Michgan Tech

Branch Target Buffer Simulation

Project II

Writen by: liang YAN

Computer Science

Last modified: December 2013

Project Requirement

1. Develop a new class which implements a branch target buffer (BTB). The class should accept the total size of the buffer in bytes as a parameter. The BTB is constructed by valid bit, type bits, target address bits and tag bits;

2. The program should process the entire trace, going through the steps of : (1) Reading an input record; (2) Performing a prediction using the address if branch; (3) Comparing the prediction with the actual values found on the record. (4) Updating the direction predictor and the BTB with the actual values. Note that every branch and jump should be entered to the BTB.

3. Your combined predictor should operate as follows: (a) Check the BTB for a hit. A BTB hit means the tag portion of the address matches the stored TAG. (b) A BTB miss predicts not taken (i.e., next PC is PC+8) irrespective of the direction predictor outcome. (c) A BTB hit for a jump predicts taken and the target address is provided by the BTB. The next PC is the target address. 1(d) A BTB hit for a branch predicts taken if the direction predictor predicts taken. It predicts not taken if the direction predictor predicts not taken. The next PC is calculated accordingly.

4. Make sure that you correctly account for each misspredict ion by classifying them, (a) BTB miss : This is a branch or jump which would be taken. Due to BTB miss, next PC was predicted to be the fall-through address. (b) Wrong target : This is a jump, or a correctly direction predicted branch. BTB hit, but the stored address in the BTB was wrong. (c) Direction : The BTB hit and provided correct target address but the direction prediction was wrong.

5. Test your setup with 8KB of tournament predictor and 2, 4, and 8 KB of BTB space. Plot the misprediction rates as a stacked bar graph, using the classification above.

6. Write a report about your findings. In your report, it will be helpful to cover: The implementation process, the difficulties you have encountered,Your fully commented C++ program and class descriptions ,Branch prediction measurements itemized above, A discussion of the results you have obtained.

0.1 Introduction

According to the requirement, we have to design a branch target address which has three different spaces(2Kbytes,4Kbytes,8Kbytes), meanwhile it combines with a tournaPredictor which has a 8Kbytes size. At last, We need to measure the BTB by countering its hit miss, wrong target address and wrong direction based on a trace file which is composed by instructions.

0.2 Implementation

I choose C++ as the develop language, main.cpp contains the main exection, class btBuffer Implements the BTB simaulation, I took the class tournaPredictor from Project I,only add a new member variable currentPrediction which is used by the btBuffer.

0.2.1 main execute function

Reading from the trace file, get the exact value line by line. The instruction address, Introduction type, the taken or not taken state, and the target address. First I verify the value of instruction type, only processed when the type of instruction is 1 or 2. Loop execute the btb simaulation with different buffer size.

0.2.2 Branch target buffer simaulation

There are three key parts in this section. First, we need to find out the entry numbers of btb According to its size. Secondly, we could get the entry value and could update it. Third, we need to count the measurement variables correctly.

0.2.2.1 How BTB works

Before everything, we should figure out when we could use the targetAddress in a BTB Entry and when we should update the BTB Entry. In real CPU, BTB works together with a BPB(tournaPredictor here), the BTB only works when the branch is taken, its prediction is taken and two tags are same, (we could take a jump instruction as a branch instruction which is always taken and prediction is taken.) All the conditions must be achived. BTB update is much easier to understand, we only modify it when get a taken branch instruction.

0.2.2.2 Entry number computation

According to the basic idea of BTB, we need to seperate the pc address to index and tag, and the power of index could be the entry numbers of BTB , also , in the BTB, which is composed by valid,type,targetAddress and tag, the tag should equal to the tag in PC, so we could get a equation like this (For 2Kbytes):
$2K \geq EntryNumber \times OneEntrySize$
$2^{14} \geq 2^{n_{index}} \times (1 + 3 + 32 + 32 - 2 - n_{index})$
we could get $n_{index} = 8$,then the size of BTB is: $2^8 \times (1+3+32+32-2-8) = 1.8K \leq 2K$, accordingly we could get $n_{index} = 9, 10$ for 4K and 8K. I use entryNum = byteSize gg 3; for example: byteSize = $2048 = 2^{11}$ then EntryNum = $2^8 = 256$.

