University of Central Florida
Department of Computer Science
CDA 5106: Fall 2021
Machine Problem 2: Branch Prediction

1. Ground rules

i. All students must work alone.
ii. Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TAs will scan source code through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt severely: 0 on the project and referral to the Office of Student Conduct for sanctions.
iii. You must do all your work in C/C++ or Java language. Exceptions must be approved. The C language is fine to use as that is what many students are trained in. Basic C++ extensions to the C language (classes instead of structs) are encouraged (but by no means required) because it enables more straightforward code reuse.
iv. Use of Linux environment is required. This is the platform where the TAs will compile and test your simulator.

2. Project Description

In this project, you will construct a branch predictor simulator and use it to design branch predictors well suited to the SPECint95 benchmarks.

3. Simulator Specification

3.1 N-Bit Branch Predictor

3.1.1 Smith n-bit counter predictor

Model a smith n-bit counter predictor with parameters (a,b) where

- a represents the actual branch prediction outcome
- b represents the counter bit.

The initial values of the counter will be initialized to 1, 2, 4 and 8 when the number of bits is 1, 2, 3 and 4 respectively.

<u>Steps:</u>
When you get a branch from the trace file, there are three steps:

(1) Determine the branch's **actual outcome**

(2) Make a prediction based on the content of the counter. In a 3-bit counter predictor, for instance, if the counter value is greater than or equal to 4, then the branch is predicted *taken*, else it is predicted *not taken*.
(3) Update the branch predictor based on the branch's actual outcome. The branch's counter is incremented if the branch was taken, decremented if the branch was not taken. In a 3-bit counter predictor, for instance, the counter *saturates* at the extremes (**0** and **7**), however.

3.2 GShare Branch Predictors

Model a *gshare* branch predictor with parameters {m,n}, where:

- *m* is the number of low-order PC bits used to form the prediction table index. **Note:** discard the lowest two bits of the PC, since they are always zero, i,e., use bits ***m+1*** through **2** of the PC.
- *n* is the number of bits in the global branch history register. **Note:** *n<=m*. **Note:** *n* may equal zero, in which case we have the simplest *bimodal* branch predictor.

3.2.1. n=0: bimodal branch predictor

When n=0, the *gshare* predictor reduces to a simple *bimodal* predictor. In this case, the index is based on only the branch's PC, as shown in **Figure 1**.

Entry in the prediction table

An entry in the prediction table contains a single 3-bit counter. All entries in the prediction table should be initialized to **4** ("weakly taken") when the simulation begins.

Regarding branch interference

Different branches may index the same entry in the prediction table. This is called "interference". Interference is not explicitly detected or avoided: it just happens. (There is no tag array, no tag checking, and no "miss" signal for the prediction table!)

Steps:

When you get a branch from the trace file, there are three steps:

(4) Determine the branch's **index** into the prediction table.
(5) Make a prediction. Use **index** to get branch's counter from the prediction table. If the counter value is greater than or equal to **4**, then the branch is predicted *taken*, else it is predicted *not-taken*.
(6) Update the branch predictor based on the branch's actual outcome. The branch's counter in the prediction table is incremented if the branch was taken, decremented if the branch was not taken. The counter *saturates* at the extremes (**0** and **7**), however.

