Homework 3 - CIS501 Fall 2011

Instructor:	Prof. Milo Martin
Due:	Tuesday, October 25th (at the **start** of class)
Instructions:	This is an *individual* work assignment. Sharing of answers or code is strictly prohibited. For the short answer questions, **Show your work** to document how you came up with your answers.
Note:	This assignment has two parts (see below) due on the same day. Please print out your answers for each part and turn them in as **two distinct stapled documents**.
Extentions:	For this assignment, there are two important changes to the late policy. First, **no extensions on part A**; I want to give out the solutions to this part before the exam, so that isn't consistent with allowing extensions. Second, **if you taken an extension on part B, it is a two-class (one-week) extension**. Thus, if you do end up taking an extension on part B, it won't interfere with studying for the midterm exam.

Part A

1. **Performance and CPI.** Assume a typical program has the following instruction type breakdown:

    - 30% loads
    - 10% stores
    - 50% adds
    - 8% multiplies
    - 2% divides

    Assume the current-generation processor has the following instruction latencies:

    - loads: 4 cycles
    - stores: 4 cycles
    - adds: 2 cycles
    - multiplies: 16 cycles
    - divides: 50 cycles

    If for the next-generation design you could pick one type of instruction to make twice as fast (half the latency), which instruction type would you pick? Why?

2. **Averaging.** Assume that a program executes one branch ever 4 instructions for the first 1M instructions and a branch every 8 instructions for the next 2M instructions. What is the

average number instructions per branch for the entire program?

3. **Amdahl's Law.** A program has 10% divide instructions. All non-divide instructions take one cycle. All divide instructions take 20 cycles.

    a. What is the CPI of this program on this processor?

    b. What percent of time is spent just doing divides?

    c. What would the speedup be if we sped up divide by 2x?

    d. What would the speedup be if we sped up divide by 5x?

    e. What would the speedup be if we sped up divide by 20x?

    f. What would the speedup be if divide instructions were infinitely fast (zero cycles)?

4. **Performance and ISA.** Chip A executes the ARM ISA and has a 2.5Ghz clock frequency. Chip B executes the x86 and has a 3Ghz clock frequency. Assume that on average, programs execute 1.5 times as many ARM instructions than x86 instructions.

    a. For Program P1, Chip A has a CPI of 2 and Chip B has a CPI of 3. Which chip is faster for P1? What is the speedup for Program P1?

    b. For Program P2, Chip A has a CPI of 1 and Chip B has a CPI of 2. Which chip is faster for P2? What is the speedup for Program P2?

    c. Assuming that Programs P1 and P2 are equally important workloads for the target market of this chip, which chip is "faster"? Calculate the average speedup.

5. **ISA Modification and Performance Metrics.** You are the architect of a new line of processors, and you have the choice to add new instructions to the ISA if you wish. You consider adding a fused multiply/add instruction to the ISA (which is helpful in many signal processing and image processing applications). The advantage of this new instruction is that it can reduce the overall number of instructions in the program by converting some pairs of instructions into a single instruction. The disadvantage is that its implementation requires extending the clock period by 5% and increases the CPI by 10%.

    a. Calculate under what conditions would adding this instruction lead to an overall performance increase.

    b. What qualitative impact would this modification have on the overall MIPS (millions of instructions per second) of the processor?

    c. What does this say about using MIPS as a performance metric?

6. **Caching.** Consider the following progression of on-chip caches for three generations of chips (loosely modeled after the Alpha 21064, 21164, and 21264).

a. Assume the first generation chip had a direct-mapped 8KB single-cycle first-level data cache. Assume T_{hit} for this cache is one cycle, T_{miss} is 100 cycles (to accesses off-chip memory), and the miss rate is 20%. What is the average memory access latency?

b. The second generation chip used the same direct-mapped 8KB single-cycle first-level data cache, but added a 96KB second-level cache on the chip. Assume the second-level cache has a 10-cycle hit latency. If an access misses in the second-level cache, it takes an additional 100 cycles to fetch the block from the off-chip memory. The second-level cache has a global miss rate of 4% of memory operations (which corresponds to a local miss rate of 20%). What is the average memory access latency?

c. The third generation chip replaced the two-levels of on-chip caches with a single-level of cache: a set-associative 64KB cache. Assume this cache has a 5% miss rate and the same 100 cycle miss latency as above. Under what conditions does this new cache configuration have an average memory latency lower than the second generation configuration?

