CA Assignment Q2

Chinmay Sultania-IMT2021540

Anushka Singh -IMT2021531

Q2a.

We are going to look at the simple static branch prediction policies of "always predict taken" and "always predict not taken". Write a program to read in the trace and calculate the mis-prediction rate (that is, the percentage of conditional branches that were mis-predicted) for these two simple schemes.

The following are needed for the report:

• Results and output screenshots

• Which of these two policies is more accurate?

ANS:

Explanation of code:

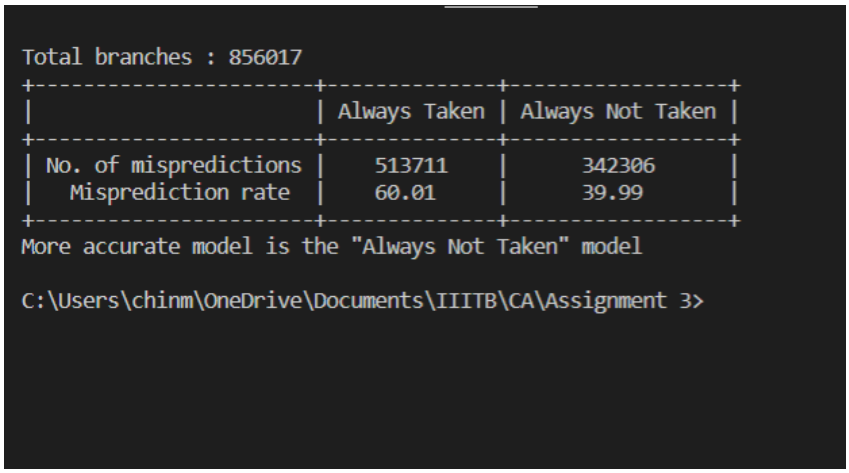
- Open trace file.
- Initiate two counters, misprediction for Always Taken and Not Taken.
- Then we find the percentage of both and return as output in tabular form using the prettytable module.

```
from prettytable import PrettyTable
myTable=PrettyTable([' ','Always Taken','Always Not Taken'])
count_t=0
count_n=0
f=open('file.txt','r')
for i in f:
    s=i.split()
    if(s[1]=='T'):
        count_t+=1
    else:
        count_n+=1
ans1=round(count_t/(count_t+count_n)*100,2)
ans2=round(count_n/(count_t+count_n)*100,2)
myTable.add_row(['No. of mispredictions',count_n,count_t])
myTable.add_row(['Misprediction rate',ans2,ans1])
print("Total branches : {}".format(count_t+count_n))
print(myTable)
print("More accurate model is the \"Always Not Taken\" model")
```

Calculations used:

- We have the no. of mispredictions for the “Always Taken” and the “Always Not Taken” model stored in variables count_n and count_t respectively.
- This is the count of ‘T’ in the actual branch while the predictor was “Always Not Taken” and vice-versa.
- Now to find the percentage of mispredictions we do the following calculations : (no.of mispredictions/total branches)*100.
- So the misprediction percentage for “Always Not Taken” is stored in ans1 and calculated using : (count_t/(count_t + count_n))*100.
- The misprediction percentage for “Always Taken” is stored in ans2 and calculated using : (count_n/(count_t + count_n))*100.

Output/Result Screenshot:



Accuracy of Policies:

The results obtained are:

- Misprediction rate when “Always Taken”: approx. 60%
- Misprediction rate when “Always Not Taken”: approx. 40%

Hence, by this we can say that misprediction is higher when we predict “Always Taken”. Therefore, always predicting “Not Taken” is more accurate.

Q2b.

The simplest dynamic branch direction predictor is an array of 2 n two-bit counters. It is advised to follow the notation discussed in class: strongly taken (00), weakly taken (01), weakly not taken (10), and strongly not taken (11). Prediction: To make a prediction, the predictor selects a counter from the table using the lower order n bits of the instruction's address (its program counter value). The direction prediction is made based on the value of the counter. Training: After each branch (correctly predicted or not), the hardware increments or decrements the corresponding counter to bias the counter toward the actual branch outcome (the outcome given in the trace file). Initialization: Although initialization doesn't affect the results in any significant way, your code should initialize the predictor to "strongly taken". Your task is to analyze the impact of predictor size on prediction accuracy. Write a program to simulate the two-bit predictor. Use your program to simulate varying sizes of the predictor. Generate data for predictors with 2 2 , 2 3 , 2 4 , 2 5 ... 2 20 counters (the address is a 32-bit binary number). These sizes correspond to predictor index sizes of 2 bits, 3 bits, 4 bits, 5 bits, ... 20 bits. Generate a line plot of the data using MS Excel or some other graphing program. On the y axis, plot "percentage of branches mis-predicted" (a metric in which smaller is better). On the xaxis plot the log of the predictor size (basically, the number of index bits).

The following are needed for the report:

• Results and output screenshots

• The line plot described above

• What is the best mis-prediction rate obtained in the analysis carried out?

• How large must the predictor be to reduce the number of mis-predictions by approximately half as compared to the better of "always taken" and "always not taken"? Give the predictor size both in terms of number of counters as well as bits.

• At what point does the performance of the predictor pretty much max out? That is, how large does the predictor need to be before it basically captures almost all of the benefits of a much larger predictor.

