Lab 1: Bootstrapping

16 September, 2024

Overview

In this lab you will implement four branch outcome prediction algorithms that we learned in class and compare their accuracy and implementation complexity. The branch predictors that you will implement are:

- Static predictor (e.g., always-taken or always-not-taken)
- Bimodal / saturating counter predictor
- Two-level (e.g., GAp or PAg) predictor
- GShare predictor

The purpose of this lab is to understand the implementation of these predictors and to observe and measure the cost and benefit of more sophisticated predictors.

This lab requires you to implement the predictors in a Pintool-based simulation, to simulate real programs like you did in lab 0. The Pintool will send a trace of its branch instructions to your branch predictor simulator. For each branch instruction, your predictor will run its prediction algorithm, generate a prediction, and update its predictor state. As the simulation runs, the simulator should keep track of how many branches your branch predictor correctly predicts and how many total branches execute, allowing you to measure your branch prediction accuracy (some of this functionality exists already in the base code).

You will implement `predict()`, `update()`, and any other necessary functionality for four branch predictors (we recommend implementing these in the existing `bp.cpp` file). The `predict()` function will take a single argument, containing the PC value for the branch instruction. The `update()` function will take two arguments, the branch PC value, and a flag indicating whether the branch was taken or not. Where applicable, you will use these arguments to update the state of the predictor.

PC will be provided with the datatype `unsigned long`, and the `taken` argument of type `bool` informs whether the branch was taken or not.

Branch Predictor Details

The predictors that you must implement are a static predictor, a local bimodal (saturating counter) predictor, a two-level predictor (either or both of the GAp or PAg predictors from class, or any other scheme that is more effective), and a gshare predictor.

Static predictor

A static predictor always predicts either taken or not taken.

Local bimodal predictor

A local bimodal predictor conceptually maintains one saturating counter predictor per branch instruction address. Practically, such a predictor is not possible becuase there cannot be a table with one entry per branch in the program because the size of the table would be very large or effectively unbounded. You should size the table to be reasonably large (4k entries is probably the high end of reasonable).

Two-level predictor

There are many options for implementing a two-level predictor and we discussed some of them in class. One is to implement a BHT indexed by address that maintains global history and indirects into a pattern history table full of bimodal predictors. Feel free to get creative with the structure of your two-level predictor. GAp and PAg predictors work very well. BHTs, GHTs, and PHTs are both limited to reasonable sizes and you should justify the size of each structure in your write-up.

GShare predictor

A GShare predictor keeps a single global history register that gets hashed with the PC to index into a pattern history table of bimodal predictors. The PHT is limited to reasonable sizes and you should justify the size of each structure in your write-up.

Lab Logistics

Benchmarks to evaluate

You will evaluate the accuracy of your predictors using the SPEC2017 benchmark suite. As with Lab 0, you will use the functional subset of benchmarks: `gcc_s`, `mcf_s`, `omnetpp_s`, `xalancbmk_s`, `x264_s`, `leela_s`, `exchange2_s`, `xz_s` If you ran Lab 0 successfully, your SPEC2017 configuration file will point to a `run.sh`/`run.py` script, where you parse the SPEC2017 arguments (`${benchmark}` and `$command`) and call Pin. You will need to modify only this script for this lab. Use this script to sweep across different branch predictors for a benchmark at a time. Be aware: if you modify your SPEC config script, SPEC will re-build your benchmarks and everything will take significantly longer than if you only modify your run script, so be careful where you make edits in your infrastructure.

What to turn in

You should turn in all files of your pintool source code and a short write up and evaluation as a tarball (.tar.gz file). The source code needs to implement each of the predictors listed above.

You should also include a .pdf of an English language write-up that describes your design in as much detail as is necessary for us to understand what you did. You should argue in your write up that the amount of resources needed to implement your predictor is reasonable (and in doing so, you should explain how much memory is required to implement each predictor). You should evaluate your predictors on the SPEC benchmarks (as described above). We need to see branch prediction accuracies for each benchmark program that you run. A bar plot is a good format for this figure. If you run experiments using your predictor with multiple different predictor sizes, plotting results for a few different predictor table sizes to see how the accuracy changes with table sizes would be good to include in your write up as well. Please include both of your team members' andrewids in your pdf writeup; both team members should submit the pdf so we can easily track who has completed the assignment.