NC State University

Department of Electrical and Computer Engineering

ECE 463/563: Fall 2021 (Rotenberg)

Project #2: Branch Prediction (Version 1.0)

Due: Wed., November 3, 2021, 5:00 PM
Late projects will only be accepted until: Wed., November 10, 2021, 5:00 PM

1. Groundrules

1. All students must work alone. The project scope is reduced (but still substantial) for ECE 463 students, as detailed in this specification.
2. Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TAs will scan source code (from current and past semesters) through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely: 0 on the project and referral to the Office of Student Conduct.
3. A Moodle message board will be provided for posting questions, discussing and debating issues, and making clarifications. It is an essential aspect of the project and communal learning. Students must not abuse the message board, whether inadvertently or intentionally. Above all, never post actual code on the message board (unless permitted by the TAs/instructor). When in doubt about posting something of a potentially sensitive nature, email the TAs and instructor to clear the doubt.
4. It is recommended that you do all your work in the C, C++ or Java languages. Exceptions must be approved by the instructor.
5. You will submit, validate, and SELF-GRADE your project via Gradescope; the TAs will only manually grade the report and assess any late penalty. While you are developing your simulator, you are required to frequently check via Gradescope that your code compiles, runs, and gives expected outputs with respect to your current progress. This is necessary to resolve porting issues in a timely fashion (*i.e.*, well before the deadline), caused by different compiler versions in your programming environment and the Gradescope backend. This is also necessary to resolve non-compliance issues (*i.e.*, how you specify the simulator's command-line arguments, how you format the simulator's outputs, *etc.*) in a timely fashion (*i.e.*, well before the deadline).

2. Project Description

In this project, you will construct a branch predictor simulator and use it to evaluate different configurations of branch predictors.

3. Simulator Specification

3.1. ECE 463 and ECE 563 students: 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 these 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**: $n \leq m$. **Note**: *n* may equal zero, in which case we have the simple *bimodal* branch predictor.

3.1.1. 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 Fig. 1 below.

Entry in the prediction table:
An entry in the prediction table contains a single 2-bit counter. All entries in the prediction table should be initialized to **2** ("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:
(1) Determine the branch's **index** into the prediction table.
(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 **2**, 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 **3**), however.

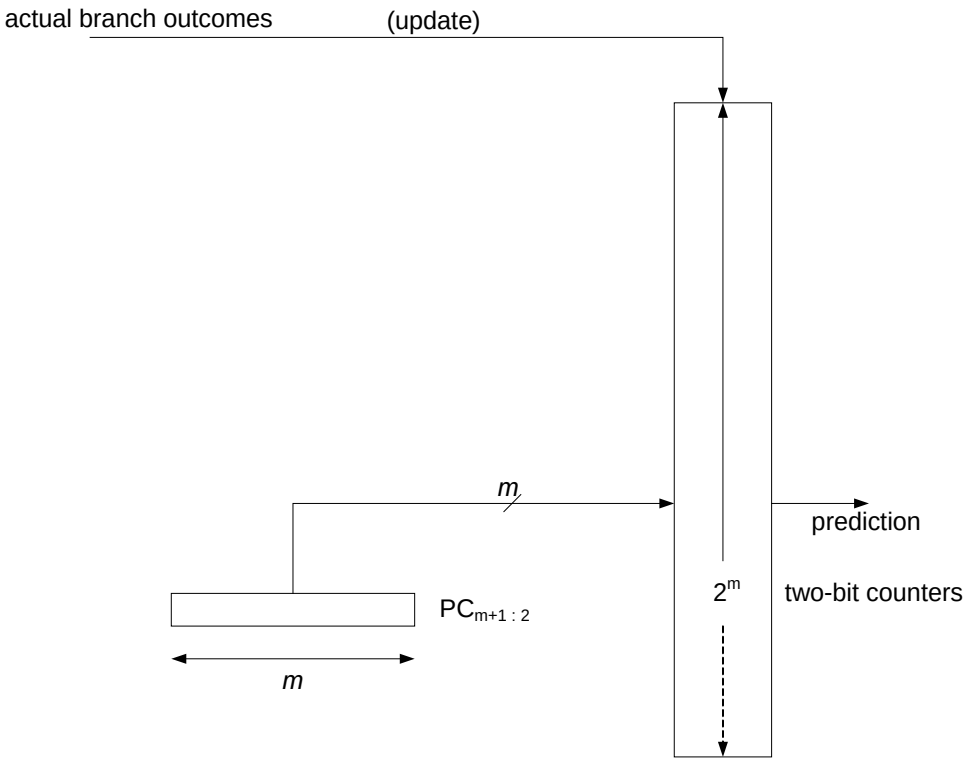

Figure 1. Bimodal branch predictor.

