CSE471 Homework Assignment #1: Branch Prediction

Due Thursday, April 16th before lecture

1 Introduction

The purpose of this assignment is to supplement your conceptual knowledge of the design of different dynamic branch prediction schemes with concrete experience in implementing them and to test your intuition about their relative performance with data that reflects the actual performance of your implementations. As you learned in class, branch prediction is a common architectural technique for avoiding pipeline delays due to branches. A processor does branch prediction as it fetches a branch instruction. In order to continue fetching instructions that will actually be executed, the processor needs to know whether the branch will be "taken" (and consequently fetch the target instruction) or "not taken" (and fetch the fall-through code). Since a branch instruction is not actually evaluated for several cycles after it's been fetched, the processor predicts the branch outcome. The processor then uses the prediction to fetch instructions that are hopefully needed next.

2 Your Task

Dynamic branch prediction is implemented as a set of hardware data structures and logic that tells the processor whether a branch is likely to be taken or not taken. In this assignment, you will write a program that simulates the behavior of three branch predictors and, using a set of benchmark programs, compare their ability to correctly predict branch outcomes.

For all three predictors you'll use the same amount of hardware for their data structures, although for each predictor the hardware may be used to hold different types of data and may be configured differently. Your hardware budget is set at a maximum of 4 KB, which is a small, but still reasonable size for current low-end or low-power processors.

For one predictor, you will vary the size of these hardware structures and analyze the impact of changing the predictor's configuration on its prediction accuracy.

Finally, you should provide a quantitative analysis in support of your conclusions about the performance of the different predictor designs. Think about the difference between the predictors and how these differences should affect performance. Does your data support your hypotheses about the relative performance of the predictors? Your report should explain the data you collect, and describe your reasons for concluding that one particular predictor design is better than another.

2.1 The Predictors

Three predictors will be involved in your study:

- Static branch prediction
- Local (i.e., per-branch), two-bit predictor
- Global branch history register, global pattern history table, two-level adaptive (or correlating) predictor.

The static predictor will serve as your base case, against which you compare the others. It has already been implemented for you, to show you how branch predictors can be implemented in our software infrastructure.

For the dynamic predictors, you should implement the prediction with two-bit saturating counters, which will be discussed in section.

2.2 The Quantitative Analysis

Once you have implemented your predictors, you need to figure out how well they predict, using prediction accuracy as your metric. You will do this by running a set of programs on all of your branch predictor simulators. (The simulation infrastructure and benchmarks are described below in Section 3.) When your simulator is finished running a program, it will automatically report how many correct predictions each of your predictors made, as well as some other summary information about the execution. Use thee metrics to gain insight into the performance of the three prediction strategies.

For each predictor create a plot that shows its *relative* accuracy for each benchmark. It may be that for some applications some predictors work better and for other applications other predictors work better. If this is the case, say so and try to hypothesize why this may be the case.

In general which strategy, basing a prediction on the behavior of individual branches or basing it on execution paths through the program, works the best. How might you change the design/configuration of the more poorly performing strategy to enable it to match the better performer? Implement your new design and empirically evaluate the extent to which its performance approaches that of the better predictor. What are your conclusions from this experiment?

Write up your results and analysis in a report, as outlined in the report handout and illustrated in the sample reports on the course web site.

3 Simulation Infrastructure

The simulation infrastructure is built using a *binary instrumentation* tool called Pin (http://pintool.org). The first discussion section will cover what binary instrumentation is and how Pin works. For this assignment, you don't need to understand how to use Pin in detail.

Pin runs a specified program, subject to a set of rules and modifications described in a "pintool". A pintool is able to identify certain events during a program's execution, and call arbitrary code (*i.e.*, code you write) when those events occur.

For this assignment you'll use a pintool called `bp`. You are going to write a set of plugins for `bp` that implement various branch prediction strategies. `bp` is designed to call the code you wrote in these plugins when it encounters a branch during execution, thus simulating a particular branch prediction technique.

To make `bp` aware of your plugins, it takes a "-predictor" argument, that takes a *comma-separated* list of `.so` files. These shared object files should implement different predictors. For example, if you have two predictors implemented, called `two-bit.so` and `global.so`, and you want to run them both on an execution of some program, you would use the following command line:

```
pin -t bp -predictor two-bit.so,global.so -- <program> <args>
```

A version of Pin that we can all share is installed at `/cse/courses/cse471/15sp/PIN`.

