6.5900
Computer System Architecture
Lab 2

Assigned Sep. 30, 2024 ***Due Oct. 18, 2024***

http://csg.csail.mit.edu/6.823/

Warning: This lab is open-ended. Do not wait until the last minute to start the lab.

Summary

Branches create control hazards in a pipelined processor. When an instruction is loaded, the processor must decide which instruction to fetch next. Most of the time, the subsequent instruction in x86 is located in the bytes immediately following the current instruction address. Thus, just fetching the next bytes for execution is a reasonable strategy. For branch instructions the next instruction may lie elsewhere in memory. In this case loading the next instruction will likely be incorrect, and the pipeline will stall as the processor fetches the correct next instruction.

Incorrectly fetching the next instruction following a branch instruction can be extremely detrimental to performance, since branch instructions account for 1/6 the total number of instructions. Architects use branch prediction to deal with control hazards in deeply pipelined general-purpose processors. At a high level, the branch prediction interface is straightforward: the branch predictor accepts an instruction address from the processor and returns a branch direction. The value of the subsequent instruction address is predicted by another structure, such as a branch target buffer (BTB). Notice that a trivial branch predictor implementation could always returns a 'not taken' result, i.e., step to the next sequential instruction. In this lab, you will research predictors and implement a branch predictor model.

Setting up

First, set up your environment for Pin. You'll need to do this each time you log in to work on the labs.

```
% add 6.823
% source /mit/6.823/Fall24/setup.sh
```

To obtain the materials for lab 2, use the following commands:

```
% cd 6.823/$USER
% cp -r $LAB2FILES ./
% git add lab2handout
% git commit -m "Lab 2 Initial Check-in"
```

In the lab2handout directory that was just created, you should find a make file, some sample source code, and a test script. Type the following at the command prompt:

```
% cd lab2handout
% make
```

Make will build the `bpredictor` Pintool. The `bpredictor` Pintool can be invoked from the command line in the following manner:

```
% pin.sh -injection child -ifeellucky -t bpredictor --
[benchmark]
```

We have provided a test perl script `lab2test.pl`. The perl script will invoke Pin using the `bpredictor` Pintool on multiple SPEC binaries. To invoke the perl script, type:

```
% ./lab2test.pl
```

In addition to this public test, we will also test your code using a few additional private benchmarks, which have similar behavior to the public benchmarks. Since the lab is open-ended, your results may differ from your classmates, but you are still encouraged to compare your results with your classmates.

We have also provided you a simple python script `accuracy.py` to check the accuracy of your design across the 10 public benchmarks. Simply run the following command:

```
% ./accuracy.py [results_directory]
```

Lab Task

The purpose of this lab is to explore branch predictor designs. Your implementation will extend the `BranchPredictor` class to implement your own branch predictor. Logically, this branch predictor will hang off of the processor core and accept branch address queries from the processor. Given a branch address, the branch predictor will respond with a bit denoting whether the branch is predicted to be taken or not taken. The `BranchPredictor` class consists of the following functions:

`makePrediction`: This function accepts the address of a branch instruction. The function will return true if the branch is predicted to be taken and false otherwise.

`makeUpdate`: This function is a feedback from the processor core. It accepts parameters: the branch address, your original prediction, and the actual direction of the branch. You will use this information to update your internal branch predictor state as you see fit.

The Pintool uses your `BranchPredictor` implementation as follows. When the tool encounters a branch instruction, it calls `makePrediction` to get your prediction. It then determines the actual branch direction and calls your `makeUpdate` function.

Notice that your branch predictor will only need to predict whether the branch is taken or not. You will not have to generate a potential branch target. Your branch predictor is allowed to use no more than 33 Kilo**bits** (33000 bits) of persistent storage. Using more than 33 Kilobits will disqualify your branch predictor from earning any credit for the accuracy portion of the lab grade. Needless to say, this would be extremely detrimental to your grade. You may use these bits in any manner you see fit, and you may use any amount of computation that your branch predictor requires. At the end of the execution, we will dump statistics about the performance of your branch predictor, which we will track in our code. Note that this limit is only on space that you actually use, so you can use data structures that require more space allocation (e.g., a 11,000 entry array of uint_8's instead of 3-bit wide entries).

As a side note, the sub-field of computer architecture related to branch prediction is rather rich and contains many important ideas. Although implementing the branch predictors discussed in lecture will likely obtain most of the credit, if you hope to beat the TA's reference branch predictor, you will probably have to do some research on your own.

Although your solution will not be graded on its performance in terms of wall clock time, you should note that your Teaching Assistant is impatient. The TA solution runs the sample testbench in less than 20 minutes on the class machines (with normal load). For grading purposes, we will allow your Pintool to run for an order of magnitude more time than ours requires (around 4 hours). After ten hours, we will kill your submission and assign a grade based on progress to that point. Do not write horrendously inefficient code: it makes kittens sad.

