Indirect Branch Prediction using Virtual Program Counter Prediction

Gulshan Mandhyan, Department of Electrical and Computer Engineering, Texas A&M University, College Station, Texas

***Abstract*— With the advent of object oriented programming languages like C++, Java, and C#, the problem of indirect branches is becoming increasingly significant. The most common example is that of the usage of Virtual functions whose targets are known only during run-time because they depend on the dynamic type of the object on which the function is called. Such indirect branch instructions impose a serious limit on the processor performance because predicting an indirect branch is more difficult than predicting a conditional branch as it requires the prediction of the target address instead of the prediction of the branch direction.**

This paper discusses the implementation of a Virtual Program Counter(VPC) Predictor using a Hash Perceptron Conditional Branch predictor. The main idea of VPC prediction is that it treats a single indirect branch as multiple virtual conditional branches. The major advantage of this mechanism is that it uses the existing conditional branch prediction hardware to predict the targets of indirect branches, without requiring any program transformation or compiler support.

I. Introduction

Current processors speculatively execute instructions beyond an unresolved branch using branch prediction for higher performance. But if the branch is mispredicted a lot of this speculative work is thrown away and the execution must restart right after the branch instruction. With wide-issue, deeply pipelined processors, this misprediction penalty increases and hence it is critical for processors to have accurate branch predictors.

The more common conditional branches can be predicted with a very good hit rate as it only involves predicting the direction of the branch. Whereas indirect branch prediction involves the prediction of the target address which can change with every dynamic instance of that branch. With the increase use of object oriented programming languages, the indirect branch misprediction has become critical factor in limiting the processor performance.

The initial work in indirect branch prediction included the use of Branch Target Buffer(BTB), which stores the target of the branch in case of misprediction and then uses the same to predict the target of the next instance of the branch. This scheme does not work well for indirect branches as the targets dynamically change.

Later PY Chang[2] proposed a new mechanism which uses the branch history along with the branch address to index into a target cache, where each cache entry contains a target address. This improved the prediction accuracy when compared to the BTB based predictor but the increased the complexity on the processor end because of the inclusion of separate storage structure just for the indirect branches.

More recent work by Seznec and Michaud[3], proposes ITTAGE predictor that is an extension of the aggressive TAGE predictor. However this also requires large separate storage just for indirect branches and complex logic to implement the Prediction by Partial Matching Algorithm.

The Virtual Program Counter Predictor[1] proposed by H. Kim, J.A Joao, C.J Lee, Y.N Patt, O. Mutlu and R. Cohn is the only algorithm which uses the already existing conditional branch predictor hardware for indirect branch prediction. It treats an indirect branch as a sequence of multiple conditional branches, called virtual branches. Each virtual branch ideally has its own unique target address that is stored in the BTB. Using this algorithm in combination with an accurate conditional branch predictor improves the prediction accuracy significantly without the need for having separate large storage structures for indirect branch prediction.

II. Virtual program Counter(VPC) prediction

A VPC predictor treats an indirect branch as a sequence of multiple conditional branches, called virtual branches. Each virtual branch is predicted using a conditional branch predictor hardware consisting of the direction predictor and the BTB. If the virtual branch is predicted to be not-taken, the predictor iterates to predict the next virtual branch in the sequence. If the virtual branch is predicted to be taken, then the target associated with the virtual branch in the BTB is used as the next fetch address, completing the prediction of the indirect branch. Like every Branch prediction algorithm, VPC algorithm also can also be divided into two parts namely: Prediction algorithm and Training algorithm

A. Prediction Algorithm

Each Virtual branch needs to address a different location in the conditional branch predictor and the BTB. For this each virtual branch needs to have a unique virtual PC address(VPCA) and a virtual global history register(VGHR) to be given as an input to the conditional branch predictor. The initial value of VPCA is taken as the original PC address itself and similarly the initial value of VGHR is taken as the global history(GHR) itself.