ANS:

Code Explanation:

- First we initialize a counter n to iterate in the range of 2 to 20 that is 19 times.
- We initialize the counter corr(*correct*) for counting the number of correct predictions.
- We then initialize an array with 00 entries 2^n times.
- Then for each time n changes we open trace file and iterate over it.
- We then split each line into its corresponding address(*fadd)* and actual prediction value*(pred).*
- We map the last n bits of the binary address to its corresponding index in the counter table.
- We now check the actual prediction value with the state of the FSM presently.
- According to that we update the state to its next corresponding state.
- We increment the value of corr if prediction is correct.
- Then using the pretty table module and calculating misprediction rates we obtain the outputs required in a tabular form.
- These values when input in excel generate the graph attached in the results section.

Calculations used:

- We assign a variable corr to store the number of times the prediction was correct.
- We have another variable total which has the total number of the branches in the trace.
- So to find the misprediction percentage we do ((total-corr)/total)*100.

```python
from prettytable import PrettyTable
MyTable=PrettyTable(['value of n','predictor size','misprediction rate'])
for n in range (2,21):
    corr=0
    tot=856017
    a=0
    arr=[]
    while a<(2**n):
        arr.append(0b00)
        a+=1
    f=open("trace.txt","r")
    for i in f:
        add=i.split()
        fadd=int(add[0])
        pred=add[1]
        rem=fadd%(2**n)
        k=arr[rem]
        if (k==0b00 or k==0b01) and pred=='T':
            corr+=1
        elif (k==0b10 or k==0b11) and pred=='N':
            corr+=1
        if (k!=0b00) and pred=='T':
            arr[rem]-=1
        if (k!=0b11) and pred=='N':
            arr[rem]+=1
    misp=round((tot-corr)*100/tot,2)
    MyTable.add_row([n,2**n,misp])
print(MyTable)
```

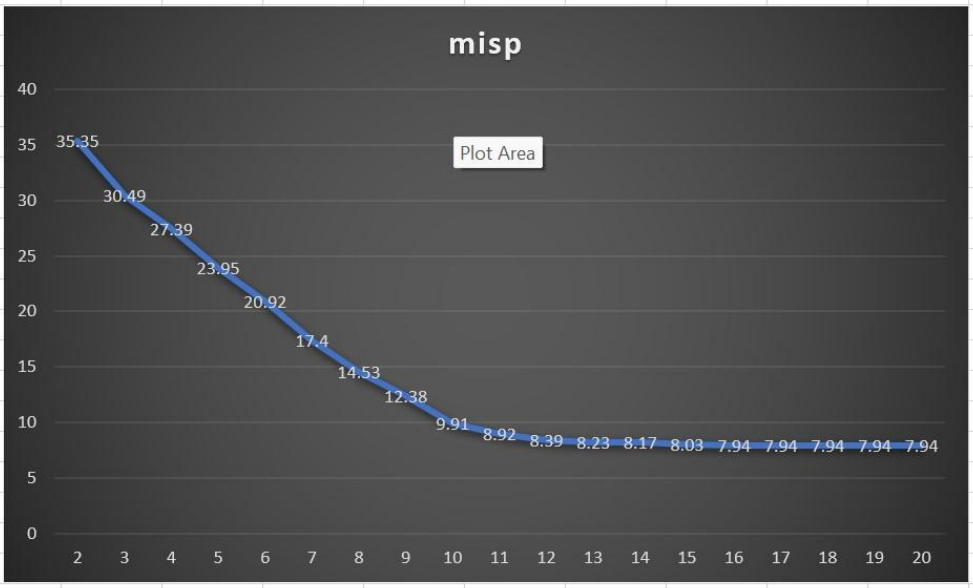
Results and output screenshots:

Output in terminal:

```
+------------+----------------+--------------------+
| value of n | predictor size | misprediction rate |
+------------+----------------+--------------------+
|     2      |       4        |       35.35        |
|     3      |       8        |       30.49        |
|     4      |       16       |       27.39        |
|     5      |       32       |       23.95        |
|     6      |       64       |       20.92        |
|     7      |      128       |        17.4        |
|     8      |      256       |       14.53        |
|     9      |      512       |       12.38        |
|     10     |      1024      |        9.91        |
|     11     |      2048      |        8.92        |
|     12     |      4096      |        8.39        |
|     13     |      8192      |        8.23        |
|     14     |     16384      |        8.17        |
|     15     |     32768      |        8.03        |
|     16     |     65536      |        7.94        |
|     17     |     131072     |        7.94        |
|     18     |     262144     |        7.94        |
|     19     |     524288     |        7.94        |
|     20     |    1048576     |        7.94        |
+------------+----------------+--------------------+
```

Graph obtained in excel:

n	misp
2	35.35
3	30.49
4	27.39
5	23.95
6	20.92
7	17.4
8	14.53
9	12.38
10	9.91
11	8.92
12	8.39
13	8.23
14	8.17
15	8.03
16	7.94
17	7.94
18	7.94
19	7.94
20	7.94



The best mis-prediction rate obtained in the analysis carried out:

We can clearly see from the graph as well as the table obtained that the approximate best misprediction rate we can achieve is 7.94%.

Size of predictor to reduce the number of mis-predictions by approximately half as compared to the better of "always taken" and "always not taken":

The better misprediction rate among the two is in “always not taken” and its value is approximately 40%. So according to the table the predictor size must be of n = 6, i.e 2^6 = 64 lines big to obtain a misprediction rate of close to half of 40% which is 20%.

Point at which the performance of the predictor pretty much max out:

According to the graph the predictor starts to flatline the misprediction rate at n = 12 and reaches the minimum approximate value of 7.94% for n = 16. Therefore at n = 16, i.e a size of 2^16 = 65,536 ,the predictor performance maxes out.