CS 6290: High-Performance Computer Architecture

Project 1

This project is intended to help you understand branch prediction and performance of out-of-order processors. You will again need the "CS6290 Project VM" virtual machine, the same one we used for Project 0. Just like for Project 0, you will put your answers in the red-ish boxes in this Word document, and then submit it in Canvas (the submitted file name should now be PRJ1.docx).

In each answer box, you **must first provide your answer to the actual question** (e.g. a number). You can then use square brackets to provide any explanations that the question is not asking for but that you feel might help us grade your answer. E.g. answer 9.7102 may be entered as **9.7102 [Because 9.71+0.0002 is 9.7102].** For questions that are asking "why" and/or "explain", the correct answer is one that concisely states the cause for what the question is describing, and also states what evidence you have for that. Guesswork, even when entirely correct, will only yield up to 50% of the points on such questions.

Additional files to upload are specified in each part of this document. **Do not archive** (zip, rar, or anything else) the files when you submit them – each file should be uploaded separately, with the file name specified in this assignment. You will lose up to 20 points for not following the file submission and naming guidelines, and if any files are missing **you will lose all the points for answers that are in any way related to a missing file** (yes, this means an automatic zero score if the PRJ1.docx file is missing). Furthermore, if it is not VERY clear which submitted file matches which requested file, we will treat the submission as missing that file. The same is true if you submit multiple files that appear to match the same requested file (e.g. several files with the same name). In short, if there is any ambiguity about which submitted file(s) should be used for grading, the grading will be done as if those ambiguous files were not submitted at all.

Most numerical answers should have **at least two decimals** of precision. Speedups should be computed to **at least 4 decimals** of precision, using the number of cycles, not the IPC (the IPC reported by report.pl is rounded to only two decimals). You lose points if you round to fewer decimals than required, or if you truncate digits instead of correctly rounding (e.g. a speedup of 3.141592 rounded to four decimals is 3.1416, not 3.1415).

As explained in the course rules, **this is an individual project**: **no collaboration with other students or anyone else is allowed**.

Part 1 [20 points]: Configuration of the Branch Predictor

The hardware of the simulated machine is described in the configuration file. In this project we will be using the cmp4-noc.conf configuration file again, but this time we will modify this file so this is a good time to make a copy so we can restore the original configuration when we need it.

The processors (cores) are specified in the "cpucore" parameter near the beginning of the file. In this case, the file specifies that the machine has 4 identical cores numbered 0 through

3 (the procsPerNode parameter is 4), and that each core is described in section [issueX]. Going to section [issueX], we see that a core has a lot of parameters, among which we see that the clock frequency is set at 1GHz, that this is an out-of-order core (inorder set to false) which fetches, issues, and retires up to 2 instructions per cycle (the "issue" parameter is set to two earlier in the file). The core has a branch predictor described in the [BPredIssueX] section, fetches instructions from a structure called "IL1" described in the [IMemory] section (this is specified by the instrSource parameter, and reads/writes data from a structure called "DL1" described in the [DMemory] section. In this part of this project, we will be modifying the branch predictor, so let's take a closer look at the [BPRedIssueX] section. It says that the type of the predictor is "Hybrid" (which does not tell us much), and then specifies the parameters for this predictor.

The "Hybrid" predictor is actually a tournament predictor. You now need to look at its source code (which is in BPred.h and BPRed.cpp files in the ~/sesc/src/libcore/ directory) and determine which of the parameters in the configuration file controls which aspect of the predictor. Hint: the "Hybrid" predictor is implemented in the BPHybrid class, so its constructor and predict method will tell you most of what you need to find out.

A) The meta-predictor in this hybrid predictor is a table that has **2048** entries and each entry is a **1** –bit counter. This meta-predictor decides, based on the PC address (i.e. the address from which we fetched the branch instruction), whether to make the prediction using a simple (no history) array of counters, or to use a (is it local or global?) **global** history predictor. The simpler (non-history) predictor uses **2** -bit counters, and has **2048** of them (this number is specified using a parameter label **localsize** in the BPredIssueX section of the configuration file). The history-based predictor has **8** bits of history, which are /BPred has **2048** entries (this number of entries is specified in the configuration file using parameter label **l2Size**), and each entry is

B) a **2 (states 0,1,2 as saturate =1, Max =2 bits = 1)** -bit counter.

