Implementing VPC with bi-mode predictor for studying indirect branch prediction

Ganesh Prabhu Sathyanarayana
Electrical and Computer Engineering Department
Texas A&M university
College Station, United States of America
UIN: 127004441

***Abstract*— To implement an indirect branch predictor, Virtual Program Counter (VPC) technique was chosen because it provides effective results with minimal additional hardware. The VPC prediction technique uses an existing conditional branch predictor, in this case it is the bi-mode predictor. The bi-mode predictor was chosen because it is an effective dynamic conditional branch predictor that reduces the number of interferences in pattern history table (PHT) accesses, thereby reducing the overall misprediction rate. The project was successful with a low misprediction rate of 0.7MPKI for indirect branches. The project helped understand the concepts of many conditional branch predictors, evolution of branch predictors and indirect branch prediction using VPC.**

Keywords-VPC, bi-mode predictor, MPKI, vitual branch, direction predictor, choice predictor.

I. INTRODUCTION

The control hazards caused by indirect branches is an interesting area of research in the field of computer architecture. An effective solution with use of minimal hardware, for mitigating these hazards was proposed in [1]. The Virtual Program Counter (VPC) indirect branch predictor converts the indirect branch prediction problem to a conditional branch prediction problem. There has been extensive development with conditional branch predictors, to effectively tackle the problem. The reason for implementing VPC as an indirect branch predictor was that it transforms the problem such that existing effective solutions may be implemented. Although most modern programs today have up to 25% indirect branches on average a cost-effective solution was hard to come across [1]. Figure 1 shows the percentage of indirect branches in modern applications VPC predictor assumes the use of a history-based conditional predictor which is quite common today. History-based predictor are extremely effective as they exploit correlation between branches extensively.

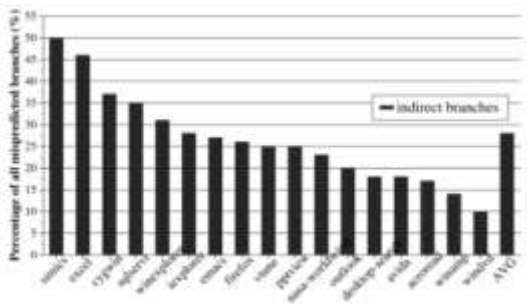

Figure 1. Percentage of indirect branches in programs

The conditional branch predictor used for the project here is a bi-mode branch predictor. The bi-mode is one of the best purely dynamic global predictors [2]. This prediction technique relies on the fact that most branches usually have a biased behavior such that they are usually taken or not taken. The global history scheme in conditional branch predictors is sometimes limited by destructive aliasing which occurs when two branches have the same global history pattern, but opposite biases. The bi-mode predictor overcomes this problem to a certain degree and hence was the choice for the conditional branch predictor. This project aimed to deliver less than 1 misprediction per 1000 instructions (MPKI). The main objective of understanding branch prediction techniques and realizing an efficient indirect branch predictor.

In the following sections, (II) contains information regarding some of the other conditional branch predictors and referred to and pertinent to the execution of this project. (III) Gives information regarding the algorithms used in the project such as the pseudo code and hardware diagrams for the same. (IV) is about the infrastructure on which the predictor was implemented. (V) talks about the results of this project. and the key takeaways from working on this project.

II. RELATED WORK

A. Evolution of bias predictors

Aliasing occurs when different branches try to access the same predictor leading to interference with current branch prediction by other branches. Aliasing can lead to either of the three, constructive, destructive or neutral interference. Constructive or neutral interferences do not negatively affect the performance of the predictor. Whereas negative interferences have adverse effects on the prediction accuracy. To mitigate the negative effects of aliasing the following branch predictors were realized. Among the predictors mentioned below, bi-mode predictor was selected for the project because of its relatively lower hardware cost and effectiveness.