Part B

Overview

In this assignment you will explore the effectiveness of branch direction prediction (taken vs not taken) on an actual program. Your task is to use the trace format (see Trace Format document) to evaluate the effectiveness of a few simple branch prediction schemes.

To do this, you'll write a program that reads in the trace and simulates different branch prediction schemes and different predictor sizes. You can write the program in whatever language you like; although we are going to ask for a printout of your source code, the end result of this assignment are the experimental results (and not the program source code).

Processing the Trace

In this assignment we want to focus only on **conditional** branches. In the trace format, conditional branches:

1. Reads the flags register (that is, conditionRegister == 'R'), and
2. Is either taken or not taken (that is, TNnotBranch != '-').

Ignore all the lines in the trace that fail to meet both the above criteria.

You'll be predicting the taken/not-taken status of each branch. You'll use the program counter from the trace to make the prediction and the taken/not-taken field to determine if the prediction was correct or not.

To help you debug your predictors, we've provided some annotated prediction results for the first couple hundred branches for various predictors (see below). Comparing your results to these annotated outputs should help ensure your predictor is working properly.

Question 1 - Static Branch Predictor

Before looking at dynamic branch prediction, we are going to look at simple "static" branch prediction policies of "always predict taken" and "always predict not taken". Modify the tracer reader to calculate the mis-prediction rate (that is, the percentage of conditional branches that were mis-predicted) for these two simple schemes.

a. What is the prediction accuracy for "always taken"?

b. What is the prediction accuracy for "always not taken"?

Bimodal Predictor

The simplest dynamic branch direction predictor is an array of 2^n two-bit saturating counters. Each counter includes one of four values: *strongly taken* (T), *weakly taken* (t), *weakly not taken* (n), and *strongly not taken* (N).

Prediction. To make a prediction, the predictor selects a counter from the table using using the lower-order *n* bits of the instruction's address (its program counter value). The direction prediction is made based on the value of the counter.

> **Note**
>
> The easiest way to calculate the index is to modulo (%) the address by the number of entries in the predictor table. However, as the table size is always a power of two, it is almost as easy to use shifts and bit-wise and/or operations to calculate the index.
>
> In the past some students have converted the integer address to a string, extracted a sub-string, and then converted the string back into an integer. Don't do this; it is unnecessary, slow, and ugly.

Training. After *each* branch (correctly predicted or not), the hardware increments or decrements the corresponding counter to bias the counter toward the actual branch outcome (the outcome given in the trace file). As these are two bit *saturating* counters, decrementing a minimum counter or incrementing a maxed out counter should have no impact.

Initialization. Although initialization doesn't effect the results in any significant way, your code should initialize the predictor to "strongly not taken" so that your results match the example traces that we've given you.

Question 2 - Bimodal Accuracy versus Predictor Size

Analyze the impact of predictor size on prediction accuracy. Write a program to simulate the

bimodal predictor. Use your program to simulate varying sizes of bimodal predictor. Generate data for bimodal predictors with 2^2, 2^3, 2^4, 2^5 ... 2^{20} counters. These sizes correspond to predictor index size of 2 bits, 3 bits, 4 bits, 5 bits, ... 20 bits. Generate a line plot of the data using MS Excel or some other graphing program. On the y-axis, plot "percentage of branches mis-predicted" (a metric in which smaller is better). On the x-axis plot the log of the predictor size (basically, the number of index bits). By plotting the predictor size in terms of number of index bits, the x-axis in essence becomes a log scale, which is what we want for this graph.

