Computer Assignment 2 **Deadline:** Friday, Nov 7, 11:59 PM

Fall 2025 (Upload it to Gradescope.)

How is this going to be graded:

This project has only **bonus** points (CA1 has 50% and CA3 will have 50% of your CA grade). The bonus will be added to your average score in CAs, and if it exceeds 100%, it will help your overall grade (recall that CAs have 30% of the overall score). The bonus will be calculated as follows:

- You will get 3% (bonus to your average CA) if your MPKI is below 5.3, but you are in the lower half of the class.
- You will get 5% if your MPKI is ranked in the top half of the class (this is relative to the others).
- You will get 8% (bonus) if you are in the top 20% of the class.
- 10% (bonus) if you are in the top 5% and an additional 3, 2, or 1% bonus for the first, second, and third place (respectively).

For example, if you get 100% in CA1, 100% in CA3, and finish in the top half of the class, your average score for CAs will be 100+5 = 105% (and this is 30% of your total grade).

Important Note: since this is a competition and the scores are entirely bonus, we will not be able to extend the deadline or accept late submissions.

Project Description

The goal of this project is to design a branch predictor. In the lecture, we described a simple 2-bit predictor, but in this project, we want you to learn more advanced design options. Your designs will be evaluated against each other to form an in-class competition/championship. The leaderboard will be posted here (Leaderboard-ECE116C-F25).

This championship was originally designed and used in the Branch Predictor Championship Competition at a top-tier computer architecture conference (ISCA) in 2016 (link), and all the credits go to the original designers and organizers of the framework. The more recent version of the competition at ISCA 2025 can be found here: https://ericrotenberg.wordpress.ncsu.edu/cbp2025/

Note that there exist available designs (e.g., see this: https://github.com/ramisheikh/cbp2025) and you are free to use them as the baseline. However, since this is a competition, you need to design something this is better than what is available online.

Code:

The infrastructure for the branch prediction competition can be downloaded from here: https://drive.google.com/file/d/1dtp3tvQUVt6rhVHmMCmbmN4NcR0if0Ja/view?usp=sharing.

It has three directories:

- ***src.*** This directory contains C++ code for reading traces and driving the branch predictor code. The file my_predictor.h is distributed with a simple example branch predictor. Modify it to implement your predictor. The Makefile in this directory will build a program called predict that will read a trace file and print the number of mispredictions per 1000 instructions on the standard output.

- ***traces.*** This directory contains the distributed traces in directories named for benchmarks. You do not need to change anything here.

- ***doc.*** This directory contains the documentation, including this file.

How to Run:

The csh script runs the program using the traces directory. Compile the program by changing to the src directory and typing make. Then run the program on all the traces by changing to the top-level cbp2 directory and typing run traces. Use this command to run:

```
“csh run traces”
```

The output of your program should be something like this:

```
traces/164.gzip/gzip.trace.bz2             12.473
traces/175.vpr/vpr.trace.bz2               13.415
traces/176.gcc/gcc.trace.bz2               11.254
traces/181.mcf/mcf.trace.bz2               15.837
traces/186.crafty/crafty.trace.bz2         5.837
traces/197.parser/parser.trace.bz2         10.008
traces/201.compress/compress.trace.bz2     7.831
traces/202.jess/jess.trace.bz2             1.562
traces/205.raytrace/raytrace.trace.bz2     2.756
traces/209.db/db.trace.bz2                 3.909
traces/213.javac/javac.trace.bz2           2.267
traces/222.mpegaudio/mpegaudio.trace.bz2   2.188
traces/227.mtrt/mtrt.trace.bz2             2.657
traces/228.jack/jack.trace.bz2             3.033
traces/252.eon/eon.trace.bz2               1.807
traces/253.perlbmk/perlbmk.trace.bz2       2.554
traces/254.gap/gap.trace.bz2               3.926
traces/255.vortex/vortex.trace.bz2         1.222
```

```
traces/256.bzip2/bzip2.trace.bz2          0.094
traces/300.twolf/twolf.trace.bz2          21.489
average MPKI: 6.305
```

"Average MPKI" (misprediction per thousands of instructions) number is what you need to report on the Google sheet.

