Lab #4: Branch Prediction Experiments

due Friday, December 3

Important note: These are not group labs. Everyone does this lab SEPARATELY.

For this lab assignment, you will write a configurable branch prediction simulator (in C, C++, or Java). Your simulator will read a branch trace (a chronological list of branch instruction program counter addresses), simulate the performance of the processor, generate branch prediction accuracy data, and calculate the execution time for the executing program. The branch trace has been generated by a simulator executing a real program. Your simulator is not the end product of this lab, but a tool you will use to complete it. In this lab, you will experiment with various branch predictor configurations and make conclusions about the best branch predictor for this set of programs.

Using the branch address trace:
An address trace is simply a list of addresses produced by a program running on a processor. These are the addresses of conditional branch instructions in the code as it is executed. These traces were generated by a simulator of a RISC processor running three programs: gcc, crafty, and mesa, from the SPEC benchmarks. The files are gcctraceout.gz, craftytraceout.gz and mesatraceout.gz, and are all compressed with gzip. They are large, so you don’t want to copy them. Instead, use the following commands to generate the trace and pipe it through your branch prediction simulator, like so (for java programs it will look slightly different):

```
gunzip -c ~cs141f/traces/gcctraceout | branchsim [branchsim args]
```

where branchsim is your branch simulator. Because your workload is three programs, you will run three simulations for each architecture you simulate, and then combine the results in some meaningful way. The simulator arguments should be look like this, so we can run it:

```
branchsim --bpalg [alg] --size [predsize] --pen [pen]
```

where [alg] is one of 1bit, 2bit, 2lev, cor, or gshare. [predsize] is the size (in bits) of the predictor, [pen] is the penalty in cycles for each mispredict. You will only use a couple of sizes for these experiments, but your simulator will work for any reasonable size (power of 2). If a power of 2 is not specified, just round up to the next power of 2. For the 2-level predictor, however, you need two sizes, which will be specified as --patsize [patsize] --patbhtsize [patbhtsize], and each will be the number of entries, not bits.

Format of the branch trace:
All lines of the branch trace are of the format:

```
PC T/N ICount
```

where PC is a 32-bit hexadecimal program counter value (the address of the branch), T/N is a single character indicating the branch was taken (T) or not taken (N). ICount is the count of instructions (including this branch) since the prior branch instruction. The instruction count information will be used to calculate execution time (or at least cycle count). A sample address trace starts out like this:

```
200a2688 T 2
200c2e30 N 33
200a271c T 30
200a1fa0 N 19
200a1fb0 N 4
200a2770 T 18
200a1fa0 N 19
200a1fb0 N 4
200a3158 N 97
200a7d3c T 51
```
The simulator output:
Your program should produce mispredict rates for the given input, as well as execution time (in cycles), and CPI. It should also print out, for information sake, total instruction count and percent of branches taken. Each trace contains exactly 10 million branches. Your simulations should process all of them.

Assumptions:
The branch prediction algorithms you will simulate are described in a separate handout. You will make the following assumptions in your simulations:
1. Every instruction takes 1 cycle to execute, including branches.
2. Correctly predicted branches incur no additional overhead (besides the one cycle to execute the branch). Mispredicted branches incur an additional 12 cycle mispredict penalty. These two assumptions taken together suggest a deep scalar in-order pipeline with 12 stages between the fetch and execute stages.
3. Assume all data structures are initialized to zero. This includes branch history tables, pattern tables, and the GHR. This won’t make any significant difference to performance, but allows us to know exactly what results you’ll produce.
4. Assume the predictors all use saturating counters, not the 2-bit state machine in the H&P book.

Analysis:
You will be using your simulator to evaluate several branch predictor configurations that use the same (or nearly the same) amount of space (in bits).
A. Compare all predictors assume 2Kbits total. This would be a 2K table of 1-bit predictors, or a 1K table of 2-bit predictors. For the 2-level predictor, there are several potential architectures – assume a 256-entry pattern table and a 128-entry branch history table. Assume all cycle times are the same. Give mispredict rate for all predictors, as well as CPI, and indicate the predictor that works best for each benchmark.
B. Repeat A, but with the assumption that the branch predictor is a critical timing path in the processor. Therefore, the gshare predictor requires a 1% higher cycle time, and the 2-level predictor a 5% higher cycle time.
C. Repeat the experiments with a total predictor size of 8Kbits. For the 2-level predictor, assume a 512-entry pattern table, and a 1024-entry branch history table. Assume equal cycle times.
D. Repeat the experiments in C, but this time assume we account for the differences in complexity differently. Assume that we cannot do the gshare and 2-level lookups in 1 cycle, so we have to add an extra cycle to the pipeline. This means that the mispredict penalty for those schemes would increase by 1 cycle.
E. In real systems we typically cannot choose a different predictor for each application. Choose some reasonable mechanism for combining results, and tell me the single predictor you would thus choose for experiment C.
F. For a small amount of extra credit, repeat the 2-level experiments in C, but this time find your own 2-level predictor, which uses 8Kbits or less, and gives the best overall performance.

Questions for lab 4:
1. In my experiments in preparing this lab, I ran a larger set of experiments (ie, more benchmarks), in which all but one of the predictors was best for at least one benchmark. Explain to me the characteristics of a program for which the 1-bit predictor would be best. Describe a program in which you would expect the 2-level predictor to be best.
2. Turn in your branch predictor simulator code with sample output.
3. In these simple experiments, performance is a linear function of mispredict rate. Describe a situation in which this is not true – in fact, you could imagine a situation where a predictor with a (slightly!) higher mispredict rate would provide higher performance (lower CPI – ignore cycle time for this question). Think about a dynamically scheduled (out-of-order) processor.
4. Which of the predictors seems to be hampered most by the small size? Which do you think would benefit most from having a near-infinite capacity?
5. Were results uniform across the three programs? In what cases did different programs give different conclusions? Speculate as to what characteristics of the programs may have been different.
6. What was the average speedup of your best design over the worst? Use experiment B in this case.
7. Your report will be turned in on paper, but we will also have electronic turnin for verification.

Hints:
- Think about how to intelligently debug and test your program. Running immediately on the entire input gives you little insight on whether it is working (unless it is way off).
- Speed matters. These simulations should take a couple minutes (or less) on an unloaded uAPE machine. If it is taking much more than that, do yourself a favor and think about what you are doing inefficiently.
- Give execution time in some reasonable and consistent form (only an issue when the clock rate varies). One way to do that is to assume particular cycle times and give the results in seconds. If you do that, assume the baseline clock rate is 1 GHz.
- Big hint for those using C: scanf("%lx %c %d\n",&pc,&takenchar,&icount);
- UNIX machines are available in AP&M B-402 and EBU2 313.
- A word to the wise: these labs will be turned in electronically – duplicates (even with lots of superficial changes) can be detected, even with prior classes.