The Agree Predictor:

The Agree predictor converts negative interference into positive or neutral interference using a biasing bit. It is latched with each line in the BTB. Here, the direction predictor does not solely determine the outcome of the branch. The biasing bit either agrees or disagrees with the

predicted direction and the outcome is predicted as taken when the bias agrees and vice versa.

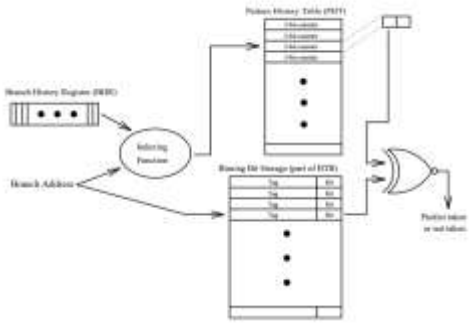

Figure 2. Agree Predictor

Although this technique has a low hardware overhead, its drawback is that the biasing cannot be updated, it is fixed which does not bode well for dynamic prediction of large programs.

The Skewed Branch Predictor:

The skewed branch predictor is based on the observation that most aliasing occurs not because the size of the pattern history table (PHT) is too small, but because of a lack of associativity with the PHT. The skewed branch predictor splits the PHT into three even banks and hashes each index to a 2-bit saturating counter in each bank using a unique hashing function per bank. The prediction is made according to a majority vote among the three banks.

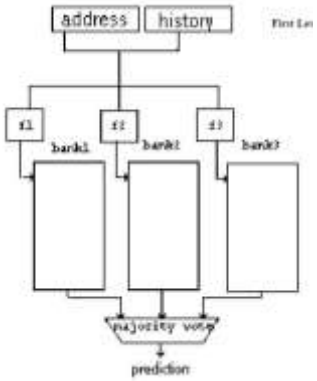

Figure 3.: Skewed Branch Predictor

The disadvantages associated with this technique are that using 1/3 - 2/3 of PHT size creates capacity aliasing, also this predictor is slow to warm up on a context switch.

III. ALGORITHMS USED IN THE PROJECT

The Bi-Mode predictor:

We have briefly discussed aliasing, destructive interferences and how some of the predictors overcome them. The bi-mode predictor belongs to that class of predictors which try to reduce aliasing. The predictor has two direction predictors which are half the size of the original table. There exists another patter history table called the choice predictor. The choice predictor is accessed using just the branch addresses whereas the direction predictors are accessed similar to gshare predictor, by XORing the branch address and the history register. The access to the direction predictors could also be done using a good hashing which reduces aliasing further.

The prediction works as follows:

- The choice predictor tells us which direction predictor to use.
- The chosen direction predictor tells us whether the branch is taken or not taken. (The direction predictor here is saturating 2-bit counters)

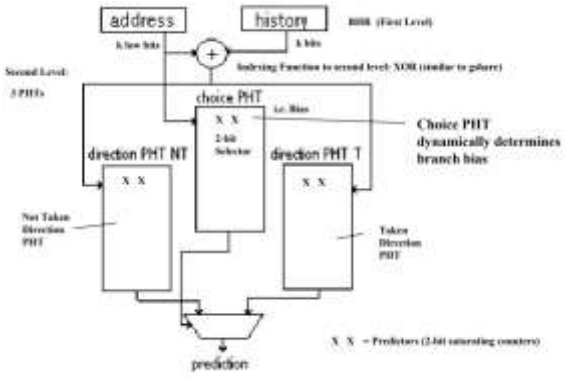

Figure 4 : Bi-mode predictor

My implementation:

The input to the prediction function is the index to choice predictor (cpi) and index to direction predictors (dpi). The output is the prediction of the chosen direction predictor.[3]

```
access_bp( cpi,  dpi)
{
Bias = CP [cpi]                                    //bias value
if(bias/2)              //true or false since it is a 2-bit predictor
prediction = DP1 [dpi]              //choose dp1 if bias is taken
else
prediction=DP2[dpi];          //choose dp2 if bias is not taken
return prediction/2;                               //make prediction
}
```