For every iteration we check if there is a BTB entry for the current VPCA. If there is a BTB miss for the current VPCA we simply stall until the branch target is known. If not then we send the VPCA and the VGHR to the conditional branch predictor(CBP). If the CBP predicts the direction of the VPCA as taken, then the target associated with that VPCA in the BTB is predicted as the next PC to be fetched and the predicted iterator(*pred_iter*) is stored. Else we go ahead repeat the same procedure for the next VPCA and VGHR. To create a unique VPCA for an iteration of an indirect branch we hash the original PC address with a randomized constant value that is specific for that iteration. The VGHR is simply left shifted by 1 for every iteration. The maximum limit of this iterator is set to value MAX_ITER. This means that if the none of the VPCA is predicted as taken for a MAX_ITER number of iterations, then we simply stall until the branch target is known.

B. Training Algorithm

Once the target is known for the branch, the VPC predictor needs to be trained in order improve the prediction based on whether the predicted target was correct or not. This involves identifying the different cases for training the predictor as listed below:

1. *The predicted target was correct:* In this case we update our conditional branch for all correct predictions. For this case we iterate through the VPCA and VGHR until the *pred_iter* iteration value. For all the virtual branches before the predicted one we train the CBP to strongly predict those branches as Not Taken. For the predicted virtual branch we train the CBP to strongly predict this branch as Taken and also update the frequently used counter value(lfu_count) for this virtual branch by 1. We will see below how this counter is used for the replacement policy for the insertion of new target in the BTB.
2. *The predicted target was incorrect:* Here we can have two cases when a target was mispredicted, one is that the one of the virtual branches has the correct target address stored in BTB and the other one is none of the virtual branches have the correct target address stored in the BTB.

   We look through all our virtual branches and check if the correct target address is already in the BTB entry associated with any of the virtual branches. We basically iterate through all the virtual branches until the target is found. If the target address for the virtual branch is same as the correct address for the indirect branch we train our CBP to predict this virtual branch as taken and also increment the *lfu_count* by 1 for this target address. Otherwise we train our CBP to predict the virtual branch as Not-taken.

   At the same time we also find the virtual branch that is least frequently used. This is used to identify which target for a virtual branch needs to be replaced if the correct target was not found and hence needs to be inserted and none of the virtual branches missed in the BTB. If any of the virtual branches miss in the BTB then it simply means that the correct target is not found and we can simply insert the correct target address in the virtual branch that missed in the BTB. When the virtual branch with the correct target address is found we simply stop the training.

III. Perceptron Branch Predictor

The VPC predictor relies heavily on the prediction accuracy of the conditional branch predictor. Hence it becomes critical to choose a highly accurate conditional branch predictor. The perceptron branch predictor proposed by D.A. Jiménez[4] proves to be more accurate than the existing dynamic global branch predictors. It is proven to improve misprediction rates on the SPEC 2000 integer benchmarks by 10.1% for a 4K byte hardware budget. Also the hardware resources for this method scale linearly with the history length which is much less when compared to other dynamic schemes which require exponential resources.

A. Perceptron Prediction Algorithm

A perceptron branch predictor basically learns the correlation between the branch outcome histories and the behavior of the current branch. A perceptron is basically represented by a vector whose elements are weights 'w_i'. The weights are signed integers which are learned during the training of a perceptron. The output 'y' is the dot product of the input 'x_i' and weights 'w_i'.

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

The weight w0 is called the bias as it is independent of the input, that means in our case it is independent of the branch outcome history. The input 'x_i' are represented as -1 or +1 based on whether the branch outcome history corresponds to Not Taken or Taken respectively. The Prediction algorithm predicts the branch to be Not Taken if the output 'y' is negative, else it predicts the branch as Taken.

B. Perceptron Training Algorithm

Once the branch outcome is known, the perceptron can be trained based on the outcome. The training algorithm basically adapts the weights of the perceptron based on the global branch history and the outcome of the current branch. The training algorithm basically looks as the following:

```
if $\text{sign}(y_{out}) \neq t$ or $|y_{out}| \leq \theta$ then
    for $i := 0$ to $n$ do
        $w_i := w_i + tx_i$
    end for
end if
```

Here θ is the threshold for the training algorithm to define a point of convergence for the training algorithm. Based on the experiments done by D.A. Jiménez, there is an interesting relationship between history length 'h' and the best threshold for the perceptron for the given history length, i.e. $\theta = 1.93h+14$. The threshold θ is used to decide how many bits are needed for each of the weights. The bits required for each weight would be $\log_2 \theta$ plus 1 since weights are signed integers.

