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.

NOTE: earlier of versions of this assignment gave you a choice of tuning `ppmtrans` or `um` and yet earlier versions asked you to tune both. We believe we have made the necessary changes to these instructions, but if you see stray references to image files, `ppmtrans`, etc. please do not be confused by them. Do report them so we can clean them up! Thank you.

2 What we expect of you

Use code-tuning techniques to improve the performance of your choice of your `um` emulator. Unless you have permission from the instructor, you must work with the same partner (if any) you had for HW6. If, with permission, you are working with a different partner, you may use either of your `um` emulators as a starting point. You will tune 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* `kcachegrind` 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*. This is important: improving your code is not the only goal; we also want to see that you have a computer scientist's approach to documenting the details that would allow someone else to understand the state of your sytem when you started, and why each of your changes had the effect that it did.
- 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: Your Universal Machine running a large benchmark, not identical to the sandmark but closely related to it. That is: for your testing, you will have the same version of sandmark you had while developing your `um`. We will benchmark your program running a test that is similar but not quite identical.

2.1 Your starting point

Please begin with your code in the state it was after the Universal Machine assignment. If your code did not work, you may fix it, or you may start with 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*.

If you fixed your UM before starting to tune it, also take baseline measurements using that version, and use those as the point of comparison for the rest of this assignment.

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.

Unfortunately, `git` is a sophisticated professional's tool, with many, many options you won't need. 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*. As a guide, you should be learning to `init` a repository, something you do once, and then to `add` files that have significant changes, and to label groups of additions by using `commit`. `git` will record every version of every file you have committed. One task you might need extra help with is if you want to go back to an older version of your files. There are several ways to do it. Look on the Web or ask for help.

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. One option is to make a new directory for each working version. Put into it a `README` that will remind you what is in the directory, what previous version it was based on, how well it worked, etc.

2.3 Laboratory notes

Begin by *choosing a data set*. For the Universal Machine, you will use the small `midmark` benchmark, the large `sandmark` benchmark, and a partial solution to the adventure game.

For *each stage* and 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`.

  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, for an early version of the assignment that focused on `ppmtrans`) 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.

Benchmark	*Time*	*Instructions*	*Rel to start*	*Rel to prev*	*Improvement*
big	30s	—	1.000	1.000	No improvement (starting point)
small	1s	75.02×10^6	1.000	1.000	
big	28s	—	0.933	0.933	Compiled with optimization turned on and linked against `-lcii-O1`
small	900ms	69.21×10^6	0.920	0.920	
big	28s	—	0.933	1.000	Compiled with optimization turned on and linked against `-lcii-O2`
small	900ms	69.18×10^6	0.920	1.000	
big	25s	—	0.833	0.893	Removed `Array_width()` call from `for` loop and placed result in local variable instead
small	833ms	62.01×10^6	0.833	0.926	
big	22s	—	0.733	0.880	Removed `array->blocks` expression from loop and placed result in local variable
small	800ms	56.16×10^6	0.800	0.960	
big	60s	—	2.000	2.727	Used explicit `for` loop instead of blocked-array mapping function. Time improved for small image but got worse for big image—undid change.
small	650ms	49.20×10^6	0.650	0.813	
big	18s	—	0.600	0.300	Changed representation of blocks so that access to elements within the blocked mapping function uses unsafe pointer arithmetic without bounds checking
small	600ms	44.89×10^6	0.600	0.923	

Table 1: Sample report for blocked image rotation (made-up data)

2.4 Analysis of assembly code

Once you have improved the code as much as you can, use `valgrind` and `kcachegrind` to find the part of your program that takes the most time. This may be a function or a loop or a `switch` statement, ... The answer to this question isn't "`main`": we want you to identify a program hot spot. (You can find it by clicking on the Self tab in `kcachegrind`.) For this final phase you may want to use the `--dump-instr=yes` option so you can see the assembly code in `kcachegrind`. The advantage of `kcachegrind` over `objump -d` is that it will tell you how many times each instruction was executed.

Once you've found the place where the most time is spent, examine the assembly code and either *identify specific ways (changes to to the assembly code itself) in which it could be improved* or *argue that there is not an obvious way to improve the assembly code*.

Things to look for include:

- Do you see opportunities to keep data in registers, where currently there are unnecessary memory accesses?
- Do you see unnecessary computation in loops?

Be alert for a *horrible* idiom in the Intel assembly language: the instruction

```
mov %esi, %esi
```

looks redundant, but it's not. This idiom stands for an instruction that zeroes out the most significant 32 bits of the 64-bit register `%rsi`. We on the COMP 40 staff think this is a bad bit of language design, but do watch out for it.

For this assignment, *there is no need to modify assembly code*.

