Contents Lecture 10

- Writing Correct C Code
- Writing Fast and Correct C Code
- Application of the methodology to writing small C code
After correctness and maintainability, speed and/or code size are usually very important qualities of C code.

If speed is not essential, it might be better to use another language such as Java or Scala, for instance if your program should run on an Android device.

Don’t optimize anything before you have a correct program.

The reference implementation should follow the specification for your code in an obviously correct way — almost no matter how slow — within practical constraints of course.
While the reference implementation almost always should be written in the same language as the final implementation, it might be a good idea to use an existing tool or language with available libraries which already (or almost) solve the problem (but not sufficiently fast).

Therefore: use whatever tool or language you think is easiest to make a first correct version with.

For instance, C++, Java, Mathematica, Matlab, Maple, Scala or something else might be easiest to use.

Of course, if you are extending an existing program you probably need to use the same language, though.

Keep a copy of the reference implementation for testing. It’s invaluable.

Then, if you didn’t already use C as implementation language, do that.

Thus, when you have a simple-to-understand and correct reference implementation, you want to write a faster version in C.
Uses of the Reference Implementation

- Testing involves validating your fast versions against the reference implementation.
- It may also include proving there are bugs in any other versions made by others — that is, if there are other versions and if your and their fast versions do not produce the same output.
- For example, if you want to write a clock-cycle true simulator of a complex superscalar microprocessor, having a simple simulator which does not model any pipeline or cache memory will be invaluable to validate the complex model — or finding the first instruction with wrong result!
Maintain correctness using the reference implementation.

Writing a faster version is what we call **implementation tuning**, which we define as:

**Definition**

**Implementation tuning** is the manual application of code transformation techniques which current state-of-the-art optimizing compilers are not capable of doing automatically.
Improving the Performance of a Correct C/C++ Program

1. If performance is good enough then go on vacation.
2. Profile your program using different tools.
3. Figure out how you can improve the most time consuming part.
   - Should you use a different optimizing compiler or other optimization flags?
   - Should you use a different algorithm and/or data structure?
   - Can you exploit something in the input to make the common case faster?
   - Can you precompute or cache values?
   - Is it possible to use mathematics to simplify the program?
   - Can you use counters to collect statistics about the behaviour of your program — if the profilers do not give you sufficient insights?
   - How can you exploit the behaviour you have detected?
4. Implement your ideas and make measurements to verify that your ideas are correct.
5. Validate your program on all test cases.
6. Go to 1.

Jonas Skeppstedt (jonasskeppstedt.net)
Profiling Tools

- **OProfile** — samples the program counter and hardware counters
- **Gprof** — also samples the program counter and analyses the call graph
- **Gcov** — counts number of times each line is executed
- All these tools are explained in the book.
- Don’t forget: `printf("counter X = %llu\n", x);`
- Simple counters can give a lot of insights — it’s usually a good idea to use `unsigned long long` for counters — otherwise you might print out nonsense, if your counters overflow.
Oprofile is a so called daemon which must be started by the super user in Linux.

On your own machine at home, give the following commands as root:
- opcontrol --no-vmlinux
- opcontrol --start

On power.cs.lth.se that has already been done and you neither can nor need to do it on that machine.

Suppose we want to profile the program 300.twolf, which is a CAD tool, from the SPEC CPU2000 benchmark suite.

SPEC = Standard Performance Evaluation Corporation is a non-profit organization which develops different benchmarks (www.spec.org).

We compile 300.twolf with -g and -O2 flags and run it as normal.

Oprofile collects statistics on all running programs and we don’t have to tell it to profile a particular program — we just run it.
There are two commands to get information from the collected statistics:
- `opreport` and `opannotate`

After having run the program, whose executable file is called `twolf_base.gcc` we give the command:
```
opreport -t 1 -l -g twolf_base.gcc
```