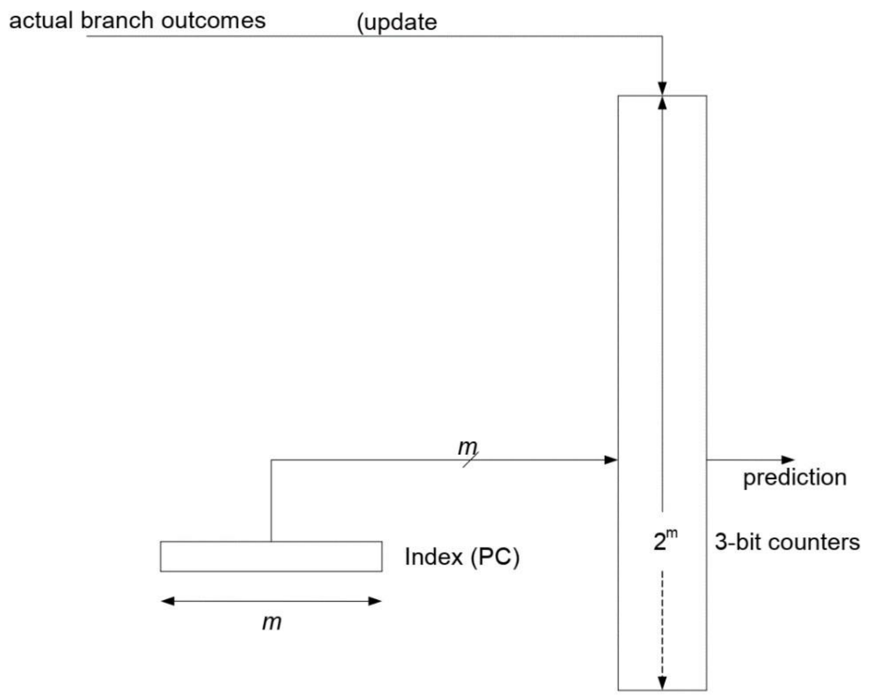

Figure 1. Bimodal branch predictor.

3.2.2. n>0: gshare branch predictor

When n>0, there is an n-bit global branch history register. In this case, the index is based on both the branch's PC and the global branch history register, as shown in **Figure 2.** The global branch history register is initialized to all zeroes (00…0) at the beginning of the simulation.

Steps:
When you get a branch from the trace file, there are three steps:

(1) Determine the branch's **index** into the prediction table. Figure 2 shows how to generate the index: the current n-bit global branch history register is XORed with the lowermost n bits of the index (PC) bits.
m-bit index= concatenate (m-n MSB bits of lowermost m bits from PC,
XOR(n-bit global branch history, LSB n bits of PC))
Example: Let PC= 302d28, m=9, n=3 , Global History register value: 000
Steps to get branch's index into the prediction table
1. *Remove last 2 bits of PC as mentioned in section 3.2 ==> 0011 0000 0010 1101 0010 1000*
2. *Get the lowermost m bits from the above value===> 101 0010 10*
3. *m-n MSB bits of the above value ===> 101 001*
4. *n bits of Global History register ==> 000*
5. *n LSB bits of PC ==> 010*
6. *XOR values from 4 and 5 ==> 010*
7. *Concatenate values from 3 and 6 to get the m-bit index ==> 101 001 010*

(2) Make a prediction. Use **index** to get the branch's counter from the prediction table. If the counter value is greater than or equal to **4,** then the branch is predicted *taken*, else it is predicted *not-taken.*
(3) Update the branch predictor based on the branch's actual outcome. The branch's counter in the prediction table is incremented if the branch was taken, decremented if the branch was not taken. The counter *saturates* at the extremes (**0** and **7**), however.
(4) Update the global branch history register. Shift the register right by 1bit position and place the branch's actual outcome into the most significant bit position of the register.