Answer the following questions base on the data you collected:

a. Given a large enough predictor, what is the best mis-prediction rate obtainable by the bimodal predictor?

b. How large must the predictor be to reduce the number of mis-predictions by approximately half as compared to the better of "always taken" and "always not taken"? Give the predictor size both in terms of number of counters as well as bytes.

c. At what point does the performance of the predictor pretty much max out? That is, how large does the predictor need to be before it basically captures almost all of the benefit of a much larger predictor.

Prediction using Global Branch History (gshare)

A gshare predictor is a more advanced dynamic branch predictor that uses the history of recently executed branches to predict the next branch. Gshare uses a history register to record the taken/not-taken history of the last *h* branches. It does this by shifting in the most recent conditional branch outcome into the low-order bits of the branch history register. It then hashes the branch history and the PC of the branch when making predictions.

Prediction. Whereas bimodal uses the lowest *n* bits of the program counter to index into the table, gshare uses the lowest *n* of the **xor** of the program counter and the history register. (Note: in C/C++/Java, the "^" operator is the xor operator). Except for this different index into the table, the prediction mechanism is otherwise unchanged from a bimodal predictor. In essence a gshare predictor with zero history bits is basically just a bimodal predictor.

> **Note**
>
> The easiest way to encode the history is to use an integer and then use bit manipulations operators such as shifts, bitwise-and (&), bitwise-or (|), xors (^), etc.
>
> Adjusting the history register can be done in three steps: shifting the bits left by one, clearing the nth bit (to model a bounded-size history), and conditionally setting the lowest bit to zero or one, depending on the taken/non-taken status of the branch.
>
> All in all, this can be done in just a few lines of C or Java code. The following two wikipedia pages might give you some hints:

> http://en.wikipedia.org/wiki/Bit_manipulation
>
> http://en.wikipedia.org/wiki/Bitwise_operation

Training. The counter used for prediction (selected by history XOR program counter) is trained just as in a bimodal predictor.

Initialization. Same as bimodal.

Question 3 - Gshare Accuracy versus Predictor Size

Similar to the experiment in question 2, analyze the impact of predictor size on prediction accuracy for a gshare predictor. Extend your program for simulating bimodal predictors to also simulate gshare predictors. Recall that a gshare predictor's configuration is determined by two numbers: the number of history bits and the size of the predictor table.

Use your program to simulate gshare predictor in 8 history bits of varying sizes (the same sizes as in the previous question). Add the gshare data to the graph you created in the previous question. That is, your new graph will have two lines.

Answer the following questions base on the data you collected:

a. Given a large enough predictor, what is the best mis-prediction rate obtainable by this gshare predictor with 8 history bits?

b. At what table size is the gshare predictor generally better than the simple bimodal predictor?

c. Explain why gshare is sometimes more accurate than bimodal.

d. Explain why bimodal is sometimes more accurate than gshare.

Question 4 - Gshare History Length vs Prediction Accuracy

The previous question used a gshare predictor with 8 bits of history. But how much history should a predictor use? Use your simulator to simulate gshare predictors with varying history lengths (zero bits of history through 19 bits of history). Simulate two different predictor sizes: 2^{10} and 2^{16} predictor entries. Plot one line for each size. The y-axis is as before. The x-axis is now history length.

a. What is the optimal history length (the history length with the lowest mis-prediction rate) for each of these two predictor sizes?

b. What do these results say about picking a history length? Generally speaking, what history lengths are the best performing?

Question 5 - Gshare Accuracy versus Predictor Size Revisited

Generate a new series of data in which the gshare history length is the same as the number of index bits. Plot a line of this gshare data on the graph from question #3. This graph should have three lines: the bimodal data, the gshare with a fixed 8-bit history, and the data for gshare where the history length grows with predictor size.

a. Looking at this new data, what is new the lowest (best) mis-prediction rate for gshare?

b. When comparing the new gshare data to bimodal, where is the crossover point between then? (That is, what is the predictor size at which gshare becomes better than bimodal?) How does this crossover point compare to the crossover point of gshare with 8-bit history versus bimodal.

c. In these experiments, we're looking at a trace of just one program. Of course, processors run many different programs. Each of these programs might have a different crossover point. What properties of a program might most significantly impact where the crossover point between bimodal and gshare occurs?