How to Design:

Please make sure to watch this pre-recorded video and slides posted on the website.

Generally, there are numerous resources available that can provide inspiration for ideas. To get started, you can search for existing write-ups and designs or read about well-known branch predictors like TAGE, G-share, and their variations. Additionally, an interesting avenue to explore would be utilizing machine learning for prediction. Keep in mind that this aspect is research-oriented and relies on your personal level of interest and enthusiasm. Feel free to brainstorm your ideas with your classmates, and remember that this is a competition so you may not want to share too much!!

Also, note that, unlike the original competition, there is no design constraint for your predictor (e.g., storage size) in this project. However, try to consider realistic tradeoffs. For example, try to balance the predictor performance with the storage requirements (e.g., don't use a predictor that needs MBs of storage, etc.). In your report (see the details below), you should explain your design decisions and the area overhead of the design.

Writing Your Branch Predictor Simulator

Write your code in my_predictor.h, replacing the simple gshare predictor that comes with this infrastructure. The code in my_predictor.h defines two classes:

- ***my_update.*** This class is subclassed from branch_update which is defined in predictor.h. Your branch predictor will return a pointer to branch_update when it makes a prediction. The direction_prediction() method in this object should return true to predict that a branch is taken, or false to predict that a branch is not taken. When the driver code updates the branch predictor with the correct outcome, the same pointer will be sent to your update code. You can expand your my_update class to include the state you want to refer to between calls to the prediction code. For example, the distributed my_update class stores the index in the gshare table.

- ***my_predictor.*** This class is subclassed from the abstract class branch_predictor defined in predictor.h. Classes subclassing from branch_predictor must define a branch_update method with the following signature: virtual branch_update *predict (branch_info &);

The driver program will call your predict method for every branch trace, giving information from the trace in a branch_info struct that is defined in branch.h. This struct contains fields for the branch address, the x86 branch opcode for a conditional branch, and a set of flags giving information on whether a branch is e.g., a return, a call, a conditional branch, etc. The return value from predict should be an object of (sub)class branch_update that gives the direction prediction for a conditional branch.

In addition to predict, you may define a method with the following signature in your my_predictor class: virtual void update (branch_update *, bool, unsigned int);

The driver program will call your update method (if any) to give you a chance to update your predictor's state. The parameters are the branch_update pointer your predict method returned, a bool that is false if a conditional branch was not taken, true otherwise, and an unsigned integer target address of the branch.

Note: The driver program only reports mispredictions for conditional branches, but it calls the branch predictor "predict" and updates methods on every branch. That's because you might want to use the information in a stream of all branches, not just conditional branches, to help you predict conditional branches. You are not required to try to predict these non-conditional branches.

The Traces

Each of the distributed trace files represents the branches encountered during the execution of 100 million instructions from the corresponding benchmark. The traces include branches executed during the execution of benchmark code, library code, and system activity such as page faults. After the traces are generated, they are encoded (credits to Daniel Jimenez). Then they are compressed with bzip2. The compression scheme is lossless; the traces sent to your predictor are bit-for-bit identical to the traces collected from the running benchmarks.

System Requirements

This infrastructure uses a few commands that are present on most Unix systems. If the pathnames for bzip2 and/or gzip on your system are different from those in trace.h then please customize them.

What to Submit:

1. Gradescope
    a. Your code. It should be well-commented. Note that there is no auto-grading on Gradescope.
    b. A PDF file briefly explaining your design methodology, tradeoffs, and your design storage overhead. There is no specific page limit, but 3 or 4 pages is reasonable. Try to be informative but concise.

2. Google Sheet
    a. Add your name, UID, and Average MPKI on the Google Sheet. Do not modify anything else, and make sure that your number is inserted correctly and matches the code uploaded on Gradescope. We will run all designs after the deadline and cross-check the numbers inserted in the Google Sheet. If there is a mismatch, we will use the number produced by your code and remove your number if you have made a mistake (rounding errors up to 0.001 are ignored). The sheet will also show the sorted list. You can work more on your design if you want to further improve your ranking, as it might change once more people submit their code. (Make sure to update your code and Google sheet if you have updated your design.)