0.2.2.3 Operating BTB Entry

```
entry line structure:
0000 0000 0000 0000 0000 0000 0000 0000
   tag (22,21,20 bits)             |->

0000 0000 0000 0000 0000 0000 0000     000          0
32bit for target address       ->|     type    valid
```

I design the BTB Entry structure like above, first 1 bit for valid, the next 3 bits for type, and then next 32 bits for targetAddress, last part is for tag. I also designed two functions to operate BTB Entry as a whole part. The functions are mainly used shit operation and operation or operation like below:

```
unsigned long long  btBuffer::SetBTBEntry(btbData btbEntry)
{
  unsigned long long btbLine = 0;
  // add pc tag to btb entry line
  btbLine += btbEntry.pcTag;
  btbLine = btbLine << 36;

  // add target address to btb entry line
  unsigned long long temp = 0;
  temp += btbEntry.targetAddress;
  temp = temp << 4;
  btbLine += temp;

  // add type
  temp=0;
  temp += btbEntry.type;
  temp = temp << 1;
  btbLine += temp;

  //add valid
  btbLine += btbEntry.valid;
```

```
  return btbLine;
}
btbData btBuffer::GetBTBEntry(unsigned long long btbLine)
{
  btbData btbEntry;
  unsigned long long temp = btbLine;

  //get the valid
  btbEntry.valid= temp & 0x1; // get the last 1 bit for

  //get the type
  temp=btbLine;
  temp = temp >> 1 ;
  btbEntry.type=temp & (0x7);

  //get the target address
  temp=btbLine;
  temp = temp >> 4 ;
  btbEntry.targetAddress=temp & (0xFFFFFFFF);

  // get the tag
  temp=btbLine;
  btbEntry.pcTag = temp >> 36;

  return btbEntry;
}
```

0.2.2.4 Count measurement variable

To measure the effective of the BTB, I count three variables, hit miss, wrong taret and wrong direction.

For hit miss: when it is a jump instruction, it is a miss if the tag of BTB is different from its real tag; when it is a branch, it is a miss if the the instruction is taken and prediction of tournaPredictor is taken but the two tags are not same. For wrong target: when it is a jump instruction, and two tags are same, but the targetAddress of BTB is different from the real one; when it is a branch instruction, it is taken and the prediction is taken too, but two target address is different, then it is a wrong target.

For direction, it only works for branch instruction, we get a hit in BTB which means two tags are same, and the branch is taken, however the predictor tells us it is not taken, we could not use the BTB targetAddress which we could use.

the whole code Implementation is below:

```
if(!currentBTBEntry.valid){
    btb[index] = SetBTBEntry(realEntry);
  }else{
    // miss hit in BTB
    if((type == 2 && currentBTBEntry.pcTag != realEntry.pcTag)||
       (type == 1 && branchTaken == 1 && currentPrediction ==1 &&
```

```
        currentBTBEntry.pcTag != realEntry.pcTag))
          missHitCount++;

    // wrong target address
    if((type == 2 && currentBTBEntry.pcTag == realEntry.pcTag)
       || (type == 1 && currentPrediction ==1 && branchTaken == 1 &&
           currentBTBEntry.pcTag == realEntry.pcTag) )
      if(currentBTBEntry.targetAddress != target)
        wrongTargetCount++;

    // wrong direction
    if(type == 1 && currentBTBEntry.targetAddress == target &&
       currentBTBEntry.pcTag == realEntry.pcTag)
      if(currentPrediction == 0 && branchTaken == 1)
        wrongDirectionCount++;

    // update the btb entry line
    if(type == 2 || (type == 1 && branchTaken == 1))
      btb[index] = SetBTBEntry(realEntry);
  }
```

0.2.3 Tournament predictor

The predictor works seperately with branch target buffer, it updates everytime when meeting a branch instruction. I basicly used the class in the Project I, but add a new member variable called currentPrediction, I will get the currentPrediction before updates the prediction table

0.2.3.1 code Implementation

```
int predictionFromSa;
   if(saPredictor.predictionTable[index]%4 <2)
    predictionFromSa = 0;
  else
    predictionFromSa = 1;

  int predictionFromGl;
  if(glPredictor.predictionTable[glPredictor.globalHTindex]%4 < 2)
    predictionFromGl = 0;
  else
    predictionFromGl = 1;

  // get the value from the tourament predictor
  int selector = predictionTable[index]%4;
  if (selector < 2) // weak select
    currentPrediction = predictionFromGl;
  else // strong select
    currentPrediction = predictionFromSa;
```

First, we get the prediction from the saturaPredictor and glPredictor, then ,we get the prediction from the selector predictor, and get the prediction of tournaPredictor finally. After get the currentPrediction, we updates all the predictors.

