Coursework 1

1.0 Simulator Structure

The simulator for branch prediction is written in Python and runs on Python 2.7. It consists of two files located in the top level directory. The first one - *static_predictor.py* - contains functions necessary for running the static predictors whereas the second file - *adaptive_predictor.py* - contains functions for running the 2 level adaptative predictor. Test files provided with this assignment are located in the package for convenience.

Both static and adaptive branch predictors are fully implemented.

1.1 Static Branch Predictors

To run a one of the static predictors - Always Taken, Always Not Taken and Guided Predictor - navigate inside the source package and execute the following command on a DICE machine:

- Always Taken Branch Predictor

  ```
  python2.7 static_predictor.py <file_name> alwaystaken
  ```

- Always Not Taken Branch Predictor

  ```
  python2.7 static_predictor.py <file_name> nevertaken
  ```

- Profile Guided Branch Predictor

  ```
  python2.7 static_predictor.py <file_name> profile
  ```

The miss rate will be printed in the terminal if a valid file is supplied.

1.1.1 Code structure

Each of the three predictors are created as a class with a method *analyze()*. Calling this method will run analysis on the file supplied. For each instruction in the text file, first it is split into tokens and wrapped in a dictionary. Secondly, a prediction is made and compared to the actual value given by the tokens and statistics counter is incremented accordingly.

Always Taken and Always Not Taken predictors return 1 and 0 respectively for their prediction - their also produce complementary results for the same files.

For the Guided Predictor, the sequence of steps above is preceded by creation of a profile and the profile is used in the prediction.

1.2 Adaptive Branch Predictor

To run the Adaptive Branch Predictor, execute the following command:

```
python2.7 adaptive_predictor.py <file_name> <history>
```

1.2.1 Code Structure

Adaptive Predictor requires more data stored in memory and more setup. Initially, an empty dictionary of branch addresses is created. As instructions are read from the file, new keys are added into the dictionary with instruction address as the key and a dictionary with states as the keys and counts (0-3) as values whenever a new state is encountered. Additionally, each address holds a key 'history' with the most recent history. Integer flags for states are being used with *0 - strongly not taken, 1 - weakly not taken, 2 - weakly taken, 3 - strongly taken*. History is kept as a list of integers and is being shifted to the right - most recent value added at the front and the oldest value pushed off from the end.

Whenever our prediction is made, the state of the current history is updated based on the outcome of the instruction. Additionally, the history is updated to reflect the most recent state of each address.

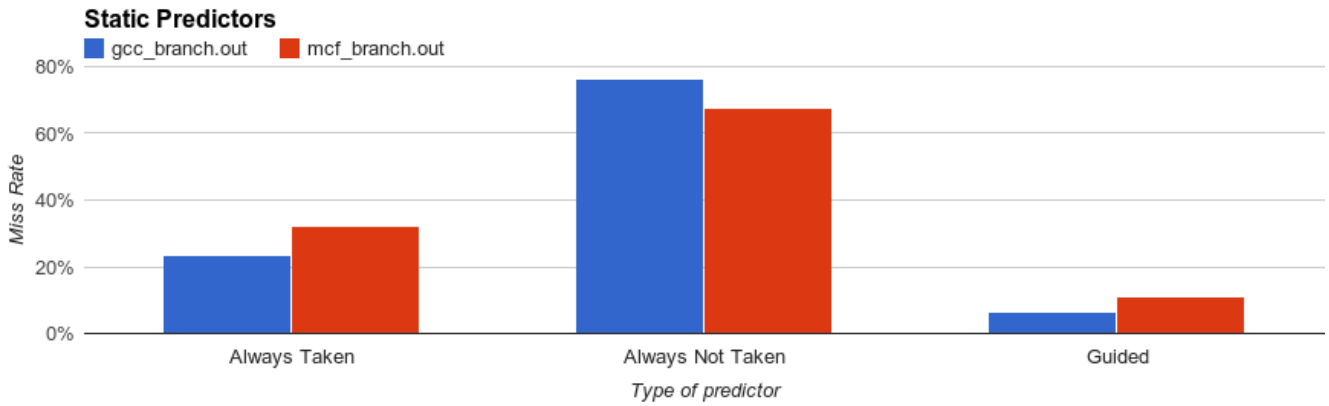
2.0 Experiments

The experiments in this section will be run on the provided files - *gcc_branch.out* and *mcf_branch.out*.