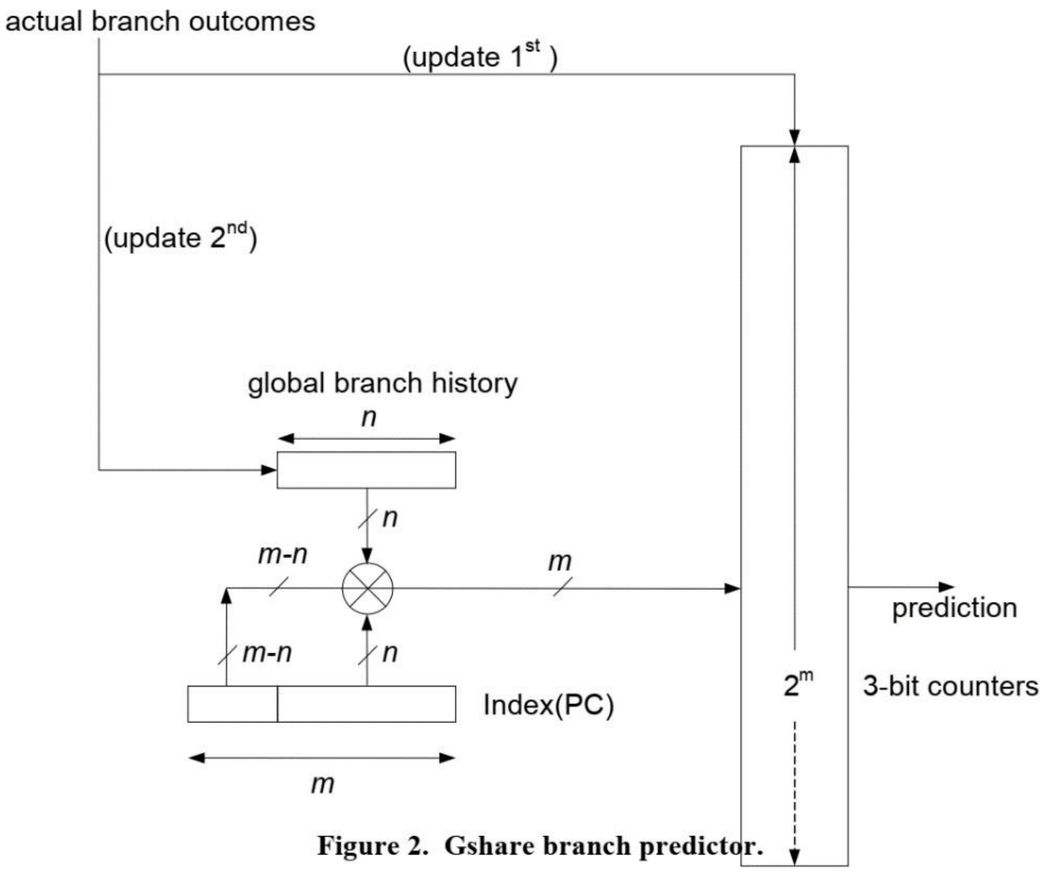

Figure 2. Gshare branch predictor.

4. Inputs to Simulator

The simulator reads a trace file in the following format:

```
<hex branch PC> t|n
<hex branch PC> t|n
...
```

Where

- `<hex branch PC>` is the address of the branch instruction in memory. This field is used to index the predictors.
- `"t"` indicates the branch is actually taken (Note! *Not* that it is predicted taken!). Similarly, `"n"` indicates the branch is actually not-taken.

Example:

```
00a3b5fc t
00a3b604 t
00a3b60c n
...
```

5. Outputs from Simulator

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

a. Number of accesses to the predictor (i,e., number of branches)
b. Number of branch mispredictions (predicted *taken* when *not-taken*, or predicted *not-taken* when *taken*)
c. Branch predictions rate (# of mispredictions / # of branches)

6. Validation and Other Requirements

6.1. Validation requirements

Sample simulation outputs will be provided on the webcourses. These are called "validation runs". You must run your simulator and debug it until it matches the validation runs.

Each validation run includes:

1. The branch predictor configuration

2. The final contents of the branch predictor
3. All measurements described in Section 5.

Your simulator must print outputs to the console (i,e., to the screen). (Also see Section 6.2 about the requirement.)

Your output must match both numerically and in formatting, because the TAs will literally "diff" your output with the correct output. You must confirm correctness of your simulator by following these two steps for each validation run:

1. Redirect the console output of your simulator to a temporary file. This can be achieved by placing `> your_output_file` after the simulator command.
2. Test whether or not your outputs match properly, by running this unix command: `diff -i -w <your_output_file> <posted_output_file>`
   The "-i –w" tags tell "diff" to treat upper-case and lower-case as equivalent and to ignore the amount of whitespaces between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the validation runs have whitespace.

6.2. Compiling and running simulator

You will hand in source code and the Tas will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see section "Grading").

1. You must be able to compile and run your simulator on Linux machine. This is required so that the Tas can compile and run your simulator.
2. Along with your source code, you must provide a **Makefile** that automatically compiles the simulator. This Makefile must create a simulator named "sim". The TAs should be able to type only "make" and the simulator will successfully compile. The TAs should be able to type only "make clean" to automatically remove object files and the simulator executable. An example Makefile is posted in the **MachineProblem2Fall2021.zip** folder, which you can copy and modify for your needs.
3. Your simulator must accept command-line arguments as follows:

   To simulate a smith n-bit counter predictor: `sim smith <B> <tracefile>`, where B is the number of counter bits used for prediction.

   To simulate a bimodal predictor: `sim bimodal <M2> <tracefile>`, where M2 is the number of PC bits used to index the bimodal table.

   To simulate a gshare predictor: `sim gshare <M1> <N> <tracefile>`, where M1 and N are the number of PC bits and global branch history register bits used to index the gshare table, respectively.

(`<tracefile>` is the filename of the input trace.)

4. Your simulator must print outputs to the console (i.e., to the screen). This way, when a TA runs your simulator, he/she can simply redirect the output of your simulator to a filename of his/her choosing for validating the results.

6.3. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator efficient is also important for a couple of reasons.

First, the TAs need to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in **2 minutes** or less. If your simulator takes longer than 2 minutes to finish a single run, please see the TAs as they may be able to help you speed up your simulator.

