**CS 4290: Advanced Computer Organization**

**Project 1**

**Due: 06/02/2020, 11:55 pm, EST**

**Important policies:**

1. Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with University policy.
2. It is acceptable for you to compare your results, and only your results, with other students to help debug your program. It is not acceptable to collaborate either on the code development or on the final experiments.
3. You should do all your work in the C or C++ programming language and should be written according to the C99 or C++11 standards, using only the standard libraries.
4. Unfortunately, experience has shown that there is a very high chance that there are errors in this project description. It is your responsibility to check Pizza often and download new versions of this project description as they become available.

**Project Description**

In this project, we will be designing a simple simulator for predicting the outcomes of branches. Trace files are provided, with each line indicating the PC value of each branch, and the actual outcome. The simulator will process these traces, attempt to predict each branch and verify the prediction.

The project has two parts:

1. Model branch predictors and validate them with simple traces and validate output.
2. Choose and design a good predictor for each of the three programs based on SPEC2006 traces: [https://www.spec.org/cpu2006/Docs/.](https://www.spec.org/cpu2006/Docs/) Then, you will compete with your classmates for the best predictor that works best on all those three programs!

**First Part: Modeling Predictors**

You will be modeling a few different predictor types. The predictors should also accept parameters so that we can vary the configuration:

1. **The predictor type -** You should implement the bimodal predictor, g-share predictor, local history predictor, and two-level adaptive branch predictor.
2. **The number of entries in the predictor table -** You should use the lower bits of an address as the hash to the table. For example, in a 4-entry table, address in 000 (in binary) should go to entry 0, 001 to entry 1, 010 to entry 2, and 011 to entry 3, and 100 to entry 0 again.
3. **Bits per counter** - The counters should be initialized as weakly taken. For example, for a 3-bit counter, the counter value is 0-7, and the initial value should be 4.
4. **Bits for each history register** - The history register should be initialized as all not-taken. For example, 2-bit history should be initialized as 00.

To follow the thought process involved in designing a branch predictor, we will also look at the tradeoffs between storage and performance; more accurate predictors tend to require more storage overhead and attempt to find the right balance.

**Specification of Simulator**

Explanation of functions you need to fill in:

void setup\_predictor(predictor\_type ptype, int num\_entries, int counter\_bits,

int history\_bits, branch\_stats\_t\* p\_stats);

* Subroutine for initializing the branch predictor. You many add and initialize any global or heap variables as needed.
* ptype: Type of predictor
* num\_entries: Depending on type of predictor, number of entries for PHT or History Register Table.
  + Note, for gshare, num\_entries should equal 1 << history\_bits.
* counter\_bits: Number of bits to use for counter
* history\_bits: Number of bits to use in history
  + Note, for gshare, history\_bits should equal log2(num\_entries).
* p\_stats: Structure containing statistics for simulation run

branch\_dir predict\_branch(std::uint64\_t pc, branch\_stats\_t\* p\_stats);

* Subroutine that queries the branch predictor for branch located at pc. The prediction result is returned as an enum branch\_dir, which can be either TAKEN ('T'), or NOT\_TAKEN ('N'). You may update any branch-related stats in p\_stats as needed.

void update\_predictor(std::uint64\_t pc, branch\_dir actual,

branch\_dir predicted, branch\_stats\_t\* p\_stats);

* Subroutine that updates the branch predictor and trains it for the next query. You may update any branch-related stats in p\_stats as needed.

void complete\_predictor(branch\_stats\_t \*p\_stats);

* Subroutine for cleaning up memory and calculating overall system statistics.

**Statistics (output)**

The simulator outputs the following statistics after completion of the run:

1. number of branches
2. number of branches predicted to be taken
3. number of branches predicted to be not-taken
4. number of correct predictions
5. misprediction rate
6. storage overhead

**Validation**

Several test traces will be provided in the test-traces directory, along with the correct output in the output directory. You must run your simulator and debug it until it matches 100% all the statistics in the validation outputs. The validation script, validate-test-traces.sh, will do all that for you. You just need to make it an executable, with chmod +x ./validate-test-trace.sh, or run it with sh ./validate-test-traces.sh. The script will run all the configuration and run a diff between validated output and your output and print out all the differences. If the script does not print anything, you are good to go to the next step. If it prints some diff, I suggest that you first find the configuration that causes that diff and then run that instance alone to debug it.

**Second Part: Choosing & Designing Predictors**

**Per-Trace Predictor**

For each benchmark in the traces directory, design a branch predictor subject to the following goals:

1. You have a total budget of 4096 bits for the entire predictor (including history and counters)
2. The predictor should have the lowest possible misprediction rate.

You may vary any parameter (ptype, num\_entries, counter\_bits, history\_bits).

The method you use to find the best predictor is your choice. You can do pure brute force, or you may come up with an intelligent method based on some insight. I will define a range based on the overall class performance, and if you are within that range, you will get the point for this part.

**[Extra Point] The Global Predictor Competition**

Now, design a branch predictor that has the lowest average miss prediction rate across all the three traces that you have. You only have one branch predictor now that should work the best across all the traces.

**Results**

* **Submit results to this link:** <https://forms.gle/BfEvxFv1J1wkAtEG8>
* **See current auto sorted results from class here:** [Link](https://docs.google.com/spreadsheets/d/1vRahr3UaBDeU8e4dLXtFb7iwoz2ZBSH_ePGcRxDfXMg/edit?usp=sharing)

**Deliverables**

What to hand in via Canvas:

* **branchsim.hpp, branchsim.cpp**
  + The commented source code for the simulator program.
* **<gt\_username>\_prj1.pdf**
  + A document with the design results of the experiments for each trace file, with a persuasive argument of the choices that were made. (An argument may be as simple as an explanation of the search procedure used to find the designs and a statement about why the procedure is complete.) This argument should include output from runs of your program. (There are multiple answers for each trace file, so I will know which students have "collaborated" inappropriately!)

Remember that your code must compile and run on a current variant of Linux (i.e., Debian, Red Hat, Ubuntu) running on an x86 architecture (i.e., Intel or AMD).

**Grading Rubric**

0% You do not hand in anything

+30% Your simulator doesn't run, does not work, but you hand in significant commented code

+55% Your simulator matches the validation outputs

+15% You ran all experiments and found the best-performing predictors

+10% Your predictor is one of the leading predictors in the class! (extra points)