3.1.2. 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 Fig. 2 below. 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 four steps:

(1) Determine the branch's **index** into the prediction table. Fig. 2 shows how to generate the index: the current n-bit global branch history register is XORed with the uppermost n bits of the m PC bits.

(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 **2**, 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 **3**), however.

(4) Update the global branch history register. Shift the register right by 1 bit position, and place the branch's actual outcome into the most-significant bit position of the register.

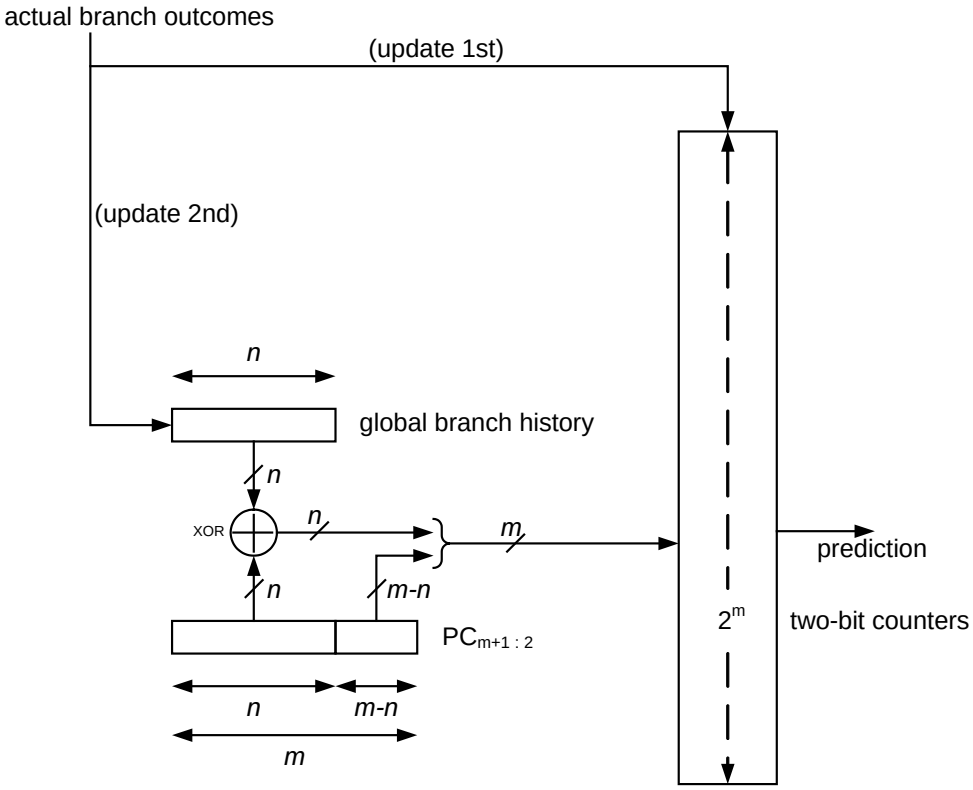

Figure 2. Gshare branch predictor.

3.2. ECE 563 students only: Hybrid branch predictor

Model a **hybrid predictor** that selects between the *bimodal* and the *gshare* predictors, using a chooser table of 2^k 2-bit counters. All counters in the chooser table are initialized to **1** at the beginning of the simulation.

Steps:

When you get a branch from the trace file, there are six top-level steps:

(1) Obtain two predictions, one from the *gshare* predictor (follow steps 1 and 2 in Section 3.1.2) and one from the *bimodal* predictor (follow steps 1 and 2 in Section 3.1.1).

(2) Determine the branch's **index** into the chooser table. The index for the chooser table is bit ***k*+1** to bit **2** of the branch PC (*i.e.*, as before, discard the lowest two bits of the PC).

(3) Make an overall prediction. Use **index** to get the branch's chooser counter from the chooser table. If the chooser counter value is greater than or equal to **2**, then use the prediction that was obtained from the *gshare* predictor, otherwise use the prediction that was obtained from the *bimodal* predictor.

(4) Update the selected branch predictor based on the branch's actual outcome. Only the branch predictor that was selected in step 3, above, is updated (if *gshare* was selected, follow step 3 in Section 3.1.2, otherwise follow step 3 in Section 3.1.1).

