CS520 Computer Architecture

Project 1 – Spring 2019

Due: 3/5, 11:59 pm

1. RULES

(1) All students must work alone. Cooperation is not allowed.
(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 through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
(3) You must do all your work in the C/C++.
(4) Your code must be compiled on remote.cs.binghamton.edu or the machines in the EB-G7 and EB-Q22. This is the platform where the TAs will compile and test your simulator. As I know, they all have the same software environment.

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

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 ≤ m. Note: n may equal zero, in which case we have the simple bimodal branch predictor.

3.1. 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 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.)

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 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.

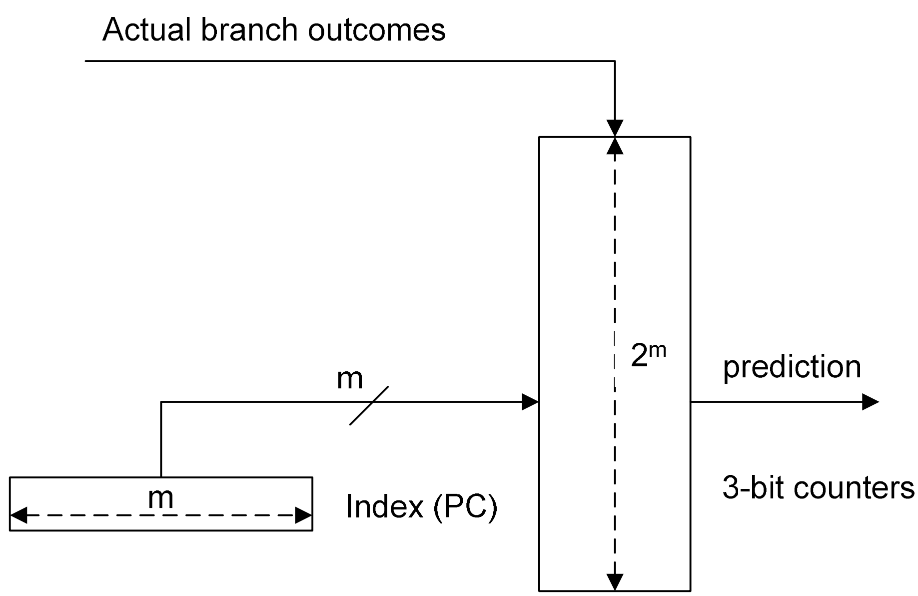

Figure 1. Bimodal branch predictor

3.2. 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.

When you get a branch from the trace file, there are four 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.

(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 1 bit position, and place the branch's actual outcome into the most-significant bit position of the register.

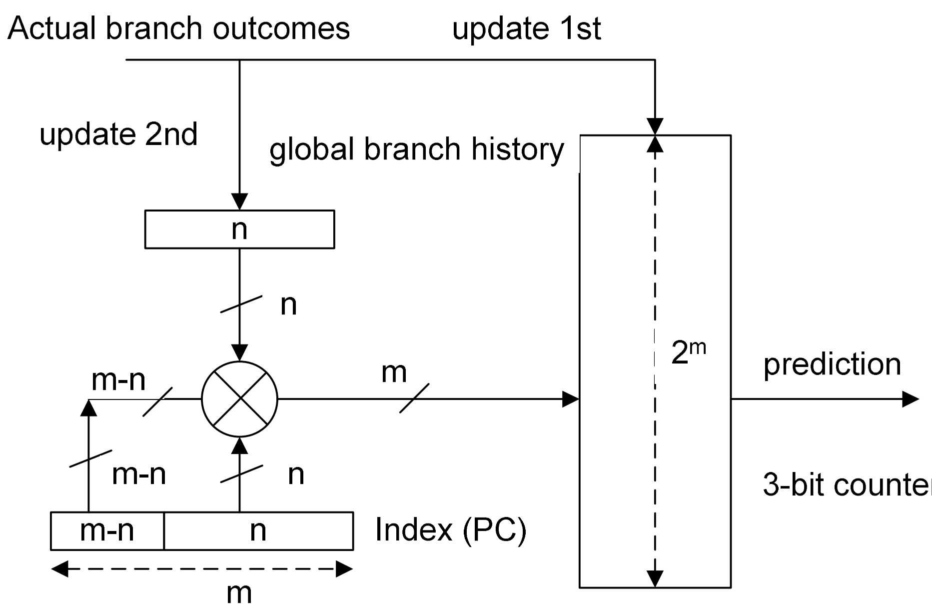

Figure 2. Gshare branch predictor

3.3. 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.

