Assignment 1: Understanding Branch Prediction

Computer Architecture

Due: Monday, February 20, 2017 at 4:00 PM

This assignment represents the first practical component of the Computer Architecture module. This practical contributes 12.5% of the overall mark for the module. It consists of a programming exercise culminating in a brief written report. Assessment of this assignment will be based on the correctness of the code and the clarity of the report as explained below. The practical is to be solved individually. Please bear in mind the School of Informatics guidelines on Academic Misconduct. You must submit your solutions before the due date shown above. Follow the instructions provided in Section 3 for submission details.

In this assignment, you are required to explore Branch Prediction techniques using the Intel Pin simulation tool. You are strongly advised to commence working on the simulator as soon as possible.

1 Overview

A branch predictor anticipates the outcome of a branch, before it is executed, to improve the instruction flow in the pipeline. In this assignment you will investigate the accuracy of various branch predictors. Towards this end, you have to implement three different branch predictors, as follows:

1. *Bimodal* Branch Predictor
2. *2-level* Global Branch Predictor
3. *Tournament* Branch Predictor that combines 1 and 2

1.1 PIN tool

Pin is a dynamic binary instrumentation engine that enables the creation of dynamic analysis tools (e.g. architectural simulators). Pin intercepts program execution and provides an API that allows you to execute high-level C++ code in response to runtime events like loads, stores, and branches. A Pin tool is a program which uses the API and specifies what has to happen in response. In this assignment, you are given an example Pin tool that intercepts branches and simulates an AlwaysTaken branch predictor. You will extend this tool to simulate three other branch predictors.

The directions below apply to DICE machines.

The Pin tool and the benchmarks are available in a directory that can be accessed through a DICE machine. Inside that directory there is a README text file that contains guidelines on how to run the simulator with the benchmarks.
To access the guidelines within the directory, type the following commands in the terminal:

1. `CAR_ASSGN1_PATH=/afs/inf.ed.ac.uk/group/teaching/car/assignment1`
2. `cd $CAR_ASSGN1_PATH`
3. `gedit README`

The directory that contains the skeleton source code for the branch predictor (`branch_predictor_example.cpp`) is further down that path, at:

`$CAR_ASSGN1_PATH/tools/pin-3.0-76991-gcc-linux/source/tools/BPExample/`

When the Pin tool runs `branch_predictor_example.cpp`, it requires a benchmark program (along with its arguments) to be given as the last command line arguments. The Pin tool runs the benchmark and while running it, passes all conditional branches to the simulated branch predictor. The given predictor inside `branch_predictor_example.cpp` is called AlwaysTakenBranchPredictor and predicts Taken for all of the branches.

Once Pin exits, it will generate a set of statistics for the simulated branch predictor in a file `BP_stats.out`.

The branch predictor simulator, in the file `branch_predictor_example.cpp`, allows for 3 different command line arguments:

1. The kind of predictor to be simulated, which include: Always Taken, Bimodal, 2-level, Tournament. The default predictor is Always Taken. To run the Always Taken branch predictor simulator, give this command line argument type: `-BP_type always_taken`
2. The number of entries in the PHT (pattern history table) of the Predictor. Default is 1024. To run the simulator with 1024 PHT entries, give this command line argument type: `-numBPEntries 1024`
3. The name of the output file to be generated,which has a default value of `BP_stats.out`. To generate an output file called MyOutput.out, give this command line argument type: `-o MyOutput.out`

In the path `$CAR_ASSGN1_PATH/benchmarks` there are 3 different benchmark programs (Gromacs, Gobmk, Sjeng) from the SPEC2000 benchmark suite. You must run your experiments with all 3 benchmarks, for the 3 requested predictors, with 3 different PHT sizes. In total, you will run 27 different experiments.

The guidelines inside the README text file contain more information about how exactly to run the branch predictor simulator and how to use the command line arguments.

1.2 Branch Predictor Parameters

The experiments must be run with the following three different sizes for the Pattern History Tables (PHTs): 128, 1024, and 4096 entries. Note that you must assess all three PHT sizes for each of the different branch predictors.

Each PHT entry is logically comprised of 2-bit saturating counters as explained in the lecture slides. The counters must be initialized to 0. Remember that you are writing a simulator in a high-level language; you are *not* designing actual circuits. As such, you may use any data types for the counters (and other program variables). Regardless of the data type you use, it must behave like a saturating 2-bit counter.

Bimodal Branch predictor

- The PHT is indexed with the least significant bits of the Program Counter of the branch instruction

2-level Global Branch Predictor

- The Global History Register is composed of as many bits as necessary to index the PHT. For example, if the PHT contains 1024 entries, then the GHR should be 10 bits long. These 10 bits contain the outcomes of the last 10 branches, with 0 denoting *not taken* and 1 denoting *taken*. GHR should be initialized to 0. The earlier comment about the data types in your simulator applies here as well.

