CSc 656 Project 1 Part b

(This document available on iLearn.)
due Thursday 4/5/2018 3:00 pm (no grace period)
(5% of your grade)
updated 3/28/2018

This is an individual assignment. Work on your own!

All code/data you'll need for this project are on unixlab.sfsu.edu.

Code: `~whsu/csc656/Code/S18/P1`
Data: `~whsu/csc656/Traces/S18/P1`

Problem 4:

You will be given trace files on unixlab.sfsu.edu in

~whsu/csc656/Traces/S18/P1

(These are in the unixlab file system. You can log on to unixlab.sfsu.edu, and access them through Unix file system commands. The directory is also linked to a web-accessible area: http://unixlab.sfsu.edu/~whsu/csc656/Traces/S18/P1)

Write two branch prediction simulators to work with these trace files. You may work in C/C++ or Java; other languages will require prior approval.

We will use traces for SPEC 2006 benchmarks, from University of Pennsylvania:

https://www.cis.upenn.edu/~milom/cis501-Fall12/traces/trace-format.html

Each line of a trace file contains 14 fields, representing information from instructions extracted from running a program. We'll focus on conditional branches. The relevant fields for this project are:

Field 1 PC: the address of the branch instruction
Field 7 Branch: this is a single character
'T' means taken, 'N' means not taken, '-' means not a branch
Field 12 Target PC: the branch target address

Note that these are Intel x86 traces, not MIPS traces. Hence, instruction addresses are ***not*** word-aligned! This means the least significant two bits of the PC or Target PC are not always zero. When computing an index for the branch prediction buffer or branch target buffer, do ***not*** remove the least significant two bits.

For this project, your simulator will only work with conditional branches. **Ignore all other instructions.**

You will develop two branch prediction simulators, System 1 and System 2. **Your code must compile and run on unixlab.sfsu.edu.** (Note that unixlab.sfsu.edu is running fairly old versions of C/C++ and Java.) Sample executables for these two systems, named sys1 and sys2, will be found on unixlab.sfsu.edu in ~whsu/csc656/Code/S18/P1. Your programs will simulate the behavior of these branch predictors:

System 1 (static prediction):

Forward branches are predicted not taken, backward branches predicted taken. This is not a hardware branch predictor; you can get the misprediction statistics from scanning the list of conditional branches in each trace file. So if a forward branch is not taken, it is predicted correctly, otherwise it is mispredicted. Similarly, if a backward branch is taken, it is predicted correctly, otherwise it is mispredicted.

System 1 will take Unix command line arguments. The first argument is the name of the trace file. The second argument (optional) is "-v", which turns on verbose mode. See **Submissions** section for format.

System 1 must work correctly for System 2 code to be graded. If your System 1 code does not work, don't even bother working on System 2.

System 2 (basic 2-bit predictor with branch target buffer):

This is the N-entry 2-bit predictor/M-entry branch target buffer from Chapter 4a slides, in the section Branch Handling 2: dynamic branch prediction. (As mentioned earlier, do ***not*** remove the least significant two bits of the PC when you form an index for a buffer.) Your code should work with any N and M, both powers of 2. Follow the definitions from the slides on how to index the array of 2-bit branch predictors, and how to update the states of each 2-bit predictor. Assume all predictors start in state 01. Remember that predictors are not tagged; different conditional branches can update the same predictor without reinitializing to state 01. The branch target buffer entries are tagged, of course.

System 2 operation in more detail:

```
For each branch Bi
        Get prediction for Bi (predict taken or predict not taken)
        If Bi is predicted taken
                Check BTB entry for Bi
                If valid bit of BTB entry == 1 && tag of BTB entry == tag of Bi
                        We have a BTB hit
                Else
                        We have a BTB miss
        Else if Bi is predicted not taken
                Do nothing
```

Check whether B_i is *actually* taken/not taken
If prediction == actual behavior, *prediction is correct*
Else *prediction is wrong*

update prediction for B_i
If B_i actually taken
    Tag of BTB entry = tag of B_i
    Valid bit of BTB entry = 1

System 2 will take Unix command line arguments. The first argument is the name of the trace file. The second argument is N (number of entries in predictor buffer, always a power of two), the third argument is M (number of entries in branch target buffer, always a power of two). The fourth argument (optional) is "-v", which turns on verbose mode. See **Submissions** section for format.

