Understanding Branch Prediction

Computer Architecture and Design - Practical 3

Due: 21-11-2019 at 1600

This handout describes the third practical component of the Computer Architecture and Design module, which contributes 35% of the practical component of the module. It consists of a programming exercise culminating in a brief written report. Assessment of this practical 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 practical, 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 practical you will investigate the accuracy of various branch predictors. Towards this end, you have to implement three different branch predictors, as follows:

1. *Local* Branch Predictor
2. *Gshare* 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 practical, 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. For marking purposes, the testing of your code will be performed in a DICE machine. Therefore you are strongly encouraged to develop your code on a DICE machine.

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. `CARD_PRAC3_PATH=/afs/inf.ed.ac.uk/group/teaching/card/practical3`
2. `cd $CARD_PRAC3_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:

`$CARD_PRAC3_PATH/tools/pin-3.5-97503-gac534ca30-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* (code provided), *Local*, *Gshare*, *Tournament.* The default predictor is *Always Taken.* To run the *Always Taken* branch predictor, give this command line argument type: `-BP_type always_taken`
2. The number of entries (default 1024) in the PHT (pattern history table) of the Predictor. To run the simulator with 1024 PHT entries, give this command line argument: `-num_BP_entries 1024`
3. The name of the output file (default `BP_stats.out`) with `"-o <filename>"`.
   - For example to generate an output file called MyOutput.out, give this command line argument type: `-o MyOutput.out`

In the path `$CARD_PRAC3_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. 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.

Local Branch predictor

- The Local Branch predictor uses *per-branch* history (i.e. history of each branch considered independently) to make a prediction. The Predictor consists of two structures:
  1) An array of 128 distinct Local History Registers (LHRs), that store the branch history of different branches;
  2) The PHT.

  The 7 Least Significant Bits (LSBs) of the Program Counter (PC) of the branch instruction are used to index into the array with the LHRs and select the relevant LHR to be used for the prediction. Each LHR is composed of as many bits as necessary to index the PHT. For example, if the PHT contains 1024 entries, then each LHR should be 10 bits long. Each n-bit LHR contains the n last outcomes of all branches that map to that LHR (i.e. all branch PCs whose 7 LSBs are identical), with 0 denoting *not taken* and 1 denoting *taken.* All LHRs should be initialized to 0. The number of LHRs must be 128 and must stay constant across all experiments(i.e. 128 LHRs is a fixed parameter, independent of PHT size).

- The selected LHR is used to index into the PHT, which yields the prediction. All PHT entries must be initialized to "11" (i.e. 3).

- To train the predictor after each prediction, the relevant LHR must be updated such that it reflects the new history and the used PHT entry must be strengthened in case of a correct prediction and weakened in case of misprediction.

  - 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'.

- Figure 1 shows a pictorial view of the Local Branch Predictor, with 128 distinct LHRs, each of which consists of 12 bits of branch history, that is used to index into a 4096-entry PHT of 2-bit saturating counters. Students are referred to [1] (Section 4) for more information.

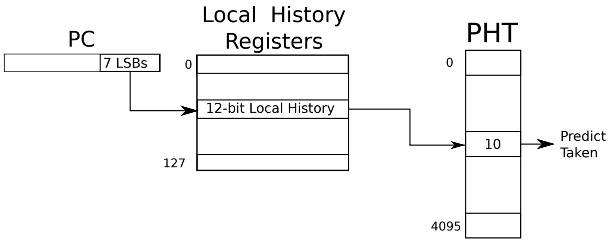

Figure 1: A Local Branch predictor with 128 Local History Registers and a PHT with 4096 2-bit saturating counters

Gshare Branch Predictor

- The Gshare Branch predictor combines a Global History Register (GHR) and the LSBs of the branch's PC to index into the PHT. The PC and the GHR are XORed in order to create the index to the PHT.
- 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. All PHT entries must be initialized to "11" (i.e. 3).
- To train the predictor after each prediction, the GHR must be updated such that it reflects the new global history and the used PHT entry must be strengthened in case of a correct prediction and weakened in case of misprediction.
- Figure 2 shows a pictorial view of the Gshare Branch Predictor with a GHR of 10 bits that gets XORed with the 10 least-significant bits of the PC of the branch in order to index into the PHT that contains 1024 entries of 2-bit saturating counters. Students are referred to [1] (Section 7) for more information. For the purpose of this practical, the number of LSBs used from the branch PC should be equal to the number of bits of the GHR.