Second, you will be running many experiments: many branch predictor configurations and multiple traces. Therefore, you will benefit from implementing a simulator that is reasonably fast.

[The following applies to students working with C.]

One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the **MachineProblem2Fall2021.zip** folder includes the -O3 optimization flag.

Note that, when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without -O3 and with -g. Optimization includes register allocation. Often, register-allocated variables are not displayed properly in debuggers, which is why you want to disable optimization when using a debugger. The -g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, etc. When you are done debugging, recompile with -O3 and without -g, to get the most efficient simulator again.

7. Tasks and Grading Breakdown

PART 1: SMITH N-BIT COUNTER PREDICTOR

(a) Match the three validation runs "val_smith_1.txt", "val_smith_2.txt" and "val_smith_3.txt", posted on the Webcourses for the SMITH N-BIT COUNTER PREDICTOR. You must match all validation runs to get credit for the experiments with the smith n-bit counter predictor, however.
(b) Simulate SMITH N-BIT COUNTER PREDICTOR configurations for 1 <= b<=6. (Note: The initial counter values will be 16 and 32 when b is 5 and 6 respectively)
(c) Use the traces in the trace directory.

Graphs: produce one graph for each benchmark. Graph title: “<benchmark>, smith”. Y-axis: branch misprediction rate. X-axis: b. Per graph, there should be only one curve consisting of 6 datapoints (connect the datapoints with a line).

PART 2: BIMODAL PREDICTOR

(d) Match the three validation runs “val_bimodal_1.txt”, “val_bimodal_2.txt” and “val_bimodal_3.txt”, posted on the Webcourses for the BIMODAL PREDICTOR. You must match all validation runs to get credit for the experiments with the bimodal predictor, however.

(e) Simulate BIMODAL PREDICTOR configurations for 7 <= m<=12. Use the traces in the trace directory.
Graphs: produce one graph for each benchmark. Graph title: “<benchmark>, bimodal”. Y-axis: branch misprediction rate. X-axis: m. Per graph, there should be only one curve consisting of 6 datapoints (connect the datapoints with a line).

PART 3: GSHARE PREDICTOR

(a) Match the three validation runs “val_gshare_1.txt”, “val_gshare_2.txt” and “val_gshare_3.txt” posted on the webcourses for the GSHARE PREDICTOR. You must match all validation runs to get credit for the experiments with the gshare predictor, however.

(b) Simulate GSHARE PREDICTOR configurations for 7 <= m<=12 and 2<=n<=m, n is even. Use the traces in the trace directory.
Graphs: produce one graph for each benchmark. Graph title: “<benchmark>, gshare”. Y-axis: branch misprediction rate. X-axis: *m*. Per graph, there should be a total 27 datapoints plotted as a family of 6 curves. Datapoints having the same value of n are connected with a line, i.e., one curve for each value of n. Note that not all curves have the same number of datapoints.

30 points: Substantial Programming effort

Item	**Points**
Substantial simulator turned in	**30**

50 points: A working simulator with matched validation runs

Item	**Points**
val_smith_1.txt	4
val_smith_2.txt	4
val_smith_3.txt	4
val_bimodal_1.txt	4
val_bimodal_2.txt	4
val_bimodal_3.txt	4

Mystery run A	7
val_gshare_1.txt	4
val_gshare_2.txt	4
val_gshare_3.txt	4
Mystery run B	7

20 points: Report

Item	Points
Graph for Smith n-Bit Counter with configurations for 1<=b<=6	**4**
Graph for Bimodal Predictor with configurations for 7 <= m<=12	**6**
Graph for gshare predictor configurations for 7 <= m<=12 and 2<=n<=m, n is even	**10**

8. What to Submit via Webcourses

You must hand in a single zip files called **project2.zip.** Below is an example showing how to create **project2.zip** from an EOS Linux machine. Suppose you have a bunch of source code files (*.cc, *.h), the Makefile, and your project report (report.doc).

```
zip project2 *.cc *.h Makefile report.doc
```

project2.zip must contain the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.

1. **Project report.** This must be a single PDF document named **report.pdf.** (*No* Word document please.) The report must include the following:
   - A cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A sample cover page is posted in the project directory.
   - See Section 7 for the required content of the report.
2. **Source code.** You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files etc.
3. **Makefile:** See Section 6.2, item #2, for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

9. Penalties

Various deductions (out of 100 points):

Up to -10 points: for not complying with specific procedures. Follow all procedures very carefully to avoid penalties.

Cheating: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a ‘0’ for the project and other disciplinary actions.