Simulator output

Sys1 should print
- the total number of conditional branches
- the number of forward branches
- the number of backward branches
- the number of forward taken branches
- the number of backward taken branches
- the number of mispredicted branches
- the misprediction rate for all branches (# mispredictions / # branches)

Sys2 should print all of the above, plus:

- the number of BTB misses
- the BTB miss rate (# BTB misses / # BTB accesses)

Your sys2 code should implement a verbose mode for debugging. Verbose mode is turned off by default, and turned on by the –v flag (see Submissions section for format). When verbose mode is on, in addition to the output generated in non-verbose mode, your simulator prints out, for each branch in the trace file, a list of integers on a single line. These are:

Order of branch in trace file (the first branch in the file is numbered 0)
Index of prediction buffer accessed (in hexadecimal)
Current state of prediction buffer
New state of prediction buffer after update
Index of BTB accessed (in hexadecimal)
Tag of BTB entry, calculated from current branch address (in hexadecimal)
Number of BTB accesses so far
Number of BTB misses so far

For example, the test trace *test.trace* contains:

```
1 48d1de -1 -1 13 - - -         0            0       48d1e2           0 SET      ADD
2 48d1de -1 -1 13 R - -         1            0       48d1e2           0 SET      ADD_IMM
1 48d1e2 -1  5 45 - - L      -264  7fffe7ff048       48d1e9           0 CMP      LOAD
2 48d1e2 45  3 44 W - -         0            0       48d1e9           0 CMP      SUB
1 48d1e9 -1  5  3 - - L      -200  7fffe7ff088       48d1f0           0 MOV      LOAD
1 48d1f0 -1 -1  0 - - -         0            0       48d1f3           0 SET      ADD
2 48d1f0 -1 -1  0 R - -         1            0       48d1f3           0 SET      ADD_IMM
1 48d1f3 -1 -1 12 - - -         0            0       48d1f6           0 XOR      ADD
1 48d1f6 13  0 13 W - -         0            0       48d1f9           0 OR       OR
1 48d1f9 -1 -1 -1 - T -        54            0       48d1fb       48d231 JMP     JMP_IMM
1 48d231 -1  3  0 - - L         0  7fffe7fefd0       48d234           0 MOV      LOAD
1 48d234  0  0 44 W - -         0            0       48d237           0 TEST     AND
1 48d237 -1 -1 -1 R N -       -25            0       48d239       48d220 J       JMP_IMM
1 48d239 -1  0  7 - - L         8      12ff228       48d23d           0 MOV      LOAD
1 48d23d -1 -1 -1 - T -        54            0       48d23f       48d1f3 JMP     JMP_IMM
1 48d1f3 -1 -1 12 - - -         0            0       48d1f6           0 XOR      ADD
1 48d1f6 13  0 13 W - -         0            0       48d1f9           0 OR       OR
1 48d1f9 -1 -1 -1 - T -        54            0       48d1fb       48d231 JMP     JMP_IMM
1 48d231 -1  3  0 - - L         0  7fffe7fefd0       48d234           0 MOV      LOAD
1 48d234  0  0 44 W - -         0            0       48d237           0 TEST     AND
1 48d237 -1 -1 -1 R T -       -25            0       48d239       48d220 J       JMP_IMM
1 48d220 -1 -1 13 R - -         1            0       48d224           0 SET      ADD_IMM
1 48d224 -1 -1 -1 - T -        54            0       48d226       48d1f3 JMP     JMP_IMM
1 48d1f3 -1 -1 12 - - -         0            0       48d1f6           0 XOR      ADD
1 48d1f6 13  0 13 W - -         0            0       48d1f9           0 OR       OR
1 48d1f9 -1 -1 -1 - N -        54            0       48d1fb       48d231 JMP     JMP_IMM
```

