COMP25212 System Architecture

March 22, 2023

Q1.

5.2% of instructions accounted for 90% of the executed instructions. Using a simple python script I summed the number of times each instruction was executed then sorted by most executed and tallied number of executions until it exceeded 90% of the total executions. I then dived the number of instructions I had tallied and divided by total instructions.
answer:**5.2%**

Q2.

428065 branches where taken
answer:**428065**

Q3.

3423933 instructions where executed so on average 8.00 instructions are run before the pc is reloaded.
answer:**8.00**

Q4.

856130 (2 x 428065) instructions where wasted making the 20.0% of instructions wasted.
answer:**20.0%**

Q5.

answer:**89.3%**

Q6.

answer:**2.6%**
Of the branches taken only 45859 where not B branches so using the calculation Q4 we get a new percent of wasted time.

Q7.

Branches at the end of loops are imperfect because at the end of a loop the branch behave differently than previously expected and only does so once. Sometimes branches are effected by unpredictable external input and cant be predicted.

Q8.

answer:**428065:239802**

Q9.

answer:**272241:239802**

Q10.

answer:
forward: 119969:154253
backward: 146936:59812

Q11.

When taking a backwards branch predict it is taken as the majority of the time it is, and when forwards assume it isnt taken. This is more effective on backwards branches.

Q12.

A high degree of associativity is preferable as the cache is usualy very small and can still be fast and due to its size means direct mapped migh result in many collisions.

Q13.

1 word or 32 bits as that is the size of the address expected by the branch.

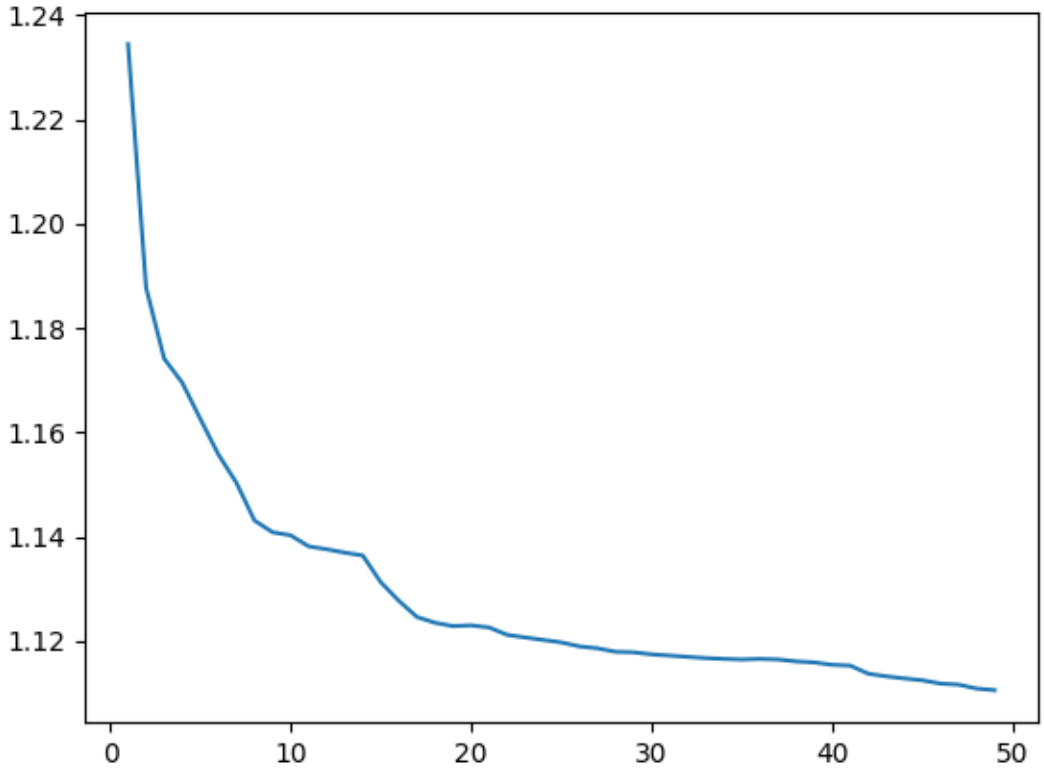
Q14.

I cache all invariant branches using a cyclic cache replacement policy. If cached then it is assumed that the branch is taken.