When you have completed the lab to your satisfaction, submit your changes to the git repository. The deadline for submission is 23:59:59 EDT 18 Oct 2024. We'll grade whatever code you have checked in by the deadline. **No Late Submissions will be accepted!** Seriously.

Lab Questions

Your response to the lab questions should be typed in lab2questions.pdf in the lab2handout directory. The course material necessary to answer some of the questions will be covered after the distribution of this handout, so if some of the material looks foreign, it probably just hasn't been covered yet. Some questions may require coding, and as such should not be put off until the last minute.

1. **It's absolutely pitiful to code like that.** Your TA has written the lab starter code in a slightly suboptimal way, in terms of simulation speed. Figure out the inefficiency and fix it. Explain your change. Does it make a big difference in speed?

2. **Hardware? Don't talk about hardware.** Explain the operation of your implementation. Detail the general algorithm that you used and explain why you chose it. Why does the algorithm that you finally chose outperform the other

algorithms you tested and those discussed in class? Describe how you allocated the storage bits of your branch predictor. Don't forget to cite your sources.

3. **Are you kidding me? Hardware?** We gave you unlimited computation to develop your branch predictor. Could your implementation be built into a processor? Explain why or why not. Discuss which elements of your algorithm would cause problems if implemented in silicon.

4. **I'm just hoping we can implement a software model.** The branch predictor you implemented is likely to be missing several of the elements of a real branch predictor. For example, in a real machine several predictions might be made before an update occurs. Why might that happen? Explain the consequences of this on your predictor in terms of impact on prediction and hardware changes that might be added to address that impact. Don't neglect the impact on misspeculation recovery.

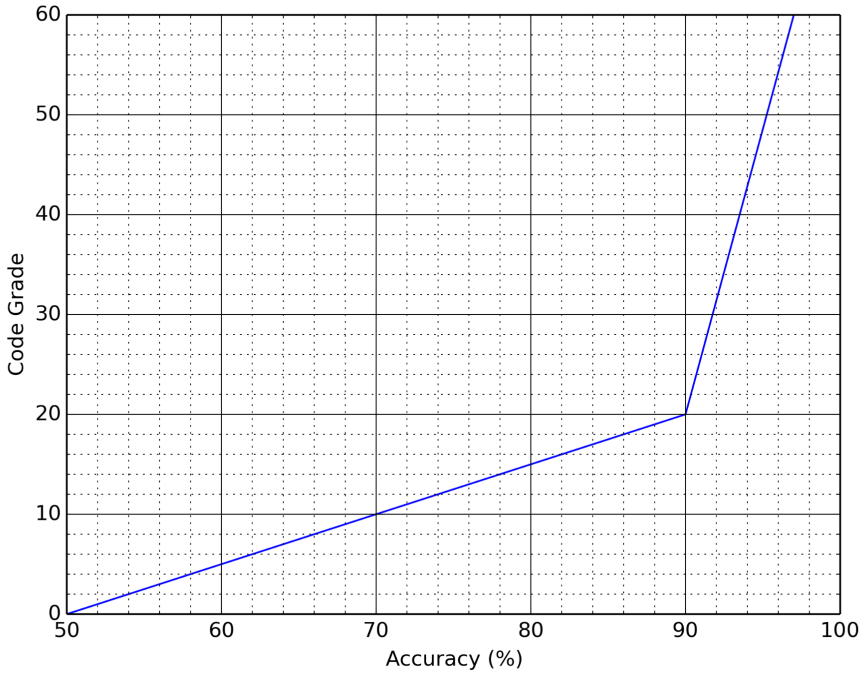
When you have answered these questions to your satisfaction, put them in a file called lab2questions.pdf in your lab2handout directory, then run the following to add, commit, and **push** them.

```
% git add lab2questions.pdf
% git commit -a -m "Lab 2 Questions Check-In"
% git push origin master
```

As with the lab code, we'll grade whatever you have pushed by the deadline.

Lab Grading

10% - Submission compiles and produces results without crashing
60% - Accuracy grade based on *geometric mean accuracy* across the public and private benchmarks. You will receive no credit for accuracy unless your accuracy exceeds 50% (which could be obtained by uniform random guessing). Increasing accuracy above 50% will improve your grade. 90% accuracy will be sufficient to earn one third of the credit for accuracy, and 97% accuracy is needed to earn full credit. Refer to the figure below for the exact grading curve.



30% - Responses to questions.

Advice on Mine Sweeping

There may be bugs in either our code or infrastructure. If you notice any `interesting' or `unexpected' behavior it could be a problem in the code or the servers that we provided. If you encounter a bug, report the problem to 6823-tas@csail.mit.edu or on Piazza, and try doing the lab on another machine.