ECE 486/586
Winter 2024
Final Project
Branch Prediction

Overview

In this project your team will simulate two branch predictors. For the first, you are required to implement a tournament predictor based upon the design employed by the Alpha 21264 described in Kessler, R.E., "The Alpha 21264 Microprocessor", IEEE Micro, March/April 1999, pp 24-36. We will also discuss this briefly in class.

You will then design and implement a *second* predictor to enter a competition among all predictors created by other students in the course.

The competition predictor can incorporate almost any branch prediction strategy you desire in an attempt to minimize mispredictions. It is subject to the following constraints:

- It must use no more than 4K + 4 bytes of storage
- All tables must be sized as powers of two
- Associative tables ≤ 8-way do not incur a size penalty
- Tables with associativity > 8-way incur a storage penalty equal to the size of the table
- If you use random replacement, it must be reproducible
- Multiplying or dividing by numbers other than powers of two are not permitted

Framework

You'll be provided with a C++ software framework including several execution traces. The framework will read a trace and call your predictor with a record containing information about each branch. Your predictor will make a prediction and then be told the actual outcome of the branch. Upon completion of the trace, the framework will print out statistics including your misprediction rate (expressed as mispredicts/1000 branches).

You are responsible for coding two functions:

```
bool get_prediction(const branch_record_c* br, const op_state_c* os)
void update_predictor(const branch_record_c* br, const op_state_c* os, bool taken)
```

The framework will open a trace file and then call your get_prediction(), passing it a record with the current branch information. Your function will return true if you're predicting the branch to be taken and false if not-taken. The framework will then call your update_predictor() to let it know the actual result of the branch so that you can update branch history, etc. The branch_record is defined in the tread.cc file in the framework. You can ignore op_state_c.

You do not need to understand details of the framework. Just add your functions to the predictor.cc file. The default one will not do any predictions but will show you how to access the branch record and can be used to demonstrate how to build the framework without errors.

The branch_record_c looks like this:

```
class branch_record_c {
public:
    branch_record_c();
    ~branch_record_c();
    void init();                 // init the branch record
    void debug_print();          // print info in branch record (for debugging)
    uint   instruction_addr;     // the branch's PC (program counter)
    uint   branch_target;        // target of branch if it's taken; unconditional are
always taken
    uint   instruction_next_addr;// PC of the static instruction following the branch
    bool   is_indirect;          // true if target is computed; false if it's PC-rel;
returns are also considered indirect
    bool   is_conditional;       // true if branch is conditional; false otherwise
    bool   is_call;              // true if branch is a call; false otherwise
    bool   is_return;            // true if branch is a return; false otherwise
};
```

You'll need your own copy of the framework. You can extract it with the following command:

```
% tar xvf ~faustm/Documents/ECE586/framework.tar
```

It will create a subdirectory called framework in your current working directory and extract everything to that directory. Do not uncompress the trace files – the framework dynamically uncompresses them as needed.

You can build the framework (and your code) with:

```
% cd framework
% make
```

This will create an executable called predictor. The program takes a single command line argument which is the base filename (without the .bz2 suffix) of a compressed trace file. There are three trace files in the traces subdirectory you can use. So, for example:

```
% ./predictor traces/DIST_INT_1
```

Grading Criteria

The project is worth 100 points as follows:

Alpha 21264 Predictor	
Compiles	10
Runs without crashing	10
Produces correct results	30
Code (quality, readability)	20
Presentation/Report	30

The competition predictor can be awarded as many as 25 bonus points as follows:

Compiles, Runs without crashing, beats Alpha predictor	10
Competition predictor in top quartile of competition predictors	5
Competition predictor in top half of competition predictors	10

The team with the top-performing predictor will win a small prize and be able to smugly mock their unworthy competitors.

Your demo/presentation should not exceed 20 minutes and must include a description of your algorithm, the rational, a "space" budget that accounts for all storage used, a description of the implementation, a description of your testing procedures, and the statistics on the public benchmarks, ending in a demonstration showing the compilation and execution of your code on the competition benchmarks. A written report summarizing the above information must be turned in at the time of your presentation.

Teams/Demos/Presentations

Enroll yourself into a group on Canvas with your teammate. To avoid problems, before adding yourself to a group, make sure it's either empty or contains someone who has agreed to be on your team.

All demos and presentations/demos will take place during finals week. After agreeing on a time slot with your teammate, reserve your team's slot in this Google doc:
https://docs.google.com/spreadsheets/d/1zuDzDvZwJ5UW-4CWl5ogyYclrDCBYv3M2lFrnRHosK8/edit?usp=sharing

Use your team number (from your Canvas self-enrollment) and both your team members' names.

Your team's work should be submitted via Canvas by one team member as a single zip file containing two directories – one for your Alpha branch predictor and one for your competition predictor (called alpha and competition, respectively).