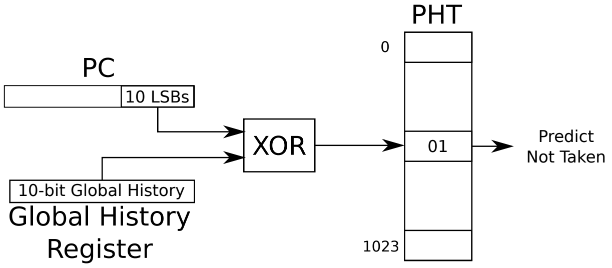

Figure 2: Gshare Branch predictor with a 10-bit GHR and a 1024-entry PHT

Tournament Branch Predictor

- The Tournament predictor is composed of three different predictors: the *Local* predictor, the *Gshare* predictor, and a *meta-predictor* that selects between the Local or the Gshare predictor to provide the actual prediction of the branch outcome.

- The PHT of the meta-predictor is indexed with the LSBs of the Program Counter (PC) of the branch instruction.

- In the meta-predictor, when the prediction bit (i.e. the MSB) of the saturating counter is 0, the Local predictor is selected to supply the prediction. When the MSB is 1, the Gshare predictor is selected. All PHT entries must be initialized to "11" (i.e. 3).

- The Tournament predictor will follow a total update policy. Both the Local and the Gshare 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 (the counter) must be weakened *only if the predictor that was not chosen was correct.* If both the Local and the Gshare predictors were incorrect, there is no update to the meta-predictor.

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

- Figure 3 shows a pictorial view of the Tournament Branch Predictor with a PHT that contains 1024 entries of 2-bit saturating counters. The 2-bit saturating counter indicates which predictor should be used (Local or Gshare) for the branch given its PC.

- As before, the experiments must be run with PHTs of 3 different sizes (128, 1024 and 4096 entries). In each experiment the PHTs of the Local Predictor and the Gshare Predictor must all have the same size. I.e. when the meta-predictor PHT is 1024 entries, then the Gshare PHT is also 1024 entries and the Local Predictor PHT is also 1024 entries.

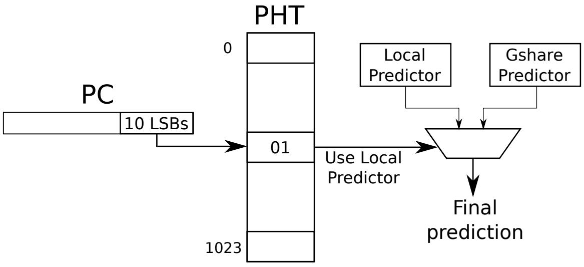

Figure 3: Tournament Branch predictor with a a 1024-entry PHT

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. To help you get started, an implementation of the Always Taken predictor is supplied. You should start with the same code as the one for the Always Taken predictor and modify it to implement new functionality for each specific predictor type.

For the purpose of this practical 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.

2 Marking Scheme

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

1. **Local Branch Predictor (30 marks)**
2. **Gshare Global Branch Predictor (30 marks)**
3. **Tournament Branch Predictor (20 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. Please refrain from using more than 3 to 4 lines for each question.

- Why and how, varying the size of the PHT impacts (or not) the prediction accuracy for each of the predictors?
- What are the advantages and disadvantages of the Local predictor?
- What are the advantages and disadvantages of the Gshare predictor?
- What benefits did you expect from the addition of the Tournament predictor? Did the results match your expectations?

Finally your report must contain 3 graphs (one for each benchmark) summarizing your results by showing the prediction accuracy of all three predictors for the various PHTs. The x axis should denote the PHT size and the y axis the prediction accuracy as a percentage. In each of the graphs the performance of all 3 predictors must be plotted. A sample chart is shown in figure 4.

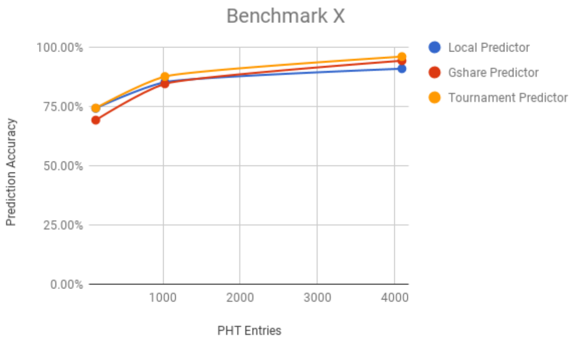

Figure 4: A sample of the chart that must be submitted in the Report. One such chart must exist in the report for each benchmark.

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 (if any). 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.

You are required to submit 1) a .cpp file containing the source code of your Branch Predictors and 2) a .pdf file containing your report. Note that *both files should be named using your student id*. For instance, a student with student-id s1234567 should submit running the following command in the DICE environment:

```
$ submit card cw3 s1234567.cpp s1234567.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 unauthorized 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 vijay.nagarajan@ed.ac.uk **and** Vasilis.Gavrielatos@ed.ac.uk if you have any issues regarding the practical.

References

[1] S. Mcfarling. Combining branch predictors. Technical report, 1993.