README

CSE240A Branch Predictor Project

Table of Contents

Introduction

As we've discussed in class, branch prediction is critical to performance in modern processors. An accurate branch predictor ensures that the front-end of the machine is capable of feeding the back-end with correct-path instructions. Beyond its criticality in processor execution, branch prediction is an interesting problem. How do you make accurate predictions on little data using small, fast hardware structures.

For this Project you will be implementing various branch predictors in a simulated environment. We have provided a starting framework to help you design your predictors. The framework (main.c) will perform all of the command-line switches as well as the reading in of the trace files. You will implement your predictors by completing all of the TODOs in the predictor.c file. Note that this is the only

file in which you are able to make changes, as it will be the only file of yours we use for grading.

Code Integrity

Please make sure you do not copy a single line of code from any source. Not from other students, not from the web, not from anywhere. We have very sophisticated tools to discover if you did. This is a graduate class and we have the very highest expectations for integrity. You should expect that if you do so, even in very small amounts, you will be caught, you will be asked to leave the program, and if an international student, required to leave the country.

Get Started

As mentioned, we provide a starting framework to help you design your predictors. The source code (including some traces for testing) is in github and you can get it with `git clone https://github.com/prodromou87/CSE240A.git`.

Alternatively, you can download it from our github page.

You have the option to write your project in C, C++ or Python. We only provide a framework written in C and we strongly recommend you use it, primarily to ensure compatibility with our autograder.

If you decide to use some of the other supported languages, you will have to implement everything. You also have to make sure that running `make` in the src directory generates an executable named 'predictor'. During grading, our script will run a `make clean`, followed by a `make` command. Make sure that this step is not going to delete your code, especially if you are writing it in Python. Finally, make sure that your project runs with the exact same commands as this document describes. Python submissions must run with `./predictor`, without requiring `python ./predictor`.

Working with Docker

Your projects will be graded using gradescope and an automatic grader based on Docker. The compiler used by the autograder is gcc-5.4, and runs Ubuntu 16.04. You should be able to develop this project on your own machines, but in case you want to ensure compatibility with our autograder, we provide a Docker image with the same configuration.

You will first have to install Docker (for simplicity consider Docker a very lightweight VM) on your machine (or in a VM), following the instructions found here. Once installed, you can pull our image by opening a shell (for Windows machines, powershell seems to work better than the cmd prompt) and typing:

```
docker pull prodromou87/ucsd_cse240a
```

This command will download and build our docker image. It will take a while, but you only have to do this step once. To verify that you have the image, you can run `docker images` and check the the image is listed.

Once you have it, you can start an interactive shell in Docker with

```
docker run --rm -it prodromou87/ucsd_cse240a

The --rm flag will delete the running container once
you exit it so it doesn't keep consuming resources
from the host machine.

The -it flag will start an interactive session so it's
necessary if you want a shell to work with.
```

For development purposes, you will want to mount the project directory in Docker so you can see your code, compile and run it. To do that, you change the previous instruction slightly:

```
docker run --rm -it -v /path/to/project/directory:/path/
to/mount/point prodromou87/ucsd_cse240a

Windows users will have to write the path as follows (No-
  tice the lowercase 'c'):

docker run --rm -it -v //c/path/to/project/directory:/path
/to/mount/point prodromou87/ucsd_cse240a
```

If necessary, you can mount more directories with multiple `-v` flags. Once you get a shell, you can confirm the folder has been mounted with `ls /path/to/mount/point`. You can now compile and run your project following the instructions provided in this document. You can also modify the code on your host machine using your preferred editor and only switch to docker for compiling/running. You don't need to restart docker every time since changes in the mounted directory will immediately be visible in Docker.

Traces

These predictors will make predictions based on traces of real programs. Each line in the trace file contains the address of a branch in hex as well as its outcome (Not Taken = 0, Taken = 1):

```
<Address> <Outcome>
Sample Trace from int_1:

0x40d7f9 0
```

```
0x40d81e 1
0x40d7f9 1
0x40d81e 0
```

We provide test traces to you to aid in testing your project but we strongly suggest that you create your own custom traces to use for debugging.

Running your predictor

In order to build your predictor you simply need to run `make` in the src/ directory of the project. You can then run the program on an uncompressed trace as follows:

```
./predictor <options> [<trace>]
```

If no trace file is provided then the predictor will read in input from STDIN. Some of the traces we provided are rather large when uncompressed so we have distributed them compressed with bzip2 (included in the Docker image). If you want to run your predictor on a compressed trace, then you can do so by doing the following:

```
bunzip2 -kc trace.bz2 | ./predictor <options>
```

In either case the `<options>` that can be used to change the type of predictor being run are as follows:

```
  --help       Print usage message
  --verbose    Outputs all predictions made by your
               mechanism. Will be used for correctness
               grading.
  --<type>     Branch prediction scheme. Available
               types are:
        static
        gshare:<# ghistory>
        tournament:<# ghistory>:<# lhistory>:<# index>
        custom
```

An example of running a gshare predictor with 10 bits of history would be:

```
bunzip2 -kc ../traces/int1_bz2 | ./predictor --gshare:10
```