Part 2 [30 points]: Changing the Branch Predictor

Now we will compare some branch predictors. The LU benchmark we used in Project 0 does not really stress the branch predictor, so we will use the raytrace benchmark:

```
cd ~/sesc/apps/Splash2/raytrace
```
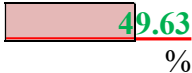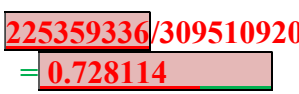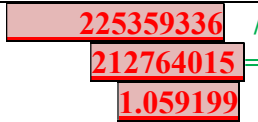
```
make
```

Now it is time to do some simulations:

A) Simulate the execution of this benchmark using the unmodified cmp4-noc configuration (with the "Hybrid" predictor). The following should all be a single command line, which has a space before -ort.out. As before, the dashes in

this command line should be the minus character but a copy-paste might result in something else that looks similar but is not a minus character, so be careful is you are copy-pasting.

```
~/sesc/sesc.opt -f HyA -c ~/sesc/confs/cmp4-noc.conf -ort.out
-ert.err raytrace.mipseb -p1 -m128 -a2 Input/reduced.env
```

Then we will modify the configuration file, so make a copy of it if you did not do this already. Then change the configuration to model an oracle (perfect) direction predictor by changing the "type" of the predictor from "Hybrid" to "Oracle", then and re-run the simulation (change the -f parameter to -f OrA so the simulation results are written to a different file). Note that overall branch prediction accuracy is not perfect in this case – only the direction predictor is perfect, but the target address predictor is a (non-oracle) BTB! After that, configure the processor to use a simple predict-not-taken predictor (type="NotTaken") and run the simulation again (now using -f NTA). Submit the three simulation report files (**sesc_raytrace.mipseb.HyA**, **sesc_raytrace.mipseb.OrA**, and **sesc_raytrace.mipseb.NTA**) in Canvas along with the other files for this project.

B) In the table below, for each simulation fill in the overall accuracy (number under BPred in the output of report.pl), the number of cycles, and the speedup relative to the configuration that uses the Hybrid predictor.

	BPred Accuracy	Cycles	Speedup vs. Hybrid
NotTaken	49.63 %	309510920	225359336/309510920 = 0.728114
Hybrid	85.67 %	225359336	1
Oracle	89.26 %	212764015	225359336 / 212764015 = 1.059199

C) Now change the processor's renameDelay parameter (in the issuesX section of the configuration file) from 1 to 8. This makes the processor's pipeline 7 stages longer. Repeat the three simulations, submit the simulation report files (**sesc_raytrace.mipseb.HyC**, **sesc_raytrace.mipseb.OrC**, and **sesc_raytrace.mipseb.NTC**) in Canvas along with the other files for this project.

D) In the table below, fill in the number of cycles with each type of predictor from Part A (simulations with the default pipeline depth) and from Part C (when the pipeline is 7 stages deeper), then compute **the speedup of shortening the pipeline for each type of predictor**, assuming that the clock cycle time stays the same (so the speedup can be computed using the number of cycles instead of execution time).

	Cycles w/ renameDelay=1	Cycles w/ renameDelay=8	Speedup of changing renameDelay from 8 to 1
NotTaken	309510920 A	391021187 B	391021187/ 309510920 = 1.263351829 B/A
Hybrid	225359336 C	247096207 D	247096207 /225359336 = 1.09645 D/C
Oracle	212764015	228653975	228653975 /212764015 = 1.07468

E) The results in Part D) lead us to conclude that better branch prediction becomes (more or less?) __more__ important when the processor's pipeline depth increases.

Explain why:

Above calculations are based on speedup = executionTime_old/executionTime_new. Since we go from 8 to 1 hence old is rename delay 8 while new is rename delay 1. Hence above calculations are based on this reasoning.

More depth in pipeline, the branch detection is pushed to later stages in pipeline resulting in increase of penalty per branch. Additionally, there is multiple instructions per cycle ($issue in config is 2). This also means that there is a lot of waste from a misprediction, especially if it is a deep (renameDelay, more stages) and/or wide (issue, more instructions per cycle) pipeline.
This is evident from our data in section D.

Based on our markings of A,B,C,D what can we say about

Relationship of Hybrid predictor vs Not Taken keeping rename delay

We do know from above data that

=> SpeedUp8To1NT > SpeedUp8To1HYD
=> B / A > D / C
=> B / D > A / C
=> 391021187/247096207   > 309510920 / 225359336
=> 1.582465355 > 1.373410684
=> SpeedUp From Not Taken to Hybrid with bigger pipeline is way better than Speed from NotTaken to hybrid with smaller pipeline.

 This implies Hybrid (HYD) branch predictor which is better branch predictor than NotTaken (NT) performs way better than NT and is more effective when pipeline stages increase or penalty per mispredicted branch increases.