Tournament Predictor

From the previous data, you can see that neither bimodal or gshare is always best. To remedy this situation, we can use a hybrid predictor that tries to capture the best of both style of predicts. A tournament predictor consists of *three* tables. The first and second tables are just normal bimodal and gshare predictors. The third table is a "chooser" table that predicts whether the bimodal or gshare predictor will be more accurate.

The basic idea is that some branches do better with a bimodal predictor and some do better with a gshare predictor. So, the chooser is a table of two-bit saturating counters indexed by the low-order bits of the PC that determines *which* of the other two table's prediction to return. For the each entry in the chooser, the two-bit counter encodes: *strongly prefer bimodal* (B), *weakly prefer bimodal* (b), *weakly prefer gshare* (g), and *strongly prefer gshare* (G).

Prediction. Access the chooser table using the low-order bits of the branch's program counter address. In parallel, the bimodal and gshare tables are accessed just as normal predictors, and each generates an independent prediction. Based on the result of the lookup in the chooser table, the final prediction is either the prediction from the bimodal predictor (if the choose counter indicates a preference for bimodal) or the prediction from the gshare predictor (otherwise).

Training. Both the gshare and bimodal predictors are trained on *every* branch using their normal training algorithm. The choose predictor is trained toward which of the two predictors was more accurate on that specific prediction:

- Case #1: The two predictors make the *same* prediction. In this case, either both predictors were correct or both predictors were wrong. In both cases, the chooser table isn't updated (as we didn't really learn anything).
- Case #2: The two predictors made *different* predictions (thus, one of the predictors was correct and the other incorrect). In this case, the chooser counter is trained (incremented or decremented by one) toward preferring that of the predictor that was correct. For example, if the gshare predictor was correct (and thus the bimodal predictor was incorrect), the chooser

counter is adjusted by one toward preferring gshare.

As stated above, the bimodal and gshare tables are trained on *every* branch, totally independent of the chooser table.

Initialization. The chooser table is initialized to *strongly prefer bimodal*. The gshare and bimodal tables are initialized as normal.

Question 6 - Tournament Predictor Accuracy

Add support to your simulator for the tournament predictor by adding support for the chooser table. If your code was written in a modular way, you should be able to fully re-use your gshare and bimodal code (and thus avoid re-writing or replicating code).

Let's compare the tournament predictor's accuracy versus the data from its two constituent predictors (using the data from the previous question). For now, let's compare a 2^n-counter bimodal (or gshare) versus a tournament predictor with *three* 2^n-counter tables. (We'll make the comparison more fair in terms of a "same number of bits" comparison in a moment.) As above for the gshare predictor, the gshare component of the tournament predictor should use a history length equal to the number of index bits (log of the table size). The graph will have three lines total.

a. How does the tournament predictor's accuracy compare to bimodal and gshare? Is the tournament predictor successful in meeting its goal?

b. Does the tournament predictor improve the overall peak (best-case) accuracy? If so, why? If not, what are the benefits of the tournament predictor?

Question 7 - Tournament Predictor Accuracy -- Same Size Comparison

In the previous question, the tournament predictor was given the unfair advantage of having three times the storage. In this question, run another set of experimental data in which all the predictors at the same location on the x-axis have the same storage capacity. To do this, compare a 2^n-counter predictor to a tournament predictor with the following three table sizes:

- Chooser table: 2^{n-2} counters
- Bimodal table: 2^{n-2} counters
- Gshare table: 2^{n-1} counters

As $2^{n-2} + 2^{n-2} + 2^{n-1}$ is equal to 2^n, this becomes a fair "same size" comparison.

Add a line to the graph from the previous question with this new "fair tournament" data. The graph now has four lines: bimodal, gshare, tournament, tournament-fair.

a. Compare the old and new tournament predictor data. What was the impact of moving to the smaller (fairer) table sizes?

b. Once adjusted to be a fair-size comparison, does the tournament predictor succeed in its goal

of being the best of bimodal and gshare?

c. Given a fixed transistor budget for a branch predictor (say, 4KBs), what predictor approach would you use?