0.2.4 Measurement

I count all the branch and jump instructions once they go into the stepSimulation of btb, I also count the misprediction numbers when it met the requirement. then get the Misprediction rate through the class member function. At last put all key data in a result.txt file.

0.3 Result anlysis

The result look like below:

Type	Table size	total execute instructions	Misprediction count	Misprediction Rate
hit miss	2048	78515912	3476399	0.044276
wrong target	2048	78515912	1041264	0.013262
wrong direction	2048	78515912	3584392	0.045652
hit miss	4096	78515912	1840090	0.023436
wrong target	4096	78515912	1164418	0.014830
wrong direction	4096	78515912	3905548	0.049742
hit miss	8192	78515912	867150	0.011044
wrong target	8192	78515912	1181053	0.015042
wrong direction	8192	78515912	4116827	0.052433

From the table and figure, we could find that according to the increase of BTB size, the hit miss is decreasing, but wrong target and the wrong direction works on the opposite way. It makes sense, we have more Entry, it would hit more, but the problem is more at the same time, so we should notice the balance of the BTB size.

0.4 The difficulties

1. To understand how a BTB work, this is also the purpose of this project. The TA gave me a lot of help for this problem, also I reviewed the class record, finally I figured it out, it felt good.

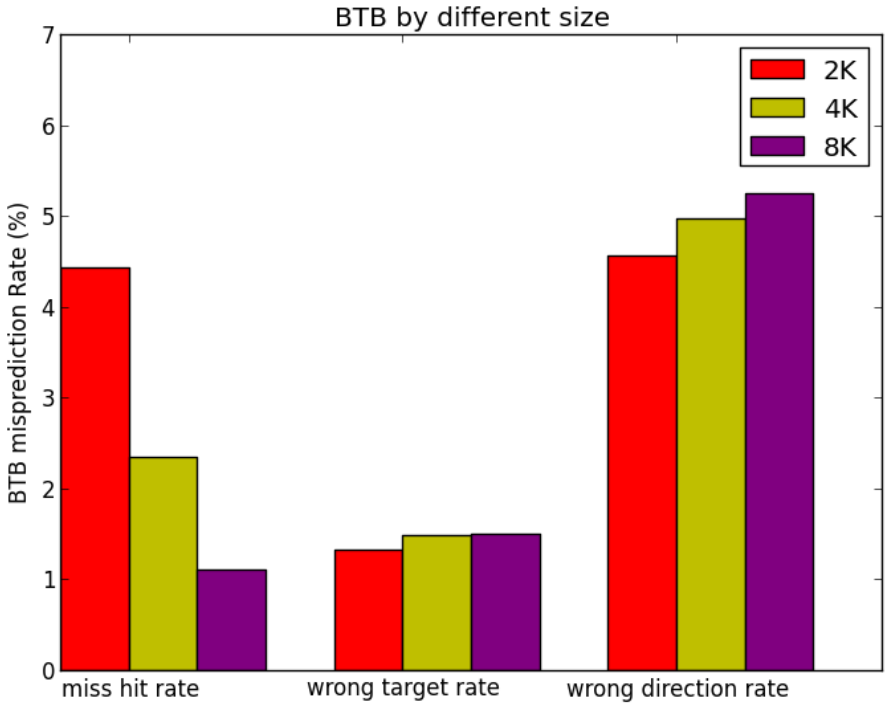

FIGURE 1

2. segfault error. It's not like the same error in Project I,as I used a loop for differnt BTB size, I could only run through the first iteration, I usef gdb and valgrind to fine the error position, it is caused by the test file, so I moved the fopen file out of loop , and use rewind funtion to let the stream pointer back to the head of the test file, problem is resolved.

3. BTB Entry bit operation. This part let me clealy know about the bit operation in C++.

0.5 Conclution

The effect of BTB is better than what I thought before. So many instruction could be repeated frequently, so it is necessary for modern cpu having the BTB and BPB. Also, I noticed most hit miss comes from jump instruction, so some addition structure for jump is helpful such as RAS. Finally, increase the size of BTB, could decrease the hit miss , but it would increase the wrong target and the wrong direction, so we should notice the balance.