EC 513 Computer Architecture
Lab 2
Worth 12%

Assigned Feb. 13, 2018

Due Feb. 27, 2018

https://ascslab.org/courses/ec513/index.html

Warning: This lab is competitive and open-ended. Do not wait until the last minute to attempt to complete the lab.

Summary:

Branches create instruction 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. The mechanism that architects use to deal with instruction hazards is branch prediction. At the high level, branch prediction is somewhat simple to describe a branch predictor interface: the branch predictor accepts an address from the processor and returns a branch direction. The actual address is predicted by another structure such as a BTB. Notice that a trivial branch predictor implementation 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:

To obtain the materials for lab 2, `download the file at:`

`http://bit.ly/2GbqTKR`

Extract the file in your **labhandouts/** directory, next to lab 0.

In the lab2handount directory that just got created, you should find a make file, some sample source code, and a test script. Type the following at the command prompt:

```
% cd lab2handout
% chmod a+x lab2test.pl
% make
```

Make will build the bpredictor Pin tool. Attempting to run the Pin tool will result in a failure message, stating that no Pin tool has been implemented. The bpredictor Pin tool can be invoked from the command line in the following manner:

```
% pin -t bpredictor -- target_executable
```

We have provided a test perl script `/lab2test.pl`. The perl script will invoke Pin using the bpredictor Pin tool on multiple PARSEC and SPLASH2 binaries. To invoke the perl script, type:

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

Although we provide a script that will test your Pin tool, the script will not verify your Pin tool, and we will not release the expected results of the test cases. Furthermore, we reserve the right to run test cases not included in the released test script. You are encouraged to compare your results with your classmates.

Lab Task:

The purpose of this lab is to explore branch predictor architectures. To implement this task, you 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 taken or not taken. The Branch Predictor class consists of the following functions:

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

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

In the pin tool that we have implemented, when we encounter a branch instruction, we will call ***makePrediction*** to get your prediction. We will then determine the correct prediction and call your ***updatePrediction*** function.

Notice that your branch predictor will only need to predict whether the branch is taken or not. You will not have to present a potential branch target. Your branch predictor is allowed to use no more than 33 Kilobits of persistent storage. Using more than 33 Kilobits will disqualify your branch predictor from the competition portion of the assignment. 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.

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 proposed in class will likely obtain half of the correctness credit, if you hope to best 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 TA is an impatient person. His solution runs the sample testbench in less than one hour on the class machines (with nominal load). For grading purposes, we will allow your pin tool to run for an order of magnitude more time than ours requires (around 10 hours). After ten hours, we will kill it 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 blackboard. The deadline for submission is 23:59:59 EST 27 February 2018. ***No Late Submissions will be accepted!***

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 require substantial additional coding, and as such should not be put off until the last minute.

1. **It's absolutely pitiful to code like that.** Your instructor has written the lab handout 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? 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 mispeculation recovery.

Lab Grading:

10% - Submission compiles

60% - **Competitive grade based on the following break-down based on aggregate performance on the public and private benchmarks. Notice that defeating the TA's implementation garners full credit.**

- 20% - below 80% accuracy
- 30% - Any implementation achieving greater than 80% accuracy
- 40% - Any submission achieving 95% accuracy on the benchmarks
- 50% - Any implementation achieving 98% accuracy on the benchmarks
- 60% - Any implementation that beats the TA's implementation.

30% - Quality of lab question responses.