- The `-t 1` tells `opreport` to only report functions which take more than one percent of the execution time.
- The `-l` tells `opreport` to list function names.
- The `-g` tells `opreport` to print debug info, namely source file and line numbers.
<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>linenr</th>
<th>info</th>
<th>symbol</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>59620876</td>
<td>27.8859</td>
<td>118</td>
<td>dimbox.c</td>
<td>new_dbox_a</td>
<td></td>
</tr>
<tr>
<td>28968605</td>
<td>13.5492</td>
<td>82</td>
<td>uloop.c</td>
<td>uloop</td>
<td></td>
</tr>
<tr>
<td>27464843</td>
<td>12.8459</td>
<td>14</td>
<td>ucxx2.c</td>
<td>ucxx2</td>
<td></td>
</tr>
<tr>
<td>20427816</td>
<td>9.5545</td>
<td>211</td>
<td>dimbox.c</td>
<td>term_newpos_a</td>
<td></td>
</tr>
<tr>
<td>19016543</td>
<td>8.8944</td>
<td>82</td>
<td>dimbox.c</td>
<td>new_dbox</td>
<td></td>
</tr>
<tr>
<td>17456727</td>
<td>8.1649</td>
<td>254</td>
<td>dimbox.c</td>
<td>term_newpos_b</td>
<td></td>
</tr>
<tr>
<td>6011636</td>
<td>2.8118</td>
<td>181</td>
<td>dimbox.c</td>
<td>term_newpos</td>
<td></td>
</tr>
<tr>
<td>5792645</td>
<td>2.7093</td>
<td>10</td>
<td>xpickint.c</td>
<td>XPICK_INT</td>
<td></td>
</tr>
<tr>
<td>4336363</td>
<td>2.0282</td>
<td>178</td>
<td>clean.c</td>
<td>add_penal</td>
<td></td>
</tr>
<tr>
<td>3824800</td>
<td>1.7889</td>
<td>11</td>
<td>acceptt.c</td>
<td>acceptt</td>
<td></td>
</tr>
<tr>
<td>3802342</td>
<td>1.7784</td>
<td>81</td>
<td>clean.c</td>
<td>old_assgnto_new2</td>
<td></td>
</tr>
<tr>
<td>3698904</td>
<td>1.7301</td>
<td>146</td>
<td>clean.c</td>
<td>sub_penal</td>
<td></td>
</tr>
<tr>
<td>3219648</td>
<td>1.5059</td>
<td>40</td>
<td>rand.c</td>
<td>Yacm_random</td>
<td></td>
</tr>
</tbody>
</table>
With the command opannotate -s twolf_base.gcc

Number of samples for a source code line reflects its execution time.

The mapping from instruction address to source code line cannot be perfect due to different optimization including instruction scheduling made by the compiler.
Adding the flag `-a` gives also assembler code:

```assembly
:       if( rowsptr[row] == 0 ) {
5394458 2.5205 :1000b848:        lbz    r10,0(r11)
      :1000b84c:        cmpwi  cr7,r10,0
      :1000b850:        beq-   cr7,1000b834 <new_dbox_a+0x1ac>
```
Effects of Recompilation

- The samples of all executable files are collected in the directory:
  
  `/var/lib/oprofile/samples/current/{root}`

- Our executable file for 300.twolf is `twolf_base.gcc` and is located in the real directory:
  
  `/opt/spec/cpu2000/benchspec/CINT2000/300.twolf/run/00000002`

- So the samples of the file are put in a directory in the ”shadow” directory:
  
  `/var/lib/oprofile/samples/current/{root}/opt/spec/cpu2000/benchmark/CINT2000/300.twolf/run/00000002`

- If our program is recompiled after collecting the samples, the executable file will **not** match the samples and no profiling info can be printed.

- Therefore, if we need to recompile, we should give the executable file a different name!
More Profiling Issues

- If you don’t get any samples collected, try making the program execute for a long time.
- Note that Oprofile does not need any special compilation flag (except -g to make the output more useful) so your program will have no special instrumentation code.
- The overhead of sampling typically slows down the machine by less than one percent.
Gprof was developed by UC Berkeley and described in an article published 1982.

Gprof stands for graphical profiler.

Gprof requires the compilation flag -pg

Gprof prints how long time is spent in each function just as Oprofile.

In addition to that, however, Gprof also shows how many times each function is called by different functions.
The Flat Profile made by Gprof

- Here we can see the time in each function and the number of times they were called, as well as the time in each function invocation (self) and including called functions (total).
- The numbers are quite similar to what Oprofile found.

Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>self seconds</th>
<th>total time</th>
<th>total seconds</th>
<th>calls</th>
<th>s/call</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26.29</td>
<td>67.07</td>
<td>67.07</td>
<td>118824440</td>
<td>0.00</td>
<td>0.00</td>
<td>new_dbox_a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.67</td>
<td>101.94</td>
<td>34.87</td>
<td>120</td>
<td>0.29</td>
<td>2.09</td>
<td>uloop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.10</td>
<td>135.35</td>
<td>33.41</td>
<td>97434069</td>
<td>0.00</td>
<td>0.00</td>
<td>ucxx2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.01</td>
<td>158.32</td>
<td>22.97</td>
<td>65546055</td>
<td>0.00</td>
<td>0.00</td>
<td>new_dbox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.32</td>
<td>179.53</td>
<td>21.21</td>
<td>62176704</td>
<td>0.00</td>
<td>0.00</td>
<td>term_newpos_a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.21</td>
<td>197.93</td>
<td>18.40</td>
<td>56647736</td>
<td>0.00</td>
<td>0.00</td>
<td>term_newpos_b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.88</td>
<td>207.83</td>
<td>9.90</td>
<td>103242054</td>
<td>0.00</td>
<td>0.00</td>
<td>accepttt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>215.89</td>
<td>8.06</td>
<td>65546055</td>
<td>0.00</td>
<td>0.00</td>
<td>term_newpos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.43</td>
<td>222.08</td>
<td>6.19</td>
<td>446826298</td>
<td>0.00</td>
<td>0.00</td>
<td>Yacm_random</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.40</td>
<td>228.20</td>
<td>6.12</td>
<td>229846431</td>
<td>0.00</td>
<td>0.00</td>
<td>XPICK_INT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.94</td>
<td>233.14</td>
<td>4.94</td>
<td>97434069</td>
<td>0.00</td>
<td>0.00</td>
<td>old_assgnto_new2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.86</td>
<td>237.89</td>
<td>4.75</td>
<td>203003477</td>
<td>0.00</td>
<td>0.00</td>
<td>sub_penal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.85</td>
<td>242.62</td>
<td>4.73</td>
<td>203003477</td>
<td>0.00</td>
<td>0.00</td>
<td>add_penal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.41</td>
<td>246.21</td>
<td>3.59</td>
<td></td>
<td></td>
<td></td>
<td>call___do_global_ctors_aux</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Call Graph made by Gprof

- Above ucxx2 we see which two functions have called ucxx2 and how many times.
- Below ucxx2 we see which functions it has called.
- For example, new_dbox_a was called 113295472 times by ucxx2 out of a total of 118824440 times.

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>0.22</td>
<td>139002/97434069</td>
<td>ucxx2 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33.36</td>
<td>152.74</td>
<td>97295067/97434069</td>
<td>uloop [3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63.95</td>
<td>0.00</td>
<td>113295472/118824440</td>
<td>new_dbox_a [5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.32</td>
<td>0.00</td>
<td>56647736/62176704</td>
<td>term_newpos_a [7]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.40</td>
<td>0.00</td>
<td>56647736/56647736</td>
<td>term_newpos_b [8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.13</td>
<td>0.00</td>
<td>48879688/65546055</td>
<td>new_dbox [6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.78</td>
<td>0.90</td>
<td>81087580/103242054</td>
<td>acceptt [9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.01</td>
<td>0.00</td>
<td>48879688/65546055</td>
<td>term_newpos [13]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.94</td>
<td>0.00</td>
<td>97434069/97434069</td>
<td>old_assgnto_new2 [15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.56</td>
<td>0.00</td>
<td>194868138/203003477</td>
<td>sub_penal [16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.54</td>
<td>0.00</td>
<td>194868138/203003477</td>
<td>add_penal [17]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.49</td>
<td>0.00</td>
<td>9546996/10319822</td>
<td>dbox_pos_2 [19]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.40</td>
<td>0.00</td>
<td>97434069/105569408</td>
<td>new_old [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.13</td>
<td>0.00</td>
<td>97434069/97434069</td>
<td>find_new_pos [21]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.60</td>
<td>0.00</td>
<td>19364958/19364958</td>
<td>new_assgnto_old2 [23]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.54</td>
<td>0.00</td>
<td>29182920/40860532</td>
<td>dbox_pos [22]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.26</td>
<td>0.00</td>
<td>38729916/41026101</td>
<td>add_cell [26]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jonas Skeppstedt (jonasskeppstedt.net)  Lecture 10  2019  17 / 26
Gcov

