces.

The VPC predictor exceeded the accuracy of the baseline predictor by a large margin.

ACKNOWLEDGMENT

I would like to thank Prof. Daniel A. Jiménez for teaching me about branch prediction; and encouraging me to explore modern branch prediction techniques by organizing a project-based contest on indirect branch prediction.

REFERENCES

[1] Kim, H., Joao, J. A., Mutlu, O., Lee, C. J., Patt, Y. N., and Cohn, R. (2007). VPC prediction. *ACM SIGARCH Computer Architecture News*, 35(2), 424. doi:10.1145/1273440.1250715

[2] Jiménez, D. A., and Lin, C. (2001). Dynamic branch prediction with perceptrons. *HPCA-7*.

[3] Jiménez, D. A., and Lin, C. (2002). Neural Methods for Dynamic Branch Prediction. *ACM Transactions on Computer Systems*, Vol. 20, 369–397.

[4] Tarjan, D., and Skadron, K. (2005). Merging path and gshare indexing in perceptron branch prediction. *ACM Transactions on Architecture and Code Optimization*, 2(3), 280-300. doi:10.1145/1089008.1089011