(5) Note that the *gshare's* global branch history register must always be updated, even if *bimodal* was selected (follow step 4 in Section 3.1.2).

(6) Update the branch's chooser counter using the following rule:

	Results from predictors:		
	both incorrect or both correct	***gshare correct, bimodal incorrect***	***bimodal correct, gshare incorrect***
Chooser counter update policy:	no change	increment (but saturates at 3)	decrement (but saturates at 0)

4. Inputs to Simulator

4.1. Traces

Traces are posted on the Moodle site. 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 into 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
...
```

4.2. Command-line arguments to the simulator

The simulator executable built by your Makefile must be named "sim" (the Makefile is discussed in Section 6).

Your simulator must accept command-line arguments as follows: (Note: <tracefile> is the filename of the input trace.)

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.

To simulate a hybrid predictor: **`sim hybrid <K> <M1> <N> <M2> <tracefile>`**, where K is the number of PC bits used to index the chooser table, M1 and N are the number of PC bits and global branch history register bits used to index the gshare table (respectively), and M2 is the number of PC bits used to index the bimodal table.

5. Outputs from Simulator

The simulator outputs the following after completion of the run:

1. The simulator command (which indicates the branch predictor configuration and trace file).
2. The following measurements:
    a. number of predictions (*i.e.*, number of dynamic branches in the trace)
    b. number of branch mispredictions (predicted *taken* when *not-taken*, or predicted *not-taken* when *taken*)
    c. branch misprediction rate (# mispredictions/# predictions)
3. The final contents of the branch predictor.

See Section 6 regarding the formatting of these outputs and validating your simulator.

6. Submit, Validate, and Self-Grade with Gradescope

Sample simulation outputs are provided on the Moodle site. These are called "validation runs". **Refer to the validation runs to see how to format the outputs of your simulator.**

You must submit, validate, and self-grade[1] your project using Gradescope. Here is how Gradescope (1) receives your project (zip file), (2) compiles your simulator (Makefile), and (3) runs and checks your simulator (arguments, print-to-console requirement, and "diff -iw"):

1. **How Gradescope receives your project: zip file.** While you are developing your simulator, you may continuously submit new zip files to Gradescope containing the latest version of your project. The latest submission is the one that is considered for your grade. Gradescope will accept a zip file consisting of three things: your source code, a Makefile to compile your source code, and your project report. In the early stages of your project, before creating the report, your zip file will have only source code and a Makefile. Once the report is completed, your zip file will contain everything.
    - Report (included in the zip file once available): The report must be a PDF file named "report.pdf" located at the top level of the zip file, because that is what Gradescope looks for when checking completeness of the submission. 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 available on the Moodle site.
        - See Section 7 for the required content of the report.
    - Makefile: The Makefile must be at the top level of the zip file, because Gradescope runs "make" with the expectation that the Makefile is at the top level.
    - Source code: Whether your source code is at the top level of the zip file or in directories below the top level, your Makefile must be designed to compile your source code, accordingly.

2. **How Gradescope compiles your simulator: Makefile.** Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named "sim". An example Makefile is posted on the Moodle site, which you can copy and modify for your needs.

3. **How Gradescope runs and checks your simulator: arguments, print-to-console requirement, and "diff -iw".**
    - Your simulator executable (created by your Makefile) must be named "sim" and take command-line arguments in the manner specified in Section 4.2, because Gradescope assumes these things.
    - Your simulator must print outputs to the console (*i.e.*, to the screen), because Gradescope assumes this.

[1] The mystery runs component of your grade will not be published until we release it. The report will be manually graded by the TAs. Any late penalty will be manually assessed by the TAs.

- Your output must match the validation runs both numerically and in terms of formatting, because Gradescope runs “diff -iw” to compare your output with the correct output. The -iw flags tell “diff” to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace 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. Note, however, that extra or missing blank lines are NOT ok: “diff -iw” does not ignore extra or missing blank lines.

