COMP40 Assignment: Code Improvement Through Profiling

There is no design document for this assignment.

Contents

1 Purpose and overview

The purpose of this assignment is to learn to use profiling tools to apply your knowledge of machine structure and assembly-language programming.

2 What we expect of you

Use code-tuning techniques to improve the performance of your choice of *one* of two programs:

- Your `ppmtrans` program doing 90-degree and 180-degree *blocked* rotations of an image like `/comp/40/images/large/mobo.jpg`. This image has 50 million pixels and will not fit in the cache.
- Your Universal Machine running the large "sandmark" benchmark. If you and your partner do not have a working Universal Machine between you, it will be acceptable to borrow a Universal Machine from another student, but *only* if you have already submitted what you have for the Universal Machine assignment.

The key parts of the assignment are to *identify bottlenecks using* `valgrind` and to *improve the code by increments*. You will therefore want to *do most of your profiling on small inputs, as long as they are sufficient to give usefully measurable execution times and to exercise the code paths of interest.*

Your grade will be based on three outcomes:

- Your *laboratory notes* about the *initial state* of your program and *each stage of improvement*, including *differences from the previous stage*.
- Your analysis of the assembly code of the most expensive procedure in your final program.
- The *performance of your final-stage program*, measured as follows:
  - For the image rotation, we will choose a large image and rotate it by both 90 degrees and 180 degrees. We will do the two rotations independently and sum the squares of the running times.
  - Your Universal Machine running a large benchmark, not identical to the sandmark but closely related to it.

2.1 Your starting point

Please begin with your code in the state it was after the array-rotation and Universal Machine assignments. If your code did not work, you may fix it, or you may start with the published array-rotation solution code and another student's Universal Machine. If you have not yet completed the UM, you may not look at another student's UM until you have submitted.

Please take baseline measurements of your code *as submitted*.

2.2 Tracking changes as you make them

During this assignment, you may run into a dead end that requires you to go back to a previous version of your code. We *recommend*, but do not require, that you use `git` to *keep track of each stage of improvement* in your code.

Your first source for all things `git` should be a fellow student who has learned `git` in one of Ming Chow's courses, or failing that, Ben Lynn's online tutorial *Git Magic*. One task you might need extra help with is if you want to go back to an older snapshot and create a *branch* starting from that snapshot.

If you do not choose to learn git then you should definitely find some other means of keeping the original versions of your code and compile scripts, and every important intermediate version as well.

2.3 Laboratory notes

Begin by *choosing a data set*. For image rotation, choose one large image (at least 25 megapixels) and three small images (about 100 thousand pixels each).[1] For the Universal Machine, you will use the small `midmark` benchmark, the large `sandmark` benchmark, and a partial solution to the adventure game.

For *either program*, at *each stage*, for *each input*, please

- Report the user-mode time required to execute the program on the input, as measured with the `time` command (for information, try `man 1 time` and see the examples below). Be aware that *the C shell has a built-in* `time` *command*, and it stupidly writes to standard output instead of standard error. If you are using the C shell, you will need to use `/usr/bin/time`.

  For `ppmtrans`, each input is an image, and "executing the program" means running *both* 90-degree and 180-degree *blocked* rotations on that image.

[1] If you do not have a large image, you can make a small image larger by using `pnmscale` with a scale factor greater than one.

For `um`, each input is a Universal Machine binary, and "executing the program" means running `um` on that binary, supplying a suitable test input on standard input.

- For small inputs, report the total number of instruction fetches, which you can measure by running the program under `valgrind --tool=callgrind`.
- Report *two different relative time values*:
  - The user-mode time of this stage divided by the user-mode time of the initial stage (time relative to start)
  - The user-mode time of this stage divided by the user-mode time of the *previous* stage (time relative to previous stage)

  Lower relative times are better.
- In clear, correct English, say what was the bottleneck from the previous stage and how you improved the code.

You can see some sample reports (using made-up data) in Table 1.

Note: Spring 2014 is the first year we are asking you to record *user-mode* time, which is (roughly) the time the CPU actually spent running your code. It does not include time spent running backround tasks or other programs that may be active on the same machine, and it does not include time spent waiting for disk (e.g. for page faults) or for network traffic. In previous years, this assignment asked for you to record the elapsed or *wall-clock* time. User-mode times tend to be more stable and repeatable, and are typically a better measure of the efficiency of the code. If you find any problems relating to this change, please report them to the instructor immediately.

When you change the code, it is critical that *each set of changes be small and isolated.* Otherwise you will not know what changes are responsible for what improvements.

1. Your *starting point* should be your code as submitted, compiled and linked with your original compile scripts.
2. Your *first change* should be to compile with `-O1` and to link with `-lcii40-O1`, which must come *before* other libraries.
3. Your *second change* should be to compile with `-O2` and to link with `-lcii40-O2`.
4. After that you can start profiling with `callgrind` and `kcachegrind` and improving your code based on the results. We'll be giving you practice with those tools in lab this week if you haven't already seen them.

Keep in mind that `-O2` is *usually* but not always better than `-O1` even though the GCC documentation suggests that it should be. The compiler tries to do more elaborate optimizations with `-O2`, but rarely those actually make things worse.

2.4 Analysis of assembly code

Once you have improved the code as much as you can, use `valgrind` and `kcachegrind` to find the single routine that takes the most time. (You can find it by clicking on the Self tab in `kcachegrind`.) For this