1 Introduction

In this homework, we study pairs of simple programs, which exhibit very different performance despite performing the same function, executing essentially the same algorithm, and overall looking very similar at first glance.

2 Prerequisites

You will need the tools from hw1 to do this homework. In addition, you will find `objdump -d`, `gcc -S`, or `disass` in `gdb` useful for inspecting the produced executable.

Please review the chapter on the *scientific process* in the course notes, to prepare for this homework.

3 Assignment

Your job is to explain, for each pair of programs, the fundamental cause of the performance difference. We'll use the scientific process outlined in the course notes for this. For exach pair of programs, the observation is that the execution time differs. Thus, for each pair of programs:

- Present a brief, falsifiable hypothesis explaining the discrepancy.
- Describe a falsifiable prediction that the hypothesis makes about the pair of programs.
- Present experimental results that validate the prediction.

You can use any tool you want to investigate the program, but be careful about the order of hypothesis, prediction and experiment. If you did your experiments before you came up with the final hypothesis, you've violated the principle, and are more likely to end up with an invalid hypothesis.

To give a simple example, a program that divides by 2 runs much faster than a program that divides by 3. Reading the assembly (a sort of measurement), you find that the first program uses a shift instruction instead of division. The hypothesis *the compiler replaces divide-by-two with shift-right-one* is based on your measurement, and doesn't produce any alternative predictions. A better hypothesis would be *the compiler replaces division by any factor of two with shift-right.* This hypothesis produces the falsifiable prediction that divide-by-8 is replaced by shift-right-3, which is easily tested by modifying the program.

Clone the hw2 template from `https://github.com/uicperformance/fastandslow.git`.

3.1 one vs. one_opt

`one` runs almost 1000× slower than `one_opt`. Formulate a hypothesis explaining why, use the hypothesis to produce a prediction about compiler behaviors, and test it. *hint: what happens if you try to print out the value of count after the loop?*

3.2 two_opt vs. two

`two_opt` runs much faster than `two`, despite modifying a global variable. This is somehow different from the previous case.

hint: Also, what happens if you change `count++` *to* `count*=8; count/=3;`?

3.3 three_opt vs. two_opt

`three_opt` is a lot slower than `two_opt`. What does the keyword *volatile* do? You can probably google your way to a great hypothesis, but let's see a good prediction and test. *hint: can you think of an interesting replacement for the statement* `count++`*?*

3.4 array packed vs. 64-byte spread

Use the `cachestress` program, to run these two commands:

- `cachestress -s 1048576 -i 1`
- `cachestress -s 1048675 -i 64`

The 64-byte interval version significantly slower than the 1 byte version, even though they perform the same number of operations. Describe your hypothesis and what it is based on, make falsifiable predictions, and report how you tested these predictions. *hint: what happens when you change the size? Performance counters can make a direct measurement, but can you validate your hypothesis with just timing measurements?*

3.5 larger spreads are slower, then faster again!

Now try this little bash script on the command line:

```
for((i=1;i<1000000;i*=2)); do ./cachestress -s 1048576 -i $i; done
```

It runs cachestress for a range of steps, where the execution time grows substantially up to some step size (4096 on my machines), then shrinks again eventually reaching the same speed as step size 1.

For this part, propose two falsifiable hypotheses. One hypothesis explaining why cachestress slows down as the step size increases. And a second hypothesis explaining how it speeds up again for even larger step sizes. For both hypotheses, describe an experiment and show an outcome that matches the prediction.

3.6 turn-in instructions

Similar to the previous homework, turn-in is by git classroom, using this invitation link `https://classroom.github.com/a/bVgdPARb`. Add a PDF with your answers, called hw2.pdf, to your template folder, commit, then push it to the turn-in repository as in hw1.

3.7 Double-check your submission

To make sure that you submitted everything you think you submitted (`git` can be a little tricky until you get used to it), *git clone* your turn-in repository into a fresh folder, check that your `report.pdf` is in the splash2 folder, and that RAYTRACE compiles as you would expect.