Implementing a Predictor Plugin. You will write plugins as new classes that derive from the `Predictor` base class described in `Predictor.h`. An example to get you going is declared and defined in `StaticPredictor.h/.cpp`. You should use these files as a template for all your prediction schemes.

Your predictor plugins must define three methods:

1. The predictor's constructor
2. The `predictBranch` method

3. The `updatePredictor` method

Initializing Your Predictor. Your predictor's constructor should set up your predictor's data structures. The simulator infrastructure uses the "Factory" design pattern. In order for your predictor to be correctly loaded, you need to include the factory function `extern "C" Predictor * Create()` in the same source file as your predictor's definition. This function must not be a member of your predictor's class. This function should return a new instance of your Predictor's class.

Predicting Branches and Updating Your Predictor. The `predictBranch()` method runs before a branch executes, and receives the branch instruction's address as its argument. You will write it so that it returns the decision of the predictor (either `Taken` or `NotTaken`).

`updatePredictor()` is called after a branch executes. It takes a branch instruction's address, and the outcome of that branch as its arguments. This method should update the data structures associated with your predictor based on the outcome of the branch. `updatePredictor()` does not return a value.

Building Your Plugins. Please implement one predictor per .h/.cpp pair only (*i.e.*, putting two predictors in one file won't work). You can build your plugins using g++. Be sure to specify the `-fPIC` and `-shared` options, because your plugins need to be built as shared libraries. We've included a Makefile, and you can also add your plugins' source filenames to the list that includes StaticPredictor.cpp, and it will get built when you type `make`.

3.1 Benchmarks

The PARSEC benchmarks are installed at `/cse/courses/cse471/15sp/Benchmarks/parsec-2.1`, also available via a link in the Homework section of the course web pages. You will use these programs to evaluate your predictors' prediction accuracy. You can run the parsec benchmarks using the `parsecmgmt` script that is located at `/cse/courses/cse471/15sp/Benchmarks/parsec-2.1/bin`. The script will run the benchmarks for you, and you can tell it to run them in our simulator using command line arguments.

There are several important arguments:

- **-a run** This option tells parsecmgmt to run the benchmarks
- **-p benchmark name** This specifies which benchmark to run. If you specify nothing, parsecmgmt runs them all. To see which are available, you can run **parsecmgmt -a info**
- **-n 1** This option tells parsecmgmt to run with one thread. Stick to one because our simulator is not thread-safe.
- **-i input** The input can be either "test", "simsmall", "simmedium", "simlarge", or "native". Stick to "test" and "simsmall" or your experiments will take a very long time (hours).
- **-d working directory** When the benchmarks run, they create temporary files. This option determines where those files are created. Be sure it is a directory you can write to.
- **-s run script** This option tells parsecmgmt what program to use to run the benchmark. In our case, it should use our simulator, so you need to pass in the simulator command line (*i.e.*, the pin command line). This option is finicky; you need to enclose the simulator command line in single quotes, and you need to specify absolute paths to everything involved (pin, bp, and your branch predictors).

For your reference, we provide an example of the commands you must execute to successfully run parsec using our simulator on attu. First, you must set a few environment variables using the following three commands.

```
export LD_LIBRARY_PATH=/courses/cse471/15sp/lib32
export PATH=$PATH:/cse/courses/cse471/15sp/Benchmarks/parsec-2.1/bin
export PATH=$PATH:/cse/courses/cse471/15sp/PIN/PIN
```

After setting these environment variables, you can execute the following command to run parsec.

```
parsecmgmt -a run -p blackscholes -i test -n 1 -d . -s
'pin -t /cse/courses/cse471/15sp/PIN/PIN/source/tools/CSE471-BP/obj-intel64/bp.so
-predictor /cse/courses/cse471/15sp/Assignments/HW1-BranchPrediction/StaticPredictor.so
-out ./test.out -- '
```

Notice several things about this command line. First, "`.`" is passed as the argument to "`-d`". Second, the entire "`-s`" argument is enclosed in single quotes. Third, there is a space after the "`--`" at the end of the "`-s`" argument. The "`run.sh`" script in the homework files contains this code, feel free to use this script as a starting point to automate your experiments.

Your quantitative evaluation must include the PARSEC benchmarks, but is not restricted to these programs. A cool aspect of using Pin is that Pin-based simulators can simulate *any* x86 binary. Feel free to experiment.

We recommend that while you are developing your branch predictors, you write test cases for yourself to make testing more controlled. The benchmarking infrastructure has a long turn-around time, and if you are debugging, it is more productive to use small test programs that evaluate particular branch prediction circumstances.