Implementing the predictors

There are 3 methods which need to be implemented in the predictor.c file. They are: **init_predictor**, **make_prediction**, and **train_predictor**.

```
void init_predictor();
```

This will be run before any predictions are made. This is where you will initialize any data structures or values you need for a particular branch predictor 'bpType'. All switches will be set prior to this function being called.

```
uint8_t make_prediction(uint32_t pc);
```

You will be given the PC of a branch and are required to make a prediction of TAKEN or NOTTAKEN which will then be checked back in the main execution loop. You may want to break up the implementation of each type of branch predictor into separate functions to improve readability.

```
void train_predictor(uint32_t pc, uint8_t outcome);
```

Once a prediction is made a call to train_predictor will be made so that you can update any relevant data structures based on the true outcome of the branch. You may want to break up the implementation of each type of branch predictor into separate functions to improve readability.

Gshare

```
Configuration:
    ghistoryBits    // Indicates the length of Global History kept
```

The Gshare predictor is characterized by XORing the global history register with the lower bits (same length as the global history) of the branch's address. This XORed value is then used to index into a 1D BHT of 2-bit predictors.

Tournament

```
Configuration:
    ghistoryBits    // Indicates the length of Global History kept
    lhistoryBits    // Indicates the length of Local History kept in the PHT
    pcIndexBits     // Indicates the number of bits used to index the PHT
```

You will be implementing the Tournament Predictor popularized by the Alpha 21264. The difference between the Alpha 21264's predictor and the one you will be implementing is that all of the underlying counters in yours will be 2-bit predictors. You should NOT use a 3-bit counter as used in one of the structure of the Alpha 21264's predictor. See the Alpha 21264 paper for more information on the general structure of this predictor. The 'ghistoryBits' will be used to size the global and choice predictors while the 'lhistoryBits' and 'pcIndexBits' will be used to size the local predictor.

Custom

Now that you have implemented 3 other predictors with rigid requirements, you now have the opportunity to be creative and design your own predictor. The only requirement is that the total size of your custom predictor must not exceed

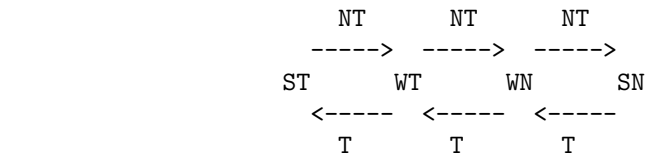
(64K + 256) bits (not bytes) of stored data and that your custom predictor must outperform both the Gshare and Tournament predictors (details below).

Things to note

All history should be initialized to NOTTAKEN. History registers should be updated by shifting in new history to the least significant bit position.

```
Ex. 4 bits of history, outcome of next branch is NT
  T NT T NT   <<  NT
  Result: NT T NT NT
```

```
All 2-bit predictors should be initialized to WN (Weakly Not Taken).
They should also have the following state transitions:
```



The Choice Predictor used to select which predictor to use in the Alpha 21264 Tournament predictor should be initialized to Weakly select the Global Predictor.

Grading

All grading will be done with respect to your predictor's Mispredicitons Rate, as well as its correctness (for Gshare and Tournament) compared to our implementation.

You get 10 points for correctness. If your predictions match the correct output, you get full points. You get 30 points if your custom predictor beats one of the other two, and +10 points (40 total) if you beat both of them.

Finally, the 10 best predictors (in terms of misprediction rate), will receive extra points. First place gets 10 points, second place gets 9, third gets 8 and so on. **The maximum grade is 60. If your custom predictor does not rank in the top 10, the maximum score you can get is 50/60.**

You should do most of your development on your own machine. If you face any issues when you submit your project in gradescope, try to run your project in our Docker image to ensure compatibility with the autograder, or post the error message in Piazza.

Turn-in instructions

DUE: May 26 2017 - Submissions after 11:59:59 PM are considered late

Late projects are allowed for at most 3 days after the project due date. For each day late a 10% grade penalty is automatically attached (e.g. 3 days late = 30% penalty). After 3 days, missing projects will recieve a zero. A project is considered late at 12:00:01 AM (Which is 1 second past Midnight).

To submit your project, compress the 'src' folder in a **.zip** (not .rar) file and upload it in gradescope. Select only the source files required for compilation and nothing else (binaries, traces etc should not be submitted). This will trigger our autograder to begin grading. You are allowed to submit multiple times.

Gradescope has a time limit for autograders. Make sure that your code does not do anything unnecessary. For comparison purposes, we will announce our implementation's running time as soon as we optimize it.

Our autograder runs two sets of tests:

We first ensure that your code is compatible with our autograder. If your code fails any of these tests, you will be notified immediately, so don't leave the screen before you see the grading outcome. Specifically, this set of tests checks that:
- `make` produces an executable named `predictor`
- The output produced by your executable has the expected format

Once you pass the compatibility test, we grade the output produced by your code. You will be able to see your score on some of our test cases, but some will be hidden. Your overall grade will not be visible until after the project's due date.

Note: Gradescope expects pass/fail tests but we will be reporting percentages. If you don't score 100%, Gradescope considers it a failed tests. Do not be concerned when you see failed tests (but be concerned if your score is low and re-submit)