The updating algorithm works as follows:

- Only the direction predictor chosen is updated with the value of the outcome for every branch.
- The Choice predictor is updated for all branches with the value of the outcome, except in the case it gives a prediction opposite to the branch outcome but the right prediction was made by the direction predictor chosen.[3]

My implementation:

The input to the prediction function is the index to choice predictor (cpi), index to direction predictors (dpi), the prediction for that branch and outcome for that branch.

```
update_bp( cpi,  dpi, prediction , outcome)
{
  if(CP[cpi]/2)//update dp1 if that was the choice
  {
     //2-bit predictor updation
     if(outcome)
        if(DP1[dpi]<3) DP1[dpi]++;
     else
        if(DP1[dpi]>0) DP1[dpi]--;
  }
  else//update dp2 if that was the choice
  {
     //2-bit predictor updation
    if(outcome)
      if(DP2[dpi]<3) DP2[dpi]++;
    else
      if(DP2[dpi]>0) DP2[dpi]--;
  }
   //update CP always with outcome except when CP
   prediction is opposite of outcome but overall
   prediction was correct

  if(!(((CP[cpi]/2)!=outcome)&&(pred==outcome)))
  {
    //2-bit predictor updation
    if(outcome)
       if(CP[cpi]<3) CP[cpi]++;
   else
      if(CP[cpi]>0) CP[cpi]--;
   }
}
```

This updating scheme ensures that branches which are biased to be taken will have their predictions in the "taken" direction predictor table, and branches which are biased not to be taken will have their predictions in the "not taken" direction predictor table. Therefore, most of the information stored in the "taken" direction predictor table will be "taken" and any aliasing is more likely not to be destructive.[3]

Virtual Program Counter for indirect branch prediction:

The VPC implementation follows the high-level conceptual diagram shown in Figure 5. Here the global history register (VGHR) which is gives context information associated with each virtual branch. The program counter (PC) is virtualized by hashing it for each iteration and storing it in VPCA. The VPCA for each virtual branch is unique when PC is XORed with a random value selected from a table of constants using the iteration value. [1]

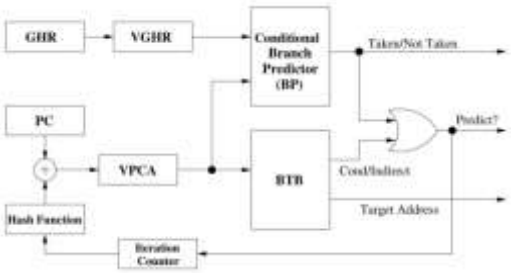

Figure 5: High-level conceptual diagram of VPC

The rest of the hardware is similar to that of a conditional branch predictor with BTB for target predictions Both conditional and indirect branches are fed to the bi-mode predictor. The bi-mode predictor is not aware that it is predicting an indirect branch. An iterative scheme that virtualizes the indirect branch is applied and that allows the conditional predictor to treat the indirect branch as multiple conditional branches. When a virtual branch is taken a target prediction is made by accesssing the Branch Target Buffer(BTB).

If the BTB access is a hit, the BTB entry provides the branch. VPC prediction algorithm continues iterating only if all of the following three conditions are satisfied:

- The first iteration hits in the BTB.
- The branch type indicated by the BTB entry is an indirect branch, and
- The prediction outcome of the first iteration is not-taken.