F) From simulation results you have collected up to this point, there are at least two good ways to estimate how many cycles are wasted when we have a branch misprediction in the processor that has the default pipeline depth, i.e. what the branch misprediction penalty (in cycles) was for simulations in Part A). Enter your best estimate here HyA-19.80370423, NTA-15.855035 , OrA-18.2615465759 and then explain how you got it:

Equation 1 : Total no of Cycles wasted due mispredictions from report
($MisBr/100.0) * $Cycles …… (1)

Equation 2: Total no of instructions that are mispredicted
Total Branch inst(BJ,Call, ret etc)% * total inst * predictor inaccuracy
($BJ/100) * $nInst * ((100-$Total)/100) ……. (2)

Equation 3: Total cycles wasted based on equation (2)
Equation (2) * $AVG_CYCLES_WASTED_PER_BR_MISSPRED …. (3)

Now   (1)   =   (3)   so   solving   for
$AVG_CYCLES_WASTED_PER_BR_MISSPRED (X)

X = (($MisBr/100.0) * $Cycles) / (($BJ/100) * $nInst * ((100-$Total)/100))

Now X value for each of predictors is as follows:-

Not Taken Predictor
NTA

Part 3 [50 points]: Which branches tend to be mispredicted?

In this part of the project we again use the cmp4-noc configuration. **You should change it back to its original content, i.e. what it had before we modified it for Part 2**. We will continue to use the Raytrace benchmark with the same parameters as in Part 2.

Our goal in this part of the project is to determine for each instruction in the program how many times the direction predictor (Hybrid or NotTaken) correctly predicts and how many times it mispredicts that branch. The number of times the static branch instruction was completed can be computed as the sum of two (correct and incorrect predictions) values for that static branch instruction. You should change the simulator's code to count correct and incorrect predictions for each static branch instruction separately, and to (at the end of the simulation) print out the numbers you need to answer the following questions. The printing out should be in the order in which the numbers are requested below, and your code should not be changing the simulation report in any way. Then you should, of course, run the simulation and get the simulation results with the Hybrid and also with the NT predictor.

G) In both simulations, the number of static branch instructions that are completed at least once but fewer than 20 times (i.e. between 1 and 19 completions) is `1056`, the number of static branch instructions with 20 to 199 completions is `473`, the number of static branch instructions with 200 and 1999 completions is `155`, and the number of static branch instructions with 2000+ completions is `654`.

H) The accuracy for the direction predictor, computed separately for the four groups of static branch instructions (1-19, 20-199, 200-1999, and 2000+ completions), is a follows:

	Hybrid Accuracy	NT Accuracy
1-19	69.7665	41.4251
20-199	94.6257	33.351
200-1999	97.321	42.5981
2000+	95.1442	44.4381

I) We cannot run the raytrace benchmark with a much larger input because the simulation time would become excessive. But if we did, concisely state what would happen to the overall direction predictor accuracy of the Hybrid predictor and of the NT predictor, and then explain why the predictor accuracy should change in the way you stated.

The question specifically talks of larger input. Lets assume raytrace a program, would execute based on certain input and would need to parse the input to actually do some operation. The iron law of performance relates to following

CPU Time = # of instructions x cycle per instruction x clock cycle time

Now if we have larger input then cycle per instruction and clock cycle time both would not change however no of instructions would/may change. Hence overall CPU time will increase.
However, what is impact on predictor accuracy of Hybrid Predictor and NT predictor. Let's assume that increasing the input size results longer loops in ion the design. In such a case the no of static branch instructions will not

change however, the branch prediction accuracy should improve. Lets break this down in 2 cases:-

Case 1: NOT TAKEN >> TAKEN → Firstly looking at an NT predictor. NT predictor is ends up parsing a loop more than it previously did and is a Not taken branches >> taken branches in assembly then there will be clear increase in nHits. This would increase overall accuracy. This will also be true for Hybrid Predictor.

Case 2: TAKEN >> NON TAKEN → However, if assembly code is in such a way that there are more taken branches than non-taken (taken >> non-taken) the NT predictor will go the other way. It would have more nMiss and thereby poorer accuracy. However Hybrid predictor in this case would agnostic as it also has history predictor and hence history predictor would train itself for more taken branches thereby possibly increasing accuracy on similar note as Case 1.

Case 3: TAKEN ~ NON TAKEN (equivalent) → In this case order of increase in branch instructions has resulted in nearly similar TAKEN and NON TAKEN branches, hence both predictors may show similar accuracy numbers as before.

J) Submit the **BPred.h** and **BPred.cpp** files that you have modified to produce the numbers you needed for Part 3 of this project. If you have modified any other source code in the simulator, create an **OtherCode.zip** file that includes these files and submit it, too. Also submit the output of the simulator for the two runs (as **rt.out.Hybrid** and **rt.out.NT**) Note that there is no need to submit the simulation report files (these should be the same as those from Part A).