IV. COMMON ISSUES

The first issue was with choosing of the hash function for the creation of unique VPCA for every iteration. Since the value needs to be a randomized constant for every iteration. We need a random function generator that takes my iterator as an input. For this reason I chose a Knuth multiplicative hash function implementation.

Secondly we need to decide the value of MAX_ITER for the VPC predictor. The value of MAX_ITER was increased from 2 to 16. It was found the beyond the MAX_ITER value of 12 the performance rather reduces for few applications such as perlbench due to increase in conflict misses in BTB and longer times taken for correct prediction. Third is that we need to decide the initial value of the bias(w_0) and the rest of the weights for the perceptron. After experimenting with multiple values it was analyzed that a starting value of 0 for all the weights is enough to start the prediction.

Fourth was with the implementation of the LFU replacement algorithm for the BTB. The simple implementation of maintaining a counter for each entry in BTB works perfectly fine for our predictor. The counter is updated whenever the correct target of the branch is found in the BTB. Note this does not mean that the correct target is predicted by our predictor. Next is deciding the size of the counter, an 8-bit saturating counter provides a reasonably good prediction for our predictor.

Fifth was choosing the global history length and the threshold for the Perceptron. Based on the relationship between threshold and history length discussed above($\theta = 1.93h+14$). For a history length of 64, 9 bits would be required for each weight. For our implementation we use a History length of 59 and based on the relationship above a θ=127. This allows us to use 8-bits(1 byte) for each weight and makes the algorithm less complex as it is easy to extract information byte by byte rather than 9-bits at a time.

V. EXPERIMENTATION METHODOLOGY

The infrastructure for the Branch prediction along with the traces were provided in class by professor D.A. Jiménez. The file containing the default branch prediction algorithm had to be modified to implement the VPC predictor. Each of the distributed trace files represented the branches encountered during the execution of from 100 million to over 1 billion instructions from the corresponding benchmark. The traces included branches executed during the execution of benchmark code, library code, and system activity such as page faults.

The code was compiled with the g++ 4.8.5 version with O3 compiler optimization. The script provided in the infrastructure runs our Branch predictor on all the traces and reports the Indirect MPKI(Mispredictions per Kilo Instructions) for every trace and also the average Indirect MPKI for all the traces.

VI. RESULTS

Initially the VPC predictor was implemented with the simple gshare conditional branch predictor to verify the performance of VPC algorithm in general. On running this initial implementation over the above benchmark traces an average indirect MPKI of 1.17 was achieved. This was a very good performance indicator of the VPC algorithm when compared to the conventional BTB-based predictor. As proved already for VPC predictor the performance improves in general with the increase in the value of MAX_ITER from 2 to 12. Increasing the value of MAX_ITER beyond 12 reduces the performance on few applications such as perlbench due to an increase in the conflict misses in the BTB and longer times taken for correct prediction.

On implementing the VPC predictor with the Perceptron Conditional branch predictor a performance improvement of 40.6 percent is achieved compared to the VPC predictor with a gshare predictor. For the implementation of VPC predictor an average Indirect MPKI of **0.693** is achieved. The perceptron conditional branch predictor implemented uses a Branch History(h) of length 59 bits and 1021 entry perceptron. Each entry of a Perceptron contains 60(h+1) weights of 8-bits each. A 16K entry BTB containing 4 bytes of target address and 1 byte of replacement policy bits is used along with conditional branch predictor.

VII. CONCLUSION

The major advantage of the VPC predictor is that it achieves performance provided by other indirect branch predictors that require significant storage and complexity. Since the VPC predictor performance relies heavily on the accuracy of the conditional branch predictor, its performance can be further improved in the future based on the advancement in the conditional branch prediction algorithm itself.

REFERENCES

[1] Kim H, Joao J, Mutlu O, Lee C, Patt Y, Cohn R. Virtual program counter (VPC) prediction: Very lowcost indirect branch prediction using conditional branch prediction hard-ware. IEEE Trans. Computers, 2009, 58(9): 1153–1170

[2] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In ISCA-24, 1997.

[3] A. Seznec and P. Michaud, "A Case for (Partially) Tagged Geometric History Length Branch Prediction," J. Instruction-Level Parallelism (JILP), vol. 8, Feb. 2006.

[4] D.A. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," Proc. Seventh Int'l Symp. High Performance Computer Architecture (HPCA '00), 2001.