Computer Architecture Assignment 1

Mark Nemec - s1140740 | Tutor: Boris Grot

Note: I have been able to simulate all branch predictors completely.

The purpose of this report is to describe the internal structure of the branch predictor simulators built for this assignment and summarise the experiments performed on them. The language of my choice is Python. I implemented each predictor as a separate function in one file called `pred.py`. The source code for each predictor is presented and I will try to clarify how it works. Furthermore, I wrote a `main` function which just opens the supplied file and runs these functions on it and a helper generator function `parse` which loops over the lines of the file and yields the address and whether the branch was taken or not. It also resets the file pointer after it exits the file loop.

Description of the implementation of various predictors

Static Predictor - Always Taken & Always Not Taken

In this simulator the branch is always predicted as either taken or not taken. Following is the code used for this simulator:

```
1. def static_predictor(file, pred):
2.     total, correct = 0.0, 0
3.     for addr, taken in parse(file):
4.         total += 1
5.         correct += taken == pred
6.     return 1 - (correct / total)
```

Argument `file` is the trace file being used and argument `pred` (set to either 0 or 1) determines whether the branch is always predicted as taken or not taken. Variables `total` and `correct` are both initialised to zero and while `total` is incremented with each branch encountered, `correct` is incremented whenever the prediction is correct. The predictor returns the misprediction rate. Similar pattern is used in the following predictors as well.

Profile Guided Predictor

This predictor uses a profiling file `file` to determine addresses on which branches are taken more than 50% of the time and predicts those branches as always taken and the rest as not taken. Following is the code for this predictor:

```
 1. def guided_predictor(file):
 2.   total, correct = 0.0, 0
 3.    addr_taken = defaultdict(lambda: (0, 0))
 4.    for addr, taken in parse(file):
 5.        num_taken, num_found = addr_taken[addr]
 6.        addr_taken[addr] = num_taken + taken, num_found + 1
 7.
 8.    for addr, taken in parse(file):
 9.        total += 1
10.        took, addr_total = addr_taken[addr]
11.        should_take = 1 if took > 0.5 * addr_total else 0
12.        correct += taken == should_take
13.    return 1 - (correct / total)
```

This predictor has two stages:

1. Profiling stage: lines 3 - 6

In this stage the file is profiled. We create a dictionary `addr_taken` indexed by the jump addresses of the branch instructions and for each address we save the number of times it was taken `num_taken` and the number of occurences in the file `num_found`.

2. Prediction stage: lines 8 - 12

In this stage we go through the file again and for each branch instruction we look into the `addr_taken` dictionary and compute the percentage of times the particular branch was taken in the profiling stage. If it is more than 50% than we predict that branch as taken otherwise we predict it as not taken.

Two-level Adaptive Predictor

This predictor maintains a 2-bit finite state machine for each potential history of each branch address. Following is the code for this predictor:

```
 1. def history_predictor(file, hist_bits=2, state_bits=2, init_state=0):
 2.     total, correct = 0.0, 0
 3.     states = pow(2, state_bits) - 1
 4.     half_state = states / 2.0
 5.
 6.     preds = defaultdict(lambda: defaultdict(lambda: init_state))
 7.     hists = defaultdict(lambda: (0,) * hist_bits)
 8.
 9.     for addr, taken in parse(file):
10.         total += 1
11.         pred = preds[addr]
12.         hist = hists[addr]
13.         state = pred[hist]
14.         should_take = 1 if state > half_state else 0
15.         correct += should_take == taken
16.         pred[hist] = min(state + 1, states) if taken else max(state - 1, 0)
17.         hists[addr] = hist[1:] + (taken, )
18.
19.     return 1 - (correct / total)
```

Arguments `file`, `hist_bits`, `state_bits`, `init_state` represent the trace file being used, number of bits used for history, number of states for each finite state machine predictor and the initial state of those predictors, respectively. Dictionary `hists` is used for storing the history for each branch address and the dictionary `preds` is used for storing the predictors, initialised to `init_state`. Each branch instruction in the file is then looped over and its history `hist` is retrieved and used to retrieve the state `state` of the predictor associated with the history. If the state of the predictor is greater than half of the number of states then the branch is predicted as taken. Afterwards, the state of the predictor is incremented if the branch was taken otherwise it is decremented. History is also updated by adding the last 'taken' value to the list of previous values.

Experiments

I ran the experiments for both files `gcc_branch.out` and `mcf_branch.out` on each predictor. Below are the misprediction rates:

- gcc_branch.out
  - Always Taken Predictor - 23.6110%
  - Always Not Taken Predictor - 76.3890%
  - Profile Guided Predictor - 6.6285%
  - Two-level Adaptive Predictor
    - 1-bit History - 2.7412%
    - 2-bit History - 2.6243%
    - 3-bit History - 2.5546%
    - 4-bit History - 2.5365%
- mcf_branch.out
  - Always Taken Predictor - 32.2401%
  - Always Not Taken Predictor - 67.7599%
  - Profile Guided Predictor - 10.9440%
  - Two-level Adaptive Predictor
    - 1-bit History - 9.6773%
    - 2-bit History - 9.8847%
    - 3-bit History - 9.6106%
    - 4-bit History - 9.7785%

From the misprediction rate being higher for Always Not Taken predictor than Always Taken predictor for both of the files we can deduce that they both contain more branch instructions that were taken than not taken.

The Profile Guided Predictor performed better than the Static Predictors but fell short of beating the Two-level Adaptive Predictor even though it was just by small difference. This implies that many branches were taken a lot more than 50% or a lot less than 50% of the time.

Furthermore, we can confirm that using more bits for history does indeed improve the performance of the Two-level Adaptive Predictor but the difference is marginal and since the worst-case space complexity of the predictor, $O(n * 2^k * 2^l)$ where `n` is the number of unique addresses, `k` is the number of bits used for history, and `l` is the number of bits used for states, grows exponentially it has a very high memory overhead. Moreover, as we start increasing the number of bits used for memory the misprediction rate will go up. We can confirm this by running the program: `python 2.7 pred.py gcc_branch.out fsm --hist HIST` with increasing the argument HIST.

Additionally, I ran a small script that counts the number of unique branch addresses: `sed -E 's/B(.+) [01]/\1/' [filename] | sort | uniq | wc -l` exchanging filename with both gcc_branch.out and mcf_branch.out and found out that gcc_branch.out has 286 unique addresses whereas mcf_branch.out only has 33. This could indicate that mcf_branch.out is a trace file of a more loop-heavy program. This could explain why the adaptive predictor performed worse on mcf_branch.out.