2.5 Performance of the final stages

All profiling and measurements must be done on the machines in Halligan 116 or 118. There are two kinds of machines in Halligan labs: Intel Q6700 and Intel Core i7. You may work on either one, but you must be consistent or you will go mad! All the machines in 116 and 118 are suitable but since in 2016 the machines in the two rooms are different, you must not be mixing reports of measurements run in 116 with measurements run in 118. If you have trouble finding seats, you might make some progress by using another machine to build and test experimental versions, *being sure to keep all the internmediate versions of the source and compile scripts.* When the machine you need frees up, go back and measure and record all your intermediate results on that machine. Usually but not always, what makes an improvement on one machine will help on another. *That said, there is much less chance for confusion if you just choose a room and stick to it!*

To be sure you have the right kind of machine, run the command

```
grep 'model name' /proc/cpuinfo
```

and check whether the out put is:

```
model name      : Intel(R) Core(TM)2 Quad CPU    Q6700  @ 2.66GHz
model name      : Intel(R) Core(TM)2 Quad CPU    Q6700  @ 2.66GHz
model name      : Intel(R) Core(TM)2 Quad CPU    Q6700  @ 2.66GHz
model name      : Intel(R) Core(TM)2 Quad CPU    Q6700  @ 2.66GHz
```

or:

```
model name :      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
model name :      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
model name :      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
model name :      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
model name :      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
model name :      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
model name :      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
model name :      Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
```

Any other output means you have the wrong kind of machine and your time measurements will not be consistent.

Note that the machines in Halligan 120 run a different version of the Linux system and may have different compilers, libraries, etc. You can use those for debugging and learning the tools if necessary, but all timing measurements that contribute to your reported results should be made on machines from 116 or 118. **NEED TO CHECK, THE 120 MACHINES MAY HAVE BEEN FIXED FOR FALL 2016. We will announce any changes on Piazza.**

Measure your code with both `gcc -O1` and `gcc -O2`. Neither one is faster for all problems; report the better of the two results. In your final `compile` script, use whichever gives the best results.

You will be evaluated both on your improvement relative to the code you start with and on the absolute performance of your final results. Your laboratory notes will record all your improvements and the performance of your final stages.

2.6 What to submit

Please use the `submit40-profile` script to submit the following items.

1. A `compile` file, which when run with `sh`, compiles all your source code and produces a `um` binary.
2. A `README` file which
   - Identifies you and your programming partner by name
   - Acknowledges help you may have received from or collaborative work you may have undertaken with others
   - Explains what routine in the final `um` takes up the most time, and says whether the assembly code could be improved
   - Says approximately how many hours you have spent *analyzing the problems posed in the assignment*
   - Says approximately how many hours you have spent *solving the problems after your analysis*
3. A `labnotes.pdf` file that gives your laboratory notes in nice, readable format
4. All benchmarks you used as test data.

3 Methods of improving performance

In performance, really big wins usually come from better algorithms which provide superior asymptotic complexity. But for our programs, there is not much algorithmic action; everything should be pretty much

linear in the number of UM instructions executed. You may find some non-linearities relating to the patterns of access to segments, and those might be worth addressing. You can often improve things by *changing your data structures*.

Here are some trite but true observations:

- *Memory references are expensive*, especially when data is not in the cache. In fact, compared with memory references, arithmetic with values in registers is practically free. If you give `valgrind` the `--simulate-cache=yes` option, it will will count loads and stores and also simulate the cache. I don't see how to get the load/store data without also running an expensive cache simulation.

- In general you will be improving performance of your base implementation, but it's surprisingly easy to gain speed at the expense of excessive memory usage. *As with the original `UM` assignment, you are allowed only 1000 MB of memory. You will get no credit on any benchmark where you exceed that amount. For more information on how to test this, see the "Controlling Use of CPU and Memory" section of the original UM instructions.*

- On AMD64, *calls to leaf procedures* are pretty cheap, but *calls to non-leaf procedures* can be expensive.

What if your program is nothing *but* memory references and procedure calls?! How can you make progress?

- To know what to improve, *you must profile*. Measure, measure, and measure again. Your best friends are `valgrind --tool=callgrind` and the `kcachegrind` visualizer.[1]

  Nothing is more frustrating than to spend a lot of time improving code that is rarely executed.

- The C compiler can be stupid about memory references. Because of pointer aliasing, if you write to memory, the C compiler may assume that *all* values in memory have changed, and may have to be reloaded.

- The C compiler has no idea when multiple calls to a function will return the same value. If you do have an idea, you can help out the C compiler by putting results in local variables.

- The C compiler has to assume that a function call could scribble all over memory. After a function call, values referenced through pointers may have changed. If *you* know the values haven't changed, make sure those value are sitting in local varaibles, so that the compiler knows it too.

- The C compiler is *staggeringly good* at managing local variables and putting them in machine registers. All you have to do is get your values into local variables; the compiler will do the rest. This is a big change from the 1970s!

- If a lot of time is spent in one procedure, like say `UArray_at`, you often have two choices: make each call of the procedure run faster, or change code somewhere else so the procedure is called less often.