2.1 Static Predictors

The following table and graph displays the miss rate of each of the static predictors ran on the supplied files.

	gcc_branch.out	mcf_branch.out
Always Taken	23.611%	32.24%
Always Not Taken	76.389%	67.76%
Guided	6.628%	10.944%



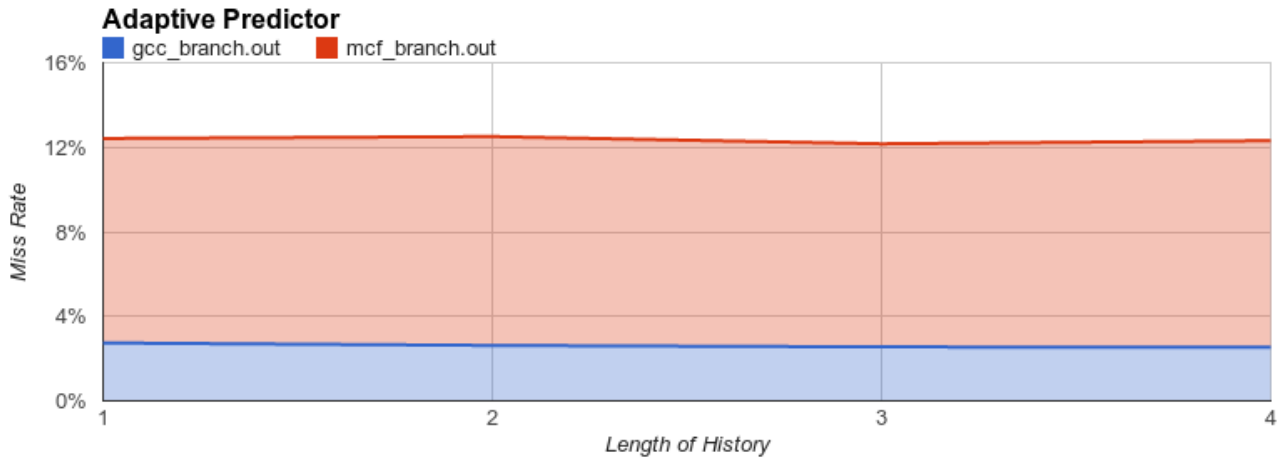
From the data, we can observe that the best performance is given by the guided predictor. This is due

to the profile being tailored for the specific dataset. On the other hand, Always Not Taken predictor performed poorly on both test files. Misprediction rate of around 70% is beyond the expected rate of picking whether to branch or not on random. Relatively common pattern of loops in programing hurts the performance of Always not Taken as there is a reasonable assumption the loop will be executed several times before exited.

2.2 Adaptive Predictor

Adaptive predictors perform better than static predictors due to their ability to learn patterns in branching and use those patter to make better predictions. On the other hand, they take some time to learn and might not perform so well for simple branches without any patterns on them. Additionally, they require histories and states to be stored for each branch which can be relatively expensive.
The table below presents the misprediction rate for the two test files:

Adaptive Predictor / Length of History	gcc_branch.out	mcf_branch.out
1	2.741185%	9.677334%
2	2.624345%	9.884718%
3	2.554613%	9.610559%
4	2.536546%	9.778526%



In the case of gcc_branch.out, adaptive predictor performs relatively well with minor variance among the results with history of length 4 performs the best. There is a decreasing trend apparent with longer histories resulting in lower misprediction rates. This, however, comes at a cost of data storage becomes exponentially more demanding as each additional bit of history doubles the number of tables required to keep to make predictions. The gcc test file has a relatively large number of distinct branches and a fairly large number of re-visits of branches which suggests the adaptive predictor can

learn the patterns.

In the case of mcf_branch.out, the longest history does not provide the best results. The best results for mcf_branch.out were obtained when history was of length 3. This would suggest that histories of longer length may be trying to find patterns in the branching that do not actual exist. The mcf test file also contains a fairly small number of distinct branches which would suggest that it may be loop/if intensive.

3.0 Conclusion

Based on the results obtained from the test files, adaptive prediction performs far better but at a higher cost of prediction. Adaptive prediction would work better on general purpose programs rather than embedded systems. For embedded systems, profiling the program may produce better results at lower storage cost.