7. Tasks, Grading Breakdown

PART 1: BIMODAL PREDICTOR

(a) [**ECE463: 25 points**] or [**ECE563: 20 points]** Gradescope will evaluate your simulator on the four validation runs "val_bimodal_1.txt", "val_bimodal_2.txt", "val_bimodal_3.txt", and "val_bimodal_4.txt", posted on the website for the BIMODAL PREDICTOR. Gradescope will also evaluate your simulator on one bimodal predictor mystery run. Each validation run and mystery run is worth ⅕ of the points for this part (5 or 4 points each). Gradescope must say that you match all four validation runs to get credit for the experiments with the bimodal predictor, however.

(b) [**ECE463: 25 points**] or [**ECE563: 20 points**] Simulate BIMODAL PREDICTOR for different sizes ($7 \leq m \leq 20$). Use the traces ***gcc***, ***jpeg***, and ***perl*** in the trace directory.
[20 or 15 points] 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 14 datapoints (connect the datapoints with a line).
[5 points] Analysis: Draw conclusions and discuss trends. Discuss similarities/differences among benchmarks.

PART 2: GSHARE PREDICTOR

(a) [**ECE463: 25 points**] or [**ECE563: 20 points**] Gradescope will evaluate your simulator on the four validation runs "val_gshare_1.txt", "val_gshare_2.txt", "val_gshare_3.txt", and "val_gshare_4.txt", posted on the website for the GSHARE PREDICTOR. Gradescope will also evaluate your simulator on one gshare predictor mystery run. Each validation run and mystery run is worth ⅕ of the points for this part (5 or 4 points each). Gradescope must say that you match all four validation runs to get credit for the experiments with the gshare predictor, however.

(b) [**ECE463: 25 points**] or [**ECE563: 20 points**] Simulate GSHARE PREDICTOR for different sizes ($7 \leq m \leq 20$), and for each size, *i.e.*, for each value of *m*, sweep the global history length *n* from 0 to *m*. Use the traces ***gcc***, ***jpeg***, and ***perl*** in the trace directory.
[20 or 15 points] Graphs: Produce one graph for each benchmark. Graph title: "<*benchmark*>, gshare". Y-axis: branch misprediction rate. X-axis: *n* (spanning n=0 to n=20). Per graph, there should be a total of 203 datapoints plotted as 14 curves. Datapoints having the same value of *m* (same predictor size) are connected with a line, *i.e.*, one curve for each value of *m*. Note that not all curves have the same number of datapoints; see the listing below for the number of datapoints for each of the 14 curves, m=7 through m=20. The rationale for this graph is to study the effect of global history length for each predictor size.
[5 points] Analysis: Draw conclusions and discuss trends. Discuss similarities/differences among benchmarks.

m=7 curve has 8 datapoints: $0 \leq n \leq 7$

m=8 curve has 9 datapoints: $0 \leq n \leq 8$
m=9 curve has 10 datapoints: $0 \leq n \leq 9$
m=10 curve has 11 datapoints: $0 \leq n \leq 10$
m=11 curve has 12 datapoints: $0 \leq n \leq 11$
m=12 curve has 13 datapoints: $0 \leq n \leq 12$
m=13 curve has 14 datapoints: $0 \leq n \leq 13$
m=14 curve has 15 datapoints: $0 \leq n \leq 14$
m=15 curve has 16 datapoints: $0 \leq n \leq 15$
m=16 curve has 17 datapoints: $0 \leq n \leq 16$
m=17 curve has 18 datapoints: $0 \leq n \leq 17$
m=18 curve has 19 datapoints: $0 \leq n \leq 18$
m=19 curve has 20 datapoints: $0 \leq n \leq 19$
m=20 curve has 21 datapoints: $0 \leq n \leq 20$

PART 3: HYBRID PREDICTOR (ECE563 students only)

[ECE563: 20 points] Gradescope will evaluate your simulator on the two validation runs "val_hybrid_1.txt" and "val_hybrid_2.txt" posted on the website for the HYBRID PREDICTOR. Gradescope will also evaluate your simulator on two hybrid predictor mystery runs. Each validation run and mystery run is worth ¼ of the points for this part (5 points each).

8. Penalties

Various deductions (out of 100 points):

-1 point for each day (24-hour period) late, according to the Gradescope timestamp. The late penalty is pro-rated on an hourly basis: -1/24 point for each hour late. We will use the "ceiling" function of the lateness time to get to the next higher hour, *e.g.*, ceiling(10 min. late) = 1 hour late, ceiling(1 hr, 10 min. late) = 2 hours late, and so forth.

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 referral to the Office of Student Conduct.

9. Advice on backups and run time

9.1. Keeping backups

It is good practice to frequently make backups of all your project files, including source code, your report, *etc*. You can backup files to another hard drive (your AFS locker in your Eos account, home PC, laptop … keep consistent copies in multiple places) or removable media (flash drive, *etc*.).

9.2. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important because 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.

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 Moodle site 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.