There are 7 branches in this file:

PC of branch	Target PC	Taken/not taken
48d1f9	48d231	1
48d237	48d220	0
48d23d	48d1f3	1
48d1f9	48d231	1
48d237	48d220	1
48d224	48d1f3	1
48d1f9	48d231	0

When we run sys1, we get

```
% ./sys1 test.trace
Number of branches = 7
Number of forward branches = 3
Number of forward taken branches = 2
Number of backward branches = 4
Number of backward taken branches = 3
Number of mispredictions = 3 0.428571
```

When we run sys2 with 8 branch prediction buffers + 4 branch target buffers, we get

```
% ./sys2 test.trace 8 4 -v
```

```
0 1 1 2 1 12347e 0 0
1 7 1 0 3 12348d 0 0
2 5 1 2 1 12348f 0 0
3 1 2 3 1 12347e 1 1
4 7 0 1 3 12348d 1 1
5 4 1 2 0 123489 1 1
6 1 3 2 1 12347e 2 1
Number of branches = 7
Number of forward branches = 3
Number of forward taken branches = 2
Number of backward branches = 4
Number of backward taken branches = 3
Number of mispredictions = 5 0.714286
Number of BTB misses = 1 0.500000
```

Traces and Reference Executables

Two trace files are provided on unixlab.sfsu.edu; each consists of 1,000,000 lines of instructions, in the UPenn trace format:

```
~whsu/csc656/Traces/S18/P1/gcc.xac
~whsu/csc656/Traces/S18/P1/sjeng.xac
```

(gcc.xac is excerpted from the gcc benchmark, sjeng.xac from the sjeng benchmark, both from SPEC 2006.)

Reference executables can be found on unixlab.sfsu.edu at

```
~whsu/csc656/Code/S18/P1/sys1
~whsu/csc656/Code/S18/P1/sys2
```

Note that these are *executables*; you have to log on to unixlab.sfsu.edu, navigate to the correct directory using Unix command line commands, and run them. (No fancy web client, sorry! *Real* systems engineers work on the command line.) See the *Submissions* section below for command line format for running the reference executables.

Measurements and results

You should make measurements for the two trace files, then fill out this table for each of the traces:

	# mispredictions	misprediction rate	# BTB misses	BTB miss rate
System 1			N/A	N/A
System 2				

(N=128,M=32)

System 2
(N=512,M=128)

mispredictions is the number of times the prediction from the prediction buffer does not match the behavior of the branch. Misprediction rate is # mispredictions / # branches.

A BTB miss occurs when a branch is predicted taken, but its target address is not in the target buffer. Branches that are not taken do not access the BTB. Hence, BTB miss rate is # BTB misses / # branches that are predicted taken.

Submit the result table with your code.

Submission (source code + results table):

Submit a tar/zip file using the iLearn submission link. Your tar/zip file should expand into a single directory tagged with your name. The directory should contain your source files and the results table; each source file should have a header that with accurate instructions on compiling and running your code on the Unix command line. If your instructions don't work perfectly, you may get a zero on the project.

Your instructions should allow us to generate two executables, sys1 and sys2 (for System 1 and 2 respectively). We will then run each one with this command line (assuming we're on unixlab.sfsu.edu) for C/C++:

```
./sys1 ~whsu/csc656/Traces/S18/P1/gcc.xac [-v]
./sys2 ~whsu/csc656/Traces/S18/P1/sjeng.xac 512 128 [-v]
etc etc
```

Or for Java:

```
java sys1 ~whsu/csc656/Traces/S17/P1/gcc.xac [-v]
java sys2 ~whsu/csc656/Traces/S17/P1/sjeng.xac 512 128 [-v]
etc etc
```