# 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[[1]](#footnote-1) 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.

1. The meta-predictor in this hybrid predictor is a table that has 2048 entries and each entry is a 2 –bit counter. This meta-predictor decides, based on the PC address (i.e. the address from which we fetched the branch/jump 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 combined with the PC address to index into an array that has  
    2048 entries (this number of entries is specified in the configuration file using parameter label l2size ), and each entry is a 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:

1. 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 -m128 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.

1. 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** | 46.62 % | 218337352 C | 0.7150 X |
| **Hybrid** | 84.70 % | 156109008 C | 1.0000 X |
| **Oracle** | 88.15 % | 147629257 C | 1.0574 X |

1. Now change the processor’s renameDelay parameter (in the issuesX section of the configuration file) from 1 to 7. This makes the processor’s pipeline 6 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.
2. 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 6 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=7** | **Speedup of changing renameDelay from 7 to 1** |
| **NotTaken** | 218337352 C | 269050018 C | 1.2323 X |
| **Hybrid** | 156109008 C | 169916084 C | 1.0884 X |
| **Oracle** | 147629257 C | 158152936 C | 1.0713 X |

1. The results in Part D) lead us to conclude that better branch prediction becomes (fill in either “more” or “less”) more important when the processor’s pipeline depth increases. Now explain why the importance of branch prediction changes that way with pipeline depth:

We know from lectures that the overall CPI = CPIprogram + misprediction % \* penalty/misprediction. With a deeper pipeline, there are more stages that can hold instructions. If we assume that the branch resolution stage also becomes a later stage in a deeper pipeline (for example, branch resolution going from 3rd stage to 15th stage), then the penalty for mispredictions will be significantly higher. In the example case I gave, this would be a penalty of 2 cycles versus a penalty of 14 cycles. If there are also multiple instructions per cycle, this penalty becomes even greater.

If there are the same number of branch misses in two programs run on different depths, the differences will be noticeable. For my example shallow pipeline, there will be 100 misses \* 2 cycles penalty, or 200. For the deep one, there will be 100 misses \* 14 cycles penalty, 1400 cycles wasted!

Branch prediction becomes incredibly important, as a better predictor in deep pipelines means that we can avoid having to deal with a large amount of costly mispredictions. Thus, we get a slowdown (fractional speedup) when we use the deep pipeline with renameDelay = 7 instead of the shallow pipeline with renameDelay = 1. This slowdown is the inverse of the speedup from going from the deep pipeline to shallow.

1. 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 33.28 and then explain how you got it:

The SESC processor is configured to issue 2 instructions per cycle (issue = 2 in the .conf file). The program we run is also only single core. Thus, our ideal CPIprogram should be equal to 0.5. To figure out the penalty, we have to use the equation for CPI again, overall CPI = CPIprogram + misprediction % \* penalty/misprediction, in order to solve for the penalty.

From the Hybrid part A report, our IPC value is 0.91. The inverse value of the IPC, which is CPI, is 1.0989. Our total prediction accuracy is 85.95%. Our % of jumps is 12.81%.

Thus, 1.0989 = 0.5 + (100%-85.95%) \* 12.81% \* penalty

Solving this equation for penalty, we get that Penalty = 33.28 cycles

### 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/jump. The number of times the static branch/jump instruction was completed can be computed as the sum of two (correct and incorrect predictions) values for that static branch/jump instruction. You should change the simulator’s code to count correct and incorrect predictions for each static branch/jump 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.

1. In both simulations, the number of static branch/jump instructions that are completed at least once but fewer than 20 times (i.e. between 1 and 19 completions) is 1051 , the number of static branch/jump instructions with 20 to 199 completions is 473 , the number of static branch/jump instructions with 200 and 1999 completions is 163 , and the number of static branch/jump instructions with 2000+ completions is 644 .
2. The accuracy for the direction predictor, computed separately for the four groups of static branch/jump instructions (1-19, 20-199, 200-1999, and 2000+ completions), is a follows:

|  |  |  |
| --- | --- | --- |
|  | **Hybrid Accuracy** | **NT Accuracy** |
| **1-19** | 0.698304 C | 0.412763 C |
| **20-199** | 0.946237 C | 0.335116 C |
| **200-1999** | 0.966314 C | 0.4230 C |
| **2000+** | 0.956905 C | 0.467128 C |

1. 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.

Generally speaking, the overall accuracies for hybrid predictors should go up with larger input. This is particularly true in a context like matrix multiplication, where larger input matrices are being processed by the program. This increase in accuracy is due to the longer for-loops that contain predictable branch instructions. In the history-based hybrid predictor, the architecture allows it to take advantage of repeated for-loop sequences. The history will allow for better predictions, increasing accuracy. With the Not Taken predictor, the larger input usually lowers accuracy. This is due to the lack of history and constant not taken prediction. In for-loops, branches are usually taken to keep the loop going. With larger inputs, for-loops tend to be bigger, making a misprediction on every loop. Thus, the accuracy should go down.

1. 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).

1. Note that in this project “branch” is used to refer to all instructions that affect control flow, which includes branch instructions, but also jumps, function calls, etc. Also, “branch prediction” refers to the overall prediction of the address of the next instruction, not just the prediction of direction (taken or not taken). [↑](#footnote-ref-1)