Therefore Prediction Algorithm of VPC is as follows:

```
iterations=1;                                   //iteration of VPC
VPCA=branch.address;                            //VPC=PC
VGHR=ghr;                                       //VGHR=GHR
done=false;
while(!done)  do
//Iterates till predicted or max iteration value reached
index_DP =  (VGHR << (forindex))
                ^ (VPCA & (dpsize-1))       //dir pred index
index_CP= (VPCA & (cpsize-1))          //save choice pred index
index = VPCA % (btbsize-1);                     //save btb index
pred_tgt= btb[index]                            //predicted target
pred_dir= access_bp(index_CP,index_DP) //predict
                                                direction
if(pred_tgt AND pred_dir)     //if target exists in btb and is
                                                predicted taken
{
      target_prediction(pred_tgt);              //target  prediction
      pred_iter=iter;      //iteration value at which prediction
                           occurs
      done=true;         //prediction for the indirect branch
                                                is done
}
else if((predicted_target is 0)||( iterations >=MAX_ITER))
{
      direction_prediction(false);//if not found set
      done=true;
}
VPCA = (VPCA)^Hash[iter-1];                     //VPCA hash
VGHR = VGHR<<1;                                 //VGHR Left shift
VGHR &= (1<<history_length)-1;                  //mask overflow
iterations ++;                                  //iterate
}
```

The training algorithm for VPC works as follows:
The VPC predictor trains both the BTB and the bi-mode predictor for each predicted virtual branch.[1]

- If the predicted target for an indirect branch was correct then the direction predictor for the virtual branch with the correct target is updated as "taken"

whereas the direction predictors for virtual branches before the right one are updated as "not-taken".

- In case the target exists in the BTB but wasn't predicted then the direction predictor is trained as "taken" for the virtual branch that points to the target. The other virtual branches are updated as not taken in the predictor.
- In case the target does not exist in the BTB, the target is inserted in the BTB for the virtual branch and VGHR value that missed.

```
if(target_prediction()==correct_target)//when correct
                                          target predicted
{
  iter = 1; //initalize iter VPCA VGHR
  VPCA=bi.address;
  VGHR=ghr;
  while (iter < pred_iter)                // till predicted
                                          value reached
  {
    dpi =  (VGHR << (forindex))
          ^ (VPCA & (dpsize-1));
    cpi= (VPCA % (cpsize-1));
    if (pred_iter == iter)          //when predicted iteration
                                      reached
    {
      p=access_bp(cpi,dpi);
      update_bp(cpi,dpi,p,true);
       btb[VPCA % (btbsize-1)]=correct_target;
    }

    else
    {
      p=access_bp(cpi,dpi);
      update_bp(cpi,dpi,p,false);//update bp with false
                                  when till pred_iter reached
    }
    VPCA = (VPCA)^Hash[iter-1];
    VGHR = VGHR<<1;
    VGHR &= (1<<history_length)-1;
    iter++;
  }
}
else
{
  iter = 1;                       //initalize iter VPCA VGHR
  VPCA=bi.address;
  VGHR=ghr;
  istgt_correct= false;
  while((iter<=MAX_ITER)&&(istgt_correct==false))
  //When target exists in btb but not predicted
  {
  dpi =  (VGHR << (forindex))
          ^ (VPCA & (dpsize-1));
  cpi= (VPCA & (cpsize-1));
  pred_target =  btb[VPCA % (btbsize-1)];
  if(pred_target==target)//if target found in btb
  {
    p=access_bp(cpi,dpi);
    update_bp(cpi,dpi,p,true);     //update bp to taken and
                                     btb at VPCA to tgt
    btb[VPCA % (btbsize-1)]=target;
    istgt_correct= true;
   }
  else if (pred_target)
  {
    p=access_bp(cpi,dpi);
    update_bp(cpi,dpi,p,false);     //if tgt not found in btb
   }
  VPCA = (VPCA)^Hash[iter-1];//iteration scheme of
                                                    VPC
  VGHR = VGHR<<1;
  VGHR &= (1<<hlen)-1;
  iter++;
  }
  if(istgt_correct==false)              //if target not in btb
  {
    dpi=index_DP;
    cpi=index_CP;
    update_bp(cpi,dpi,direction_prediction(),true);
    //update bp and inserting tgt in btb
    btb[index] = target;
  }
 }
}
```