There are some external sources you might find useful.

[1] For some programmers in some cases, `gprof` can be a pretty good friend, but it is useful only if you have access to all the source code, including libraries. And gprof does not have a good visualization tool like `kcachegrind`. In fact, the damn thing won't even report all the data it has because it uses only two digits after the decimal point. Beastly `gprof` is not *my* friend.

- At `http://www.stevemcconnell.com/cctune.htm`, Steve McConnell has a book excerpt which despite being 15 years old is still quite good on the subject of code tuning. The table at the end is similar to what I want from you, except I want you to include all the false starts he leaves out.

  Steve's spelling could use some work, don't you think?

- Although Don Knuth invented the field, when it comes to explaining how to make programs efficient, Jon Bentley is the dean of authors. Jon has summarized some of his work at `http://www.cs.bell-labs.com/cm/cs/pearls/apprules.html`. Unfortunately, improvements in optimizing compilers have rendered many of Jon's suggestions obsolete. (Between 1982 and 1993, compilers got a *lot* better. Between 1993 and today, not so much.)

- Your book by O'Hallaron and Bryant devotes an entire chapter to code improvement (Chapter 5). There's about 15 pages' worth of really good low-hanging fruit, and then there are a lot of details.

  - The first part, through the end of Section 5.1, gives an excellent explanation of aliasing and will help you understand the pessimism with which the compiler must treat memory references.
  - Sections 5.2 and 5.3 present a basic framework and example. If you like toy benchmark programs and graphs with lines on them, these sections are for you.
  - Sections 5.4 to 5.6, which comprise only ten pages, give more detailed explanations of the most important of the techniques I've sketched above.
  - Section 5.7 tells a complicated story that explains some of the complexities of modern processors. However, the focus on the Intel Core i7 may not be helpful for understanding the behavior of the Core 2 Quad processors on the lab machines, as there are some differences.
  - Sections 5.13 and 5.14 discuss the use of a profiler. I hope you will find the class demo more informative, but the class demo was brief, and this is the place to go for additional explanation and background. Unfortunately the chapter refers to `gprof`, which is a legacy tool that I recommend against using unless you are stuck with a problem for which `valgrind` is just too slow.
  - Sections 5.8 to 5.10 describe program transformations which, for the most part, a good optimizing compiler can do better than you can.
  - Section 5.15 summarizes material in earlier sections. Perhaps you will find the summaries useful for review?
  - Sections 5.11 and 5.12 present material that I consider interesting and important but well beyond the scope of COMP 40. This material is more likely to be taught in a 100-level architecture course aimed at juniors, seniors, and beginning graduate students.

4 Partial solution to the adventure game

Figure 1 gives a a partial solution to the adventure game. This solution can be made into a benchmark that is intermediate in difficulty between the midmark and the sandmark.

```
n
take bolt
take spring
inc spring
take button
take processor
take pill
inc pill
take radio
take cache
comb processor cache
take blue transistor
comb radio transistor
take antenna
inc antenna
take screw
take motherboard
comb motherboard screw
take A-1920-IXB
comb A-1920-IXB bolt
comb A-1920-IXB processor
comb A-1920-IXB radio
take transistor
comb A-1920-IXB transistor
comb motherboard A-1920-IXB
take keypad
comb keypad motherboard
comb keypad button
s
```

Figure 1: Partial solution to the adventure game

5 Secrets of the shell-programming masters

This section survives from an earlier version of this assignment in which students tuned `ppmtrans`. You will likely not need the following for your work with `um`, but there are some interesting programming tricks here, so I'm leaving it for your reference. You may find this interesting anyway.

Here are some *untested* ideas for automatically copying files to `/data` or `/tmp` lazily, on demand. They should work with `#!/bin/bash` or `#!/bin/ksh`. The ideas are:

- You create a subdirectory `/data/$USER`.
- Your files live there.
- A file is copied (by function `in_data`) only if there's not a copy there already.

First, shell function `datafile` names a file in a subdirectory of `/data` that belongs just to you, like `/data/nr`:

```
function datafile {      # pathname of a personal file in /data
  echo "/data/$USER/$1"
}
```

Second, shell function `in_data` takes an arbitrary file and copies it to your `/data` directory. Copying is done only if a file of the same name is not already present:

```
function in_data {   # possibly copy a file to /data;return its path there
  # set -x # uncomment me to see commands execute
  typeset data="$(datafile "$(basename $1)")"
  if [[ ! -r "$data" ]]; then # file is not already in /data
    mkdir -p "$(dirname "$data")"  # make the directory
```

```
    cp -v "$1" "$data"             # copy the file
  fi
  echo "$data"                     # print the new location
}
```

You can now use this function routinely to make sure that every "big input" is in `/data`:

```
# set -x # uncomment me to see commands execute
for i in biginput1 biginput2 biginput3
do
  /usr/bin/time -f ... my_program $(in_data $i)
done
```