COMP40 Assignment: Code Improvement Through Profiling

Assignment due Thursday, December 3 at 11:59 PM. There is no design document for this assignment.

Contents

1 Purpose and overview .......................... 1

2 What we expect of you .......................... 1
  2.1 Laboratory notes ................................ 1
  2.2 Analysis of assembly code ...................... 2
  2.3 Performance of the final stages ............... 3
  2.4 What to submit ................................ 3

3 Methods of improving performance .......... 4

4 Partial solution to the adventure game ...... 6

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:

- You ppmtrans or ppmtrans-block program doing 90-degree and 180-degree rotations of an image
  like /comp/40/images/large/mobo.jpg. This image has 50 million pixels and will not fit in the
  cache. You may use blocked or unblocked versions. If you and your partner do not have a working
  version between you, you may use one of my 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.

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.
- 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 de-
    grees. 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 my array-rotation code and another student’s Universal Machine.

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 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). For the Universal Machine, you will use the small midmark benchmark, the large sandmark benchmark, and a partial solution to the adventure game.

You have three operations: rotate 90, rotate 180, and run um.

At each stage, for each input, please

• Report the wall-clock time required to execute the program, 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 small inputs, report the total number of instruction fetches measured by valgrind --tool=callgrind.

• Report the relative running time, i.e., the wall-clock time of this stage divided by the wall-clock time of the initial stage. Lower relative times are better.

• In informal English, say what was the bottleneck from the previous stage and how you improved the code.

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 -lcii -O1, which must come before other libraries.

3. Your second change should be to compile with -O2 and to link with -lcii -O2.

4. After that you can start improving the code.

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

---

1 If you do not have a large image, you can make a small image larger by using pnm scale with a scale factor greater than one.
Here is sketch with some made-up examples:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
<th>Instructions</th>
<th>Relative</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>big</td>
<td>30s</td>
<td>—</td>
<td>1.000</td>
<td>No improvement (starting point)</td>
</tr>
<tr>
<td>small</td>
<td>1s</td>
<td>$75 \times 10^6$</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>28s</td>
<td>—</td>
<td>0.933</td>
<td>Compiled with optimization turned on and linked against -lcii-O1</td>
</tr>
<tr>
<td>small</td>
<td>900ms</td>
<td>$69 \times 10^6$</td>
<td>0.920</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>28s</td>
<td>—</td>
<td>0.933</td>
<td>Compiled with optimization turned on and linked against -lcii-O2</td>
</tr>
<tr>
<td>small</td>
<td>900ms</td>
<td>$69 \times 10^6$</td>
<td>0.920</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>25s</td>
<td>—</td>
<td>0.833</td>
<td>Removed <code>Array_width()</code> call from for loop and placed result in local variable instead</td>
</tr>
<tr>
<td>small</td>
<td>833ms</td>
<td>$62 \times 10^6$</td>
<td>0.833</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>22s</td>
<td>—</td>
<td>0.733</td>
<td>Removed <code>array-&gt;blocks</code> expression from loop and placed result in local variable</td>
</tr>
<tr>
<td>small</td>
<td>800ms</td>
<td>$56 \times 10^6$</td>
<td>0.800</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>60s</td>
<td>—</td>
<td>2.000</td>
<td>Used explicit for loop instead of blocked-array mapping function. Time improved for small image but got worse for big image—undid change.</td>
</tr>
<tr>
<td>small</td>
<td>650ms</td>
<td>$49 \times 10^6$</td>
<td>0.650</td>
<td></td>
</tr>
<tr>
<td>big</td>
<td>18s</td>
<td>—</td>
<td>0.600</td>
<td>Changed representation of blocks so that access to elements within the blocked mapping function uses unsafe pointer arithmetic without bounds checking</td>
</tr>
<tr>
<td>small</td>
<td>600ms</td>
<td>$45 \times 10^6$</td>
<td>0.600</td>
<td></td>
</tr>
</tbody>
</table>

2.3 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 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 ways in which it could be improved or argue that there is not an obvious way to improve it.

Do this exercise for both `ppmtrans` and `um` binaries.

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

2.4 Performance of the final stages

All measurements will be taken on the machines in Halligan 118. This means you should do your own profiling and measurements on those machines, by remote login if necessary. The importance of using the lab machines cannot be overstated. I recently made what I thought was a minor change to my Universal Machine, intended to improve modularity. On my computer at home, 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 -01` and `gcc -02`. 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 mine, since mine has less room to be improved. Your laboratory notes will record all your improvements and the performance of your final stages.

### 2.5 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. Please discard any copies you may have of `pnmhdr.c` or `pnmhdr.o`. The early versions of this code were extremely inefficient. I have replaced them with more efficient versions which you can get by linking with `-L/comp/40/lib64 -lpnm -lnetpbm`.

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. Here's a sample; you will want to change it to suit your own image files:

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

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

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

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

rm -f $img
```

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 each final program 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, however, 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.

- 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.2

  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 also 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.

2For 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. Unquestionably gprof is not my friend.
• 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 `Array_get`, 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](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](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 5.5 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 probably still contains some grains of truth but is now almost irrelevant to today’s processors. The situation is not your authors’ fault; architects get more transistors every year, and they do things that programmers don’t understand. The last time an architect gave me a straight answer to the question “what do I need to do to ensure good performance of my machine code” was 1999. And the answer wasn’t very helpful.
  
  – Section 5.15 discusses the use of a profiler. I hope you will find the class demo more informative, but this is the place to go for additional explanation or 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.
  
  – Sections 5.11, 5.14, and 5.16 summarize material in earlier sections. Perhaps you will find the summaries useful for review?
  
  – Sections 5.12 and 5.13 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

This partial solution to the adventure game 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
take 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
take keypad motherboard
comb keypad button
s