- Gcov is the GNU version of the UNIX tool for doing test coverage analysis, called tcov.
- It shows how many times each line has been executed, which is important during testing: you haven’t tested your code if it was not executed...
- So this tool can indicate to you that new test cases must be created.
- It also can give you very valuable insights about the behaviour of your program.
- Since we must make the common case fast, we need to know which case is the common and that can often be deduced with gcov.
- Compile with gcc -fprofile-arcs -ftest-coverage
460540733: 118: for( termptr = antrmptr ; termptr ; termptr = termptr->nextterm ) {
341716293: 119:      net = termptr->net ;
341716293: 120:      dimptr = netarray[ net ] ;
341716293: 121:      if( dimptr->dflag == 0 ) {
8951527: 122:          continue ;
-: 123:      }
332764766: 124:      dimptr->dflag = 0 ;
-: 125:  
332764766: 126:      new_mean = dimptr->new_total / dimptr->numpins ;
332764766: 127:      old_mean = dimptr->old_total / dimptr->numpins ;
1664750996: 128:      for( netptr = dimptr->netptr ; netptr ; netptr = netptr->nterm ) {
1331986230: 129:          oldx = netptr->xpos ;
1331986230: 130:          if( netptr->flag == 1 ) {
341716293: 131:              newx = netptr->newx ;
341716293: 132:              netptr->flag = 0 ;
-: 133:          } else {
990269937: 134:              newx = oldx ;
-: 135:          }
1331986230: 136:          *costptr += ABS( newx - new_mean ) - ABS( oldx - old_mean ) ;
-: 137:      }
The code below tests for the most unlikely condition first!

How can we improve the loop?

```c
int c;

while ((c = getchar()) != EOF)
    if (c == 'n')
        X;
    else if (c == ' ')
        Y;
    else
        Z;
```
Test for the most likely path first!

int c;

for (;;) {
    c = getchar();
    if (c > ' ')
        Z;
    else if (c == ' ')
        Y;
    else if (c == '
')
        X;
    else if (c == EOF)
        break;
    else
        Z;
}
Assume $Z$ is so large we don’t want to duplicate it!

```c
int c;

for (;;) {
    c = getchar();
    if (c > ' ')
        L: Z;
    else if (c == ' ')
        Y;
    else if (c == '
')
        X;
    else if (c == EOF)
        break;
    else
        goto L;
}
```
Run with: `valgrind --tool=cachegrind ./a.out`

```plaintext
a6.c: T = 53.38 s
==1413==
==1413== I  refs: 1,826,470,645
==1413== I1 misses: 1,069
==1413== L2i misses: 1,059
==1413== I1 miss rate: 0.00%
==1413== L2i miss rate: 0.00%
==1413==
==1413== D  refs: 406,017,535 (405,215,452 rd + 802,083 wr)
==1413== D1 misses: 111,833,593 (111,826,864 rd + 6,729 wr)
==1413== L2d misses: 67,481,024 ( 67,474,321 rd + 6,703 wr)
==1413== D1 miss rate: 27.5% ( 27.5% + 0.8% )
==1413== L2d miss rate: 16.6% ( 16.6% + 0.8% )
==1413==
==1413== L2 refs: 111,834,662 (111,827,933 rd + 6,729 wr)
==1413== L2 misses: 67,482,083 ( 67,475,380 rd + 6,703 wr)
==1413== L2 miss rate: 3.0% ( 3.0% + 0.8% )
```
First write a correct version.

Then check if your program is sufficiently fast.

If it’s not, you need to understand why it isn’t and figure out how to improve the situation.

Using profilers give you insights into the execution of your program.

This method is applicable to the problem of making small code as well.

Measure the size and study the assembler code to see if/how you can improve it.
Profilers do not always provide "perfect" information.

For instance, Cachegrind gives cache miss rates but does not tell you why there were misses.

Some simulators can see which variables map to the same place in the cache and tell you that.

Then you probably can fix that by moving one of the variables.

Simulators are of course much slower than real machines.

However, since they can count clock cycles exactly (or at least instructions) you often don’t have to run your benchmark for so long.
For example: suppose you have a short function and you want to understand exactly what happens in it.

Instead of sampling the PC during 20 seconds or five minutes, you can run the simulator once and it almost directly can tell you the function takes 44 clock cycles to execute.

More importantly, it can visualize what happens in the pipeline so you understand exactly why it takes 44 cycles.

Therefore simulators actually can be quicker — but usually they are not.