

# COMP40 Assignment: Code Improvement Through Profiling

Assignment due Tuesday, November 22 at 11:59 PM. There is no design document for this assignment.

## Contents

<b>1</b>	<b>Purpose and overview</b>	<b>1</b>
<b>2</b>	<b>What we expect of you</b>	<b>1</b>
2.1	Your starting point . . . . .	2
2.2	Tracking changes as you make them . . . . .	2
2.3	Laboratory notes . . . . .	3
2.4	Analysis of assembly code . . . . .	4
2.5	Performance of the final stages . . . . .	5
2.6	What to submit . . . . .	6
<b>3</b>	<b>Methods of improving performance</b>	<b>7</b>
<b>4</b>	<b>Partial solution to the adventure game</b>	<b>9</b>
<b>5</b>	<b>Secrets of the shell-programming masters</b>	<b>9</b>

## 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. You will improve the performance of two programs.

## 2 What we expect of you

Use code-tuning techniques to improve the performance 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. If you and your partner do not have a working version between you, you may use one of Professor Ramsey's solutions.
- 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.

If you wish, you may make special arrangements with Professor Ramsey to replace one of these two programs with *Song Search* from COMP 15.

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*.

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 each of your final programs.
- The *performance of your final-stage programs*, 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 Professor Ramsey's array-rotation 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*. (If you have already changed your Universal Machine, don't worry; your CS department account should have received an email containing your UM 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.

- Initialize `git` by running

```
git init
```

in the directory that contains your source code.

- Take a snapshot of your initial code by running

```
git commit *.c *.h
```

- Every time you profile a stage, take another snapshot by running

```
git commit *.c *.h
```

- When you have something for your laboratory notes, “tag” the snapshot with a meaningful name, as in this example:

```
git commit *.c *.h
git tag replaced-segment-map-with-turtles
```

- Although the interface is a bit busy, Professor Ramsey likes the graphical commit tool

```
git gui
```

for seeing what has changed since the last snapshot. It may help you with your notes.

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.

## 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).<sup>1</sup> 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 program*, at *each stage*, for *each input*, please

- Report the wall-clock 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.

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 wall-clock time of this stage divided by the wall-clock time of the initial stage (time relative to start)
  - The wall-clock time of this stage divided by the wall-clock 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.

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.

---

<sup>1</sup>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.

<i>Benchmark</i>	<i>Time</i>	<i>Instructions</i>	<i>Rel to start</i>	<i>Rel to prev</i>	<i>Improvement</i>
big	30s	—	1.000	1.000	No improvement (starting point)
small	1s	$75.02 \times 10^6$	1.000	1.000	
big	28s	—	0.933	0.933	Compiled with optimization turned on and linked against -lcii-01
small	900ms	$69.21 \times 10^6$	0.920	0.920	
big	28s	—	0.933	1.000	Compiled with optimization turned on and linked against -lcii-02
small	900ms	$69.18 \times 10^6$	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	$62.01 \times 10^6$	0.833	0.926	
big	22s	—	0.733	0.880	Removed array->blocks expression from loop and placed result in local variable
small	800ms	$56.16 \times 10^6$	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	$49.20 \times 10^6$	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	$44.89 \times 10^6$	0.600	0.923	

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

2. Your *first change* should be to compile with -O1 and to link with -lcii40-01, which must come *before* other libraries.
3. Your *second change* should be to compile with -O2 and to link with -lcii40-02.
4. After that you can start profiling with callgrind and kcache-grind and improving your code based on the results.

Keep in mind that -O1 is *not* always better than -O2.

## 2.4 Analysis of assembly code

Once you have improved the code as much as you can, use valgrind and kcache-grind to find the single routine that takes the most time. (You can find it by clicking on the Self tab in kcache-grind.) 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 objdump -d is that it will tell you how many times each instruction was executed.

Once you've found the routine in which the most time is spent, examine the assembly code and either *identify specific ways (changes 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*.

Do this exercise for both ppmtrans and um binaries.

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. Whoever allowed this syntax into the assembler should be shot.

Do this exercise for both ppmtrans and um.

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 Intel Q6700 machines in Halligan 116 or 118. Almost all the machines in 116 and 118 are suitable; to be sure you have the right kind of machine, run the command*

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

and make sure the output 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
```

Any other output means you have the wrong kind of machine and your time measurements will not be consistent. In particular, The machines in 120 are *not* suitable for this assignment.

*The importance of using the lab machines cannot be overstated.*

- For consistency, you absolutely must use these machines for profiling and measurements—if necessary, log into the machines remotely.
- NR made what he thought was a minor change to his Universal Machine, intended to improve modularity. On NR's home computer, the new code was 25% slower. On the linux.cs.tufts.edu server, the new code was about the same speed. On the machines in Halligan 118, the new code was about *twice as fast*.

The machine you're using matters.

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. This means it is easier for you to get top marks if you start with your own code rather than NR's, since his has less room to be improved. 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 both `ppmtrans` and `um` binaries.
2. A `run` file, which when run with `sh` runs all your test cases. For accurate performance measurements, large inputs should be copied to `/data` or `/tmp`, so they reside on a local disk. (See Section 5.) Here's a sample for `ppmtrans`; you will want to change it to suit your own image files:

```
#!/bin/sh
. /usr/sup/use/use.sh
use comp40
img='tempfile --suffix=.ppm'

djpeg big.jpg > $img
time -f "large rotate 90: %e seconds" ./ppmtrans -block-major -rotate 90 $img > /dev/null
time -f "large rotate 180: %e seconds" ./ppmtrans -block-major -rotate 180 $img > /dev/null

for i in small1.jpg small2.jpg small3.jpg
do
    djpeg $i > $img
    time -f "rot $i 90 deg: %e seconds" ./ppmtrans -block-major -rotate 90 $img > /dev/null
    time -f "rot $i 180 deg: %e seconds" ./ppmtrans -block-major -rotate 180 $img > /dev/null
done
```

(Because we are using a sane, sensible shell, we can use `time` instead of `/usr/bin/time`.)

And here is an example for the `UM`:

```
#!/bin/sh
. /usr/sup/use/use.sh
use comp40

for i in midmark.um sandmark.umz
do
    time -f "um $i: %e seconds" um $i > /dev/null
done
```

3. 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 ppmtrans takes up the most time, and says whether the assembly code could be improved
- 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*

4. A labnotes.pdf file that gives your laboratory notes in nice, readable format

5. All images and benchmarks you used as test data, preferably in a compressed format like jpg or png

### 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 either the number of pixels (images) or the number of UM instructions executed (Universal Machine). 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 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.
- 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.<sup>2</sup>

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.

---

<sup>2</sup>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.

- 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 variables, 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.

- 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://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 will be brief, and this is the place to go for additional explanation and background—or to chase squirrels. 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.



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

Figure 1: Partial solution to the adventure game

- 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 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.

## 5 Secrets of the shell-programming masters

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
```