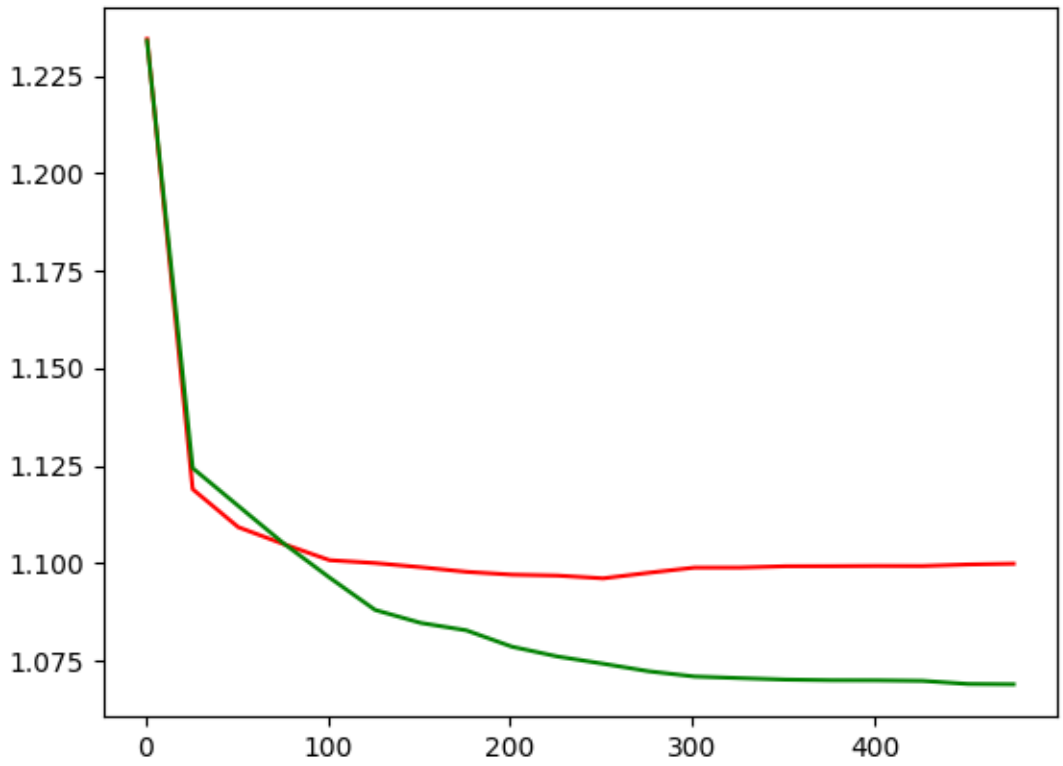
```
cycle = 0
cachesize = 15
cache = []
for i in range(cachesize):
    cache.append(None)

def searchCache(key):
    global cycle, cachesize, cache
    for entry in cache:
        if entry == key:
            return True
    return False

def addCache(key):  #cycle cache replacement
    global cycle, cachesize, cache
    cache[cycle] = key
    cycle += 1
    if cycle == cachesize:
        cycle = 0

f = open("block_profile", "r")
lines = []
noPenalty = 0
penalty = 0
for line in f:
    lines.append(list(filter(lambda a:a != " " and a != "", line.split(" "))
for line in lines:
    if line[0] == "B":
        hit = searchCache(line[2])
        if hit:
            if line.count("not") == 0:
                noPenalty += 1  #cache hit and taken
            else:
                penalty += 1    #cache hit but not taken
        else:
            if line.count("not") > 0:
                noPenalty += 1  #no cache hit not taken
            else:
                penalty += 1 #no cache hit is taken
                addCache(line[2]) #no hit and taken so store in cache
    else:
        if line.count("not") > 0:
            noPenalty += 1  # not invariant and not taken
        else:
            penalty += 1    #not invariant is taken
print(penalty)
```

```
print(noPenalty)
```

Q15.



Q16.



State aproach
Non state aproach

Q17.

answer:**0.59049**

Q18.

One one simple predictor is a loop predictor. Most loops will itterate x many times then not once and in the case of loops x is known and each loop can correctly be predicted including the last branch unlike if you used a simple dynamic branch predictor.

Q19.

Returns from functions use a return stack that predicts the function will return to the place it wascalled from. The return stack is a mirror of the call stack. This means the prediction can handle recursion as simply the same return will be pushed to the stack mutiple times. Another option is a local branch predictor that uses a history buffer for each branch and will predict based on the history. This improves over the state approach as you have access to more data and can study patern.

Q20.

A switch case statement or some if statements also qualify.