Tournament Branch Predictor

- The Tournament predictor is composed of three different predictors: the Bimodal, 2-level Global predictor, and a *meta-predictor* that selects between the Bimodal or the 2-level predictor to provide the actual prediction of the branch outcome.
- The Experiment must be run with PHTs of 3 different sizes. In each experiment the PHTs of the Bimodal Predictor and the 2-level Predictor must all have the same size. I.e. when the meta-predictor PHT is 1024 entries then the 2-level Predictor PHT is also 1024 entries and the Bimodal Predictor PHT is also 1024 entries.
- The PHT of the meta-predictor is indexed with the Global History Register.
- In the meta-predictor, when the prediction bit (i.e. the MSB) of the saturating counter is 0, the Bimodal predictor is selected to supply the prediction. When the MSB is 1, the 2-level predictor is selected.
- The Tournament predictor will follow a total update policy. Both the Bimodal and the 2-level predictors will be trained after each branch is resolved. After a correct prediction the relevant meta-predictor entry must be strengthened. After a misprediction, the relevant meta-predictor entry must be weakened *only if the predictor that was not chosen was correct*. If both the Bimodal and the 2-level predictors were incorrect, there is no update to the meta-predictor.

  - To strengthen a saturating counter means to increment it, if its prediction bit is '1', and decrement it if the prediction bit is '0'.

- To weaken a saturating counter means to decrement it, if its prediction bit is '1', and increment it if the prediction bit is '0'.

- Since the underlying behavior of the Bimodal and 2-level predictors is unmodified in the Tournament scheme, you may choose to use your existing implementations of Bimodal and 2-level predictors to implement the Tournament predictor.

1.3 Simulator infrastructure

The three requested predictors must all be implemented inside the `branch_predictor_example.cpp` file. Inside the file, you must create a class with an appropriate name for each predictor. The file `branch_predictor_example.cpp` includes comments and guidelines on how you must create your classes and the appropriate member functions. As an example, an AlwaysTaken predictor is already implemented. You should start with the same code as the one for the AlwaysTaken predictor and modify it to implement new functionality for each specific predictor type.

For the purpose of this assignment you can ignore the Branch Target Buffer(BTB) (i.e. the implementation of a BTB is not needed). The simulated predictors must only generate a prediction without specifying the target address of the branch. Additionally the predictors do not need to identify whether an instruction is a conditional branch; they can assume that every instruction is a conditional branch, because the given code feeds only conditional branches into the prediction functions.

The contents of the file branch_predictor_example.cpp should be modified only as the above guidelines specify, and not in any other way.

1.4 Reference results

The folder BPexample contains a text file called results. That text file must be filled by the student with the misprediction rates for all combinations of benchmark, type of predictor and PHT size. The misprediction rate must be the same as the one visible in the output file of the branch_predictor_example.cpp (e.g. BP_stats.out). Report your results with 3 digits after the decimal point. Some reference results are listed below:

Gromacs Bimodal 128 : 0.338
Gobmk Twolevel 1024 : 0.257
Sjeng Tournament 4096 : 0.214

Note that all 27 entries of the text file must be completed. The results file must be submitted along with the source files.

2 Marking Scheme

Correctness (80 marks). Includes code functionality and quality for the following 3 predictors:

1. **Bimodal Branch Predictor (20 marks)**

2. **2-level Global Branch Predictor (30 marks)**

3. **Tournament Branch Predictor (30 marks)**

Report (20 marks). You should write a short report (max 2 pages) on how you have implemented each predictor and where the predictor code is (source file, line numbers). In the report you should answer the following questions

- Why the different predictors yield different (or similar) results
- Why and how, varying the size of the PHT impacts (or not) the misprediction rate
- Are the results what you expected and why (or why not)

3 Format of your submission

Your submission should clearly indicate (in both the report and the code) which branch prediction techniques you have simulated completely and which you have only partially completed. Please put comments in the code that you add to make it easy for the markers to understand. Remember that your simulator will be compiled/executed on a DICE machine, so you must ensure it works on DICE. Submit an archive of your Branch Predictors source code, along with the results text file, and a soft copy of your report as follows, using the DICE environment:

```
$ submit car 1 branchPredictors.tar.gz report.pdf
```

4 Similarity Checking and Academic Misconduct

You must submit your own work. Any code that is not written by you must be clearly identified and explained through comments at the top of your files. Failure to do so is plagiarism. Note that you are required to take reasonable measures to protect your assessed work from unauthorised access. Detailed guidelines on what constitutes plagiarism and other issues of academic misconduct can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

All submitted code is checked for similarity with other submissions using software tools such as MOSS. These tools have been very effective in the past at finding similarities and are not fooled by name changes and reordering of code blocks.

5 Reporting Problems

Send an email to Artemy.Margaritov@ed.ac.uk **and** Vasilis.Gavrielatos@ed.ac.uk if you have any issues regarding the assignment.