Contents Lecture 10

- Writing Correct C Code
- Writing Fast and Correct C Code

Writing Correct C Code

- After **correctness** and **maintainability**, **speed** and/or **code size** are usually very important qualities of C code.
- ***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.

Making the Reference Implementation

- 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!

Writing Correct and Fast C Code

- 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.

Profiling Tools

- operf — 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.
- Or better: `typedef unsigned long long counter_t`

Performance monitor counters

- Special registers in the CPU
- For instance four such registers
- They can be programmed to count events
- Some of the 960 events that can be counted on POWER8:
  - PM_DATA_FROM_MEM
  - PM_L1_ICACHE_MISS
  - PM_RUN_INST_CMPL
  - PM_TM_TBEGIN
  - PM_CYC (clock cycles)
- After a selected number of events have occurred, an exception is triggered in the CPU.
- This changes the CPU state to supervisor mode, saves the PC, and jumps to the operating system kernel.
- The kernel collects the statistics (event type and PC) and returns to user mode and resumes the program.

operf on Linux

- First install it (but it is not supported on Windows Subsystem for Linux)

```
sudo bash
apt install oprofile
echo 1 > /proc/sys/kernel/perf_event_paranoid
```

- Then measure a program such as intopt (as user — not root), do:

```
operf -e CYCLES:100000:0:0:1 intopt -y 15 1
```

- The first number specifies how many events to count before triggering an exception.
- The next is a mask, the third specifies whether kernel space should be monitored, and the fourth whether user space should be monitored.
- The statistics is saved in the directory `oprofile_data`.

intopt compiled without optimization

- `opreport -t 0.7 -l`

  prints all functions in which at least 0.7 % of the cycles where sampled.

- The essential parts of the output are:

  ```
  CPU: ppc64 POWER8, speed 3491 MHz (estimated)
  Counted CYCLES events (Cycles)
  samples  %         symbol name
  36404    87.3207   pivot
  2544      6.1022   xsimplex
  844       2.0245   select_nonbasic
  673       1.6143   xinitial
  368       0.8827   extend
  ```

intopt compiled with -O3

- Redoing the measurement with the `-O3` option to gcc, we instead get:

12079	86.3279	intopt	pivot
1216	8.6907	intopt	xinitial
242	1.7296	intopt	xsimplex.constprop.0
145	1.0363	libc-2.27.so	_int_malloc

- As we can see, `xsimplex` has been cloned and specialized with constant propagation.
- The goal is to find the most time-consuming function, which in both cases is `pivot`.

opannotate -s

- To see which source line in `pivot` gets most samples, we can use

  ```
  opannotate -s
  ```

  which prints the source code file and the number of samples taken for each line.
- With optimization, the line number information is no longer accurate and therefore we show the output from the unoptimized run.
- Most samples were taken in the following loop

```
  603  1.4464: for (i = 0; i < m; i += 1) {
  104  0.2495:  if (i == row)
   55  0.1319:   continue;
 1721  4.1281:  for (j = 0; j < n; j += 1)
  743  1.7822:   if (j != col)
27868 66.8458:    a[i][j] = a[i][j]
                    - a[i][col] * a[row][j] / a[row][col];
             : }
```

gprof

- Source code must be compiled with -pg
- Run the program first.
- Then use:

```
gprof -T intopt
```

which produces the output:

```
  %   cumulative    self              self    total
time     seconds   seconds    calls  ms/call ms/call name
88.3        1.05      1.05   148711    0.01     0.01 pivot
 5.9        1.12      0.07     7038    0.01     0.16 xsimplex
 2.5        1.15      0.03   148796    0.00     0.00 select_nonbasic
```

gprof: call graph

```
                0.00        1.19    3518/3518         intopt [1]
[4]     100.0   0.00        1.19    3518          succ [4]
                0.00        1.18    3518/3519         simplex [6]
                0.01        0.00    3518/3518         extend [14]
                0.00        0.00    2444/2445         integer [23]
                0.00        0.00    2428/2429         branch [24]
                0.00        0.00    1758/3519         free_node [20]
                0.00        0.00      16/16           bound [26]
```

- intopt made all 3518 calls to succ
- succ made 3518 of 3519 calls to simplex

Execution count per line: gcov

- Compile with -fprofile-arcs and -ftest-coverage.
- Run the program.
- Then use:

```
gcov simplex.c
```

which produces `simplex.c.gcov`:

```
 4130292: 478: for (i = 0; i < m; i += 1) {
 3981581: 479:  if (i == row)
  148711: 480:   continue;
64570580: 481:  for (j = 0; j < n; j += 1)
60737710: 482:   if (j != col)
56904840: 483:    a[i][j] = a[i][j]
                      - a[i][col] * a[row][j] / a[row][col];
       -: 484: }
```

Branch statistics: gcov -b

```
gcov -b intopt.c
```

- In the file `intopt.c.gcov` we find for the function `is_integer`:

```
   33933:  327:        r = lround(x);
       -:  328:
   33933:  329:        if (fabs(r-x) < intopt_eps) {
branch  0 taken 88% (fallthrough)
branch  1 taken 12%
   29743:  330:                *xp = r;
   29743:  331:                return 1;
       -:  332:        } else
    4190:  333:                return 0;
       -:  334:
```

An Example: Ordering If-Else-Statements 1(3)

- The code below tests for the most unlikely condition first!
- How can we improve the loop?

```
int     c;

while ((c = getchar()) != EOF)
        if (c == '\n')
                X;
        else if (c == ' ')
                Y;
        else
                Z;
```

An Example: Ordering If-Else-Statements 2(3)

- Test for the most likely path first!

```
int     c;

for (;;) {
        c = getchar();
        if (c > ' ')
                Z;
        else if (c == ' ')
                Y;
        else if (c == '\n')
                X;
        else if (c == EOF)
                break;
        else
                Z;
}
```

An Example: Ordering If-Else-Statements 3(3)

- Assume Z is so large we don't want to duplicate it!

```
int     c;

for (;;) {
        c = getchar();
        if (c > ' ')
L:              Z;
        else if (c == ' ')
                Y;
        else if (c == '\n')
                X;
        else if (c == EOF)
                break;
        else
                goto L;
}
```

Cachegrind

- Run with: `valgrind --tool=cachegrind ./a.out`

```
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%  )
```

Summary

- 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.

Using Simulators

- 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.

An Example

- 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.