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 3.2) and one from the bimodal predictor (follow steps 1 and 2 in Section 3.1).

(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 3.2, otherwise follow step 3 in Section 3.1).

(5) Note that the gshare's global branch history register must always be updated, even if bimodal was selected (follow step 4 in Section 3.2).

(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

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
...

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 misprediction rate (# mispredictions / # branches)

6. Validation and Other Requirements

6.1. Validation requirements

Sample simulation outputs will be provided on the website. 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 this requirement.)

Your output must match both numerically and in terms of 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 –iw <your_output_file> <posted_output_file>”
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.

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 machines in EB-G7 and EB-Q22. This is required so that the TAs can compile and run your simulator. You also can access the machine with the same environment remotely at remote.cs.binghamton.edu via SSH.

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 will be posted on the web page, which you can copy and modify for your needs.

3. Your simulator must accept command-line arguments as follows:

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

To simulate a gshare predictor: sim gshare <M> <N> <tracefile>, where M 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.

(<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.

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 web page 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

Part1. Bimodal Predictor

(a) [20 points] Match 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. Note, we will use additional internal validation runs to verify your work. You must match all validation runs to get full credits for the experiments with the bimodal predictor.

(b) [20 points] Simulate bimodal predictor configurations for $7 \leq m \leq 12$. Use the provided traces.

Graphs [5 points]: Produce one graph for each benchmark. Graph title: "<benchmark>, bimodal". Y-axis: branch misprediction rate. X-axis: m. Per graph, there should be one curve consisting of 6 datapoints (connect the datapoints with a line).

Analysis [5 points]: Draw conclusions and discuss trends. Discuss similarities/differences among benchmarks.

Design [10 points]: For each trace, choose a bimodal predictor design that minimizes misprediction rate AND minimizes predictor cost in bits. You have a maximum budget of 16 kilobytes of storage.

When “minimizing” misprediction rate, look for diminishing returns, i.e., the point where misprediction rate starts leveling off and additional hardware provides only minor improvement. Bottom line: use good sense to provide high performance and reasonable cost. Justify your designs with supporting data and explanations.

Part 2. Gshare Predictor

(a) [20 points] Match 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. Note, we will use additional internal validation runs to verify your work. You must match all validation runs to get full credits for the experiments with the gshare predictor.

(b) [20 points] Simulate gshare predictor configurations for $7 \le m \le 12$, $2 \le n \le m$, n is even. Use the provided traces.

Graphs [5 points]: 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 of 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.

Analysis [5 points]: Draw conclusions and discuss trends. Discuss similarities/differences among benchmarks.

Design [10 points]: For each trace, choose a gshare predictor design that minimizes misprediction rate AND minimizes predictor cost in bits. You have a maximum budget of 16 kilobytes of storage. When “minimizing” misprediction rate, look for diminishing returns, i.e., the point where misprediction rate starts leveling off and additional hardware provides only minor improvement. Bottom line: use good sense to provide high performance and reasonable cost. Justify your designs with supporting data and explanations.

Part 3. Hybrid Predictor

(a) [10 points] Match the two validation runs “val_hybrid_1.txt” and “val_hybrid_2.txt” posted on the website for the HYBRID PREDICTOR. The TAs will also check that your simulator matches two mystery validation runs. Note, we will use additional internal validation runs to verify your work. You must match all validation runs to get full credits for the experiments with the hybrid predictor.

(b) [10 points] You have a maximum budget of 16 kilobytes of storage. Select your 4 best gshare predictors and 2 bimodal predictors from Part 1 and Part 2. Due to the memory budget, you may have to choose sub-optimal bimodal or gshare predictors. You must consider the chooser table size (1KB) too. From the combination of these predictors, you have 8 hybrid predictors (4 ghsare x 2 bimodal). Simulate these hybrid predictors with configurations for $4 \le k \le 12$. Use the provided traces.

Graphs [5 points]: Produce one graph for each benchmark. Graph title: “<benchmark>, hybrid”. Y-axis: branch misprediction rate. X-axis: k. Per graph, there should be 8 curves consisting of 9 datapoints (connect the datapoints with a line).

Analysis [5 points]: Draw conclusions and discuss trends. Discuss similarities/differences among benchmarks. Compare the results with gshare and bimodal only predictors.