Training is also an iterative process. Training is crucial for the working of the project. When working on the training algorithm a lot of issues were faced. Any slight misunderstanding in the training algorithm, whether it was conditional or indirect, would cause huge repercussions on the MPKI. The mistake I constantly made was not saving the context of the virtual branch during prediction which led to the wrong replacement being made in part 3 of the training algorithm[1]

IV. Insrastructure For Predictor Implementation

The predictor was developed on the infrastructure provided by Prof. Daniel Jiménez. The src folder contained the programs required to work on the predictor. "my_predictor.h" was the file in which the indirect predictor was implemented. There were two classes defined in the file, my_update and my_predictor which were subclassed from branch_update and branch_predictor defined in predictor.h. The my_update class performs the action of delivering direction and target predictions to the main program in predict.cc. "predict.cc" reads traces and simulates the predictor written in my_predictor.h on each trace. On running shell script "run" in the infrastructure the predictor is simulated and the MPKI values for all trace files, the average MPKI values for direct and indirect branches of all trace files are output on the terminal.

The infrastructure provide a level of abstraction to implement the predictor and verify the results easily.

V. Results and key takeaways

The VPC implementation drastically reduced the average MPKI of all the traces from 3.21 to 0.717. The graph below compares the BTB and VPC implementations:

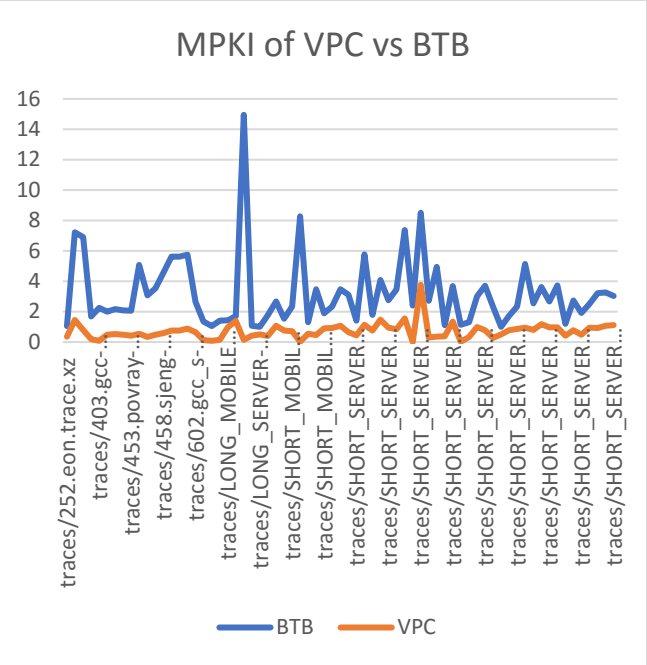

Figure 6: MPKI comparision

Key Takeaways from the project:

- The training of the predictor is crucial to achieve lower mispredictions.
- Indirect Branch Prediction algorithm is complex in comparison to conditional predictors.
- Lowering the Mispredictions is proportional to increasing the sizes of BTB and PHT.
- By virtualizing the indirect branch to multiple conditional branches. We were able to exploit correlation between branches using the history register.
- The hash function in the VPC enabled us to reduce interference in access to BTB during indirect branch prediction.

ACKNOWLEDGMENT

I would like to thank Prof. Daniel Jiménez for providing a good opportunity learn branch prediction, and also for providing the support to help realize the project. I would also like to thank Sangam Jindal who helped resolve any issues faced.

REFERENCES

[1] Hyesoon Kim, Jose A. Joao, Onur Mutlu, Chang Joo Lee ,Yale N. Patt , Robert Cohn, “Virtual Program Counter (VPC) Prediction: Very Low Cost Indirect Branch Prediction Using Conditional Branch Prediction Hardware,” IEEE Transactions on Computers; Volume: 58 Issue: 9

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

[3] http://meseec.ce.rit.edu/eecc722-fall2012/722-9-24-2012.pdf.