> **Note**
>
> The idea of gshare and tournament (or "combining") branch predictors was proposed by Scott McFarling in his seminal paper published in 1993 entitled: Combining Branch Predictors. I encourage you to look at McFarling's paper. After having completed this assignment, I think you'll find the graphs and other results in this paper familiar.

Test Cases

We're provide you with three annotated results (one for each predictor type) from running the first few hundred branches from the trace.

- bimodal3.output: bimodal with 2^3 counters:

  ```
  nNNNTNNN | b7fa3ae4  T | T  correct    3
  ```

  The columns are: table counter state, PC from input trace, branch outcome from input trace, prediction made, prediction result (correct/incorrect), running total of mis-predictions thus far.

- gshare4-3.output: gshare with 2^4 counters and history length of three:

  ```
  NNnNNntNnNNNNNNNN  NTN | b7fa3ae4  T | T  correct    5
  ```

  The columns are: table counter state, history register, PC from input trace, branch outcome from input trace, prediction made, prediction result (correct/incorrect), running total of mis-predictions thus far.

- tournament3-bimodal3-gshare4-4.output: tournament predictor with a chooser table with 2^3 counters, a bimodal with 2^3 counters, and a gshare with 2^4 counters with a history length of four:

  ```
  BBBBBBBB nNNNTNNN NNNNNnNNnNNNNNtN  TNTN | b7fa3ae4  T | T  correct    3
  ```

  The columns are: chooser predictor table, bimodal predictor, gshare predictor table, gshare history register, PC from input trace, branch outcome from input trace, prediction made, prediction result (correct/incorrect), running total of mis-predictions thus far.

What to Turn In

- **Short answers**. Turn in your answers to the above questions. I strongly urge you to type up your answers.
- **Graphs**. Turn in printouts of the following three graphs. One per sheet of paper (to make them big enough to read). Label axes and give the lines descriptive names in a legend.
  - **Graph A** [Questions #2, #3, and #5]: Include just one graph with all three lines: bimodal, gshare-8, and gshare-*n*.
  - **Graph B** [Question #4]: Include the graph describe in question four.
  - **Graph C** [Questions #6 and #7] This graph has four lines: bimodal, gshare-*n*, tournament, and tournament-fair.
- **Source Code**. Print out your final source code that can simulate the various predictors.

Hints & Comments

- **Runtime**. Running through the trace should take just a few seconds if implemented efficiently. Even if you run through the entire trace to generate each data point, it shouldn't take more than a few minutes to generate all the data you need for this assignment. If your code is taking much longer, you're probably not doing something right (ask me or the TAs about it). The most common culprit for greatly increased runtime in the past have been: (1) using string objects rather than using bit manipulation operations (shift, bit-wise and, bit-wise or, etc.) for extracting the index and tag bits or (2) allocating heap objects as part of processing each line in the trace.
- **Computer resources**. If you wish to write the code by ssh'ing into a Linux box, you can log into minus.seas.upenn.edu. These machines are multi-core multi-Ghz Intel chips, so they are quite fast. However, if your jobs run for too long (say 10 or 20 minutes) the system will kill off the job (to make sure that users don't use these machines for very long running jobs). Of course, you're also welcome to write and run the code on your personal computers or the lab machines.
- **Code reuse**. In the end, the total amount of code needed to perform these simulations isn't actually that much. Don't go crazy in making the code super modular, but conversely there is significant potential for code reuse that shouldn't be ignored.
- **Automation of data collection**. Put a little bit of though into how you're going to collect the various data for each question. If you're running your program multiple times by hand and then copying and pasting the individual results into Excel, you're doing something wrong (or at least super inefficiently). Your code should be able to perform a range of simulations in an automated fashion (either by building it into the main program of using scripts to call it with different parameters). You program should be able to split out data text files that can be imported into Excel (or whatever) to make graphs with minimal tedium.

Addendum

- None, yet.
