Tutorial: Profilers

Anton Gerdelan <gerdela@scss.tcd.ie>

A Profiler is a Tool

- Analyse performance or complexity of your program
  - What are my most time-hungry functions? - **duration**
  - What are my most commonly called functions? - **frequency**
  - .: Where should I put effort into optimisation?
  - How much memory is used and where?
- Output: tables/spreadsheets and sometimes charts

Profilers

- Web browsers have great built-in and add-on profilers
- Xcode has "*Instruments*" - very good visualisations
- GNU/Linux has **gprof** - [should be] installed in labs
    - function durations and frequencies
- Visual Studio has a profiler + lots of add-ons (Intel VTune etc.)
- **valgrind** is great - installed in labs
    - very good for memory debugging
    - cache efficiency simulation

gprof

- Compile your program with the **`-pg`** flag

```
gcc -pg -o myprogram main.c
```

- Run the program, do normal stuff for a while

```
./myprogram
```

- This spits out an output log file called `gmon.out`
- Run `gprof` on the log to produce results tables

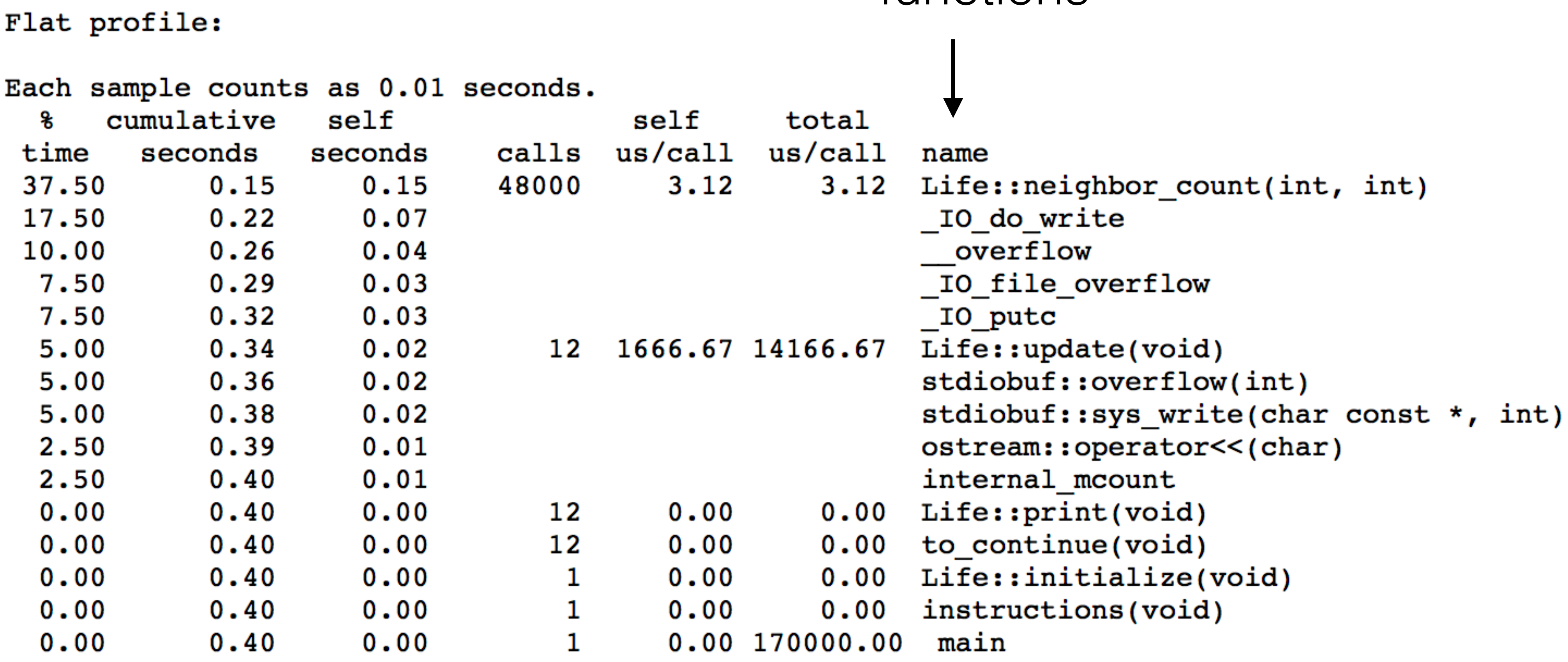
```
gprof myprogram gmon.out > results.txt
```

- Delete `gmon.out` between runs to restart results collection

```
rm gmon.out
```

Results: Flat Profile

functions



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 37.50      0.15     0.15    48000     3.12     3.12  Life::neighbor_count(int, int)
 17.50      0.22     0.07                             _IO_do_write
 10.00      0.26     0.04                             __overflow
  7.50      0.29     0.03                             _IO_file_overflow
  7.50      0.32     0.03                             _IO_putc
  5.00      0.34     0.02       12  1666.67 14166.67  Life::update(void)
  5.00      0.36     0.02                             stdiobuf::overflow(int)
  5.00      0.38     0.02                             stdiobuf::sys_write(char const *, int)
  2.50      0.39     0.01                             ostream::operator<<(char)
  2.50      0.40     0.01                             internal_mcount
  0.00      0.40     0.00       12     0.00     0.00  Life::print(void)
  0.00      0.40     0.00       12     0.00     0.00  to_continue(void)
  0.00      0.40     0.00        1     0.00     0.00  Life::initialize(void)
  0.00      0.40     0.00        1     0.00     0.00  instructions(void)
  0.00      0.40     0.00        1     0.00 170000.00  main
```

% of total program time
used by each func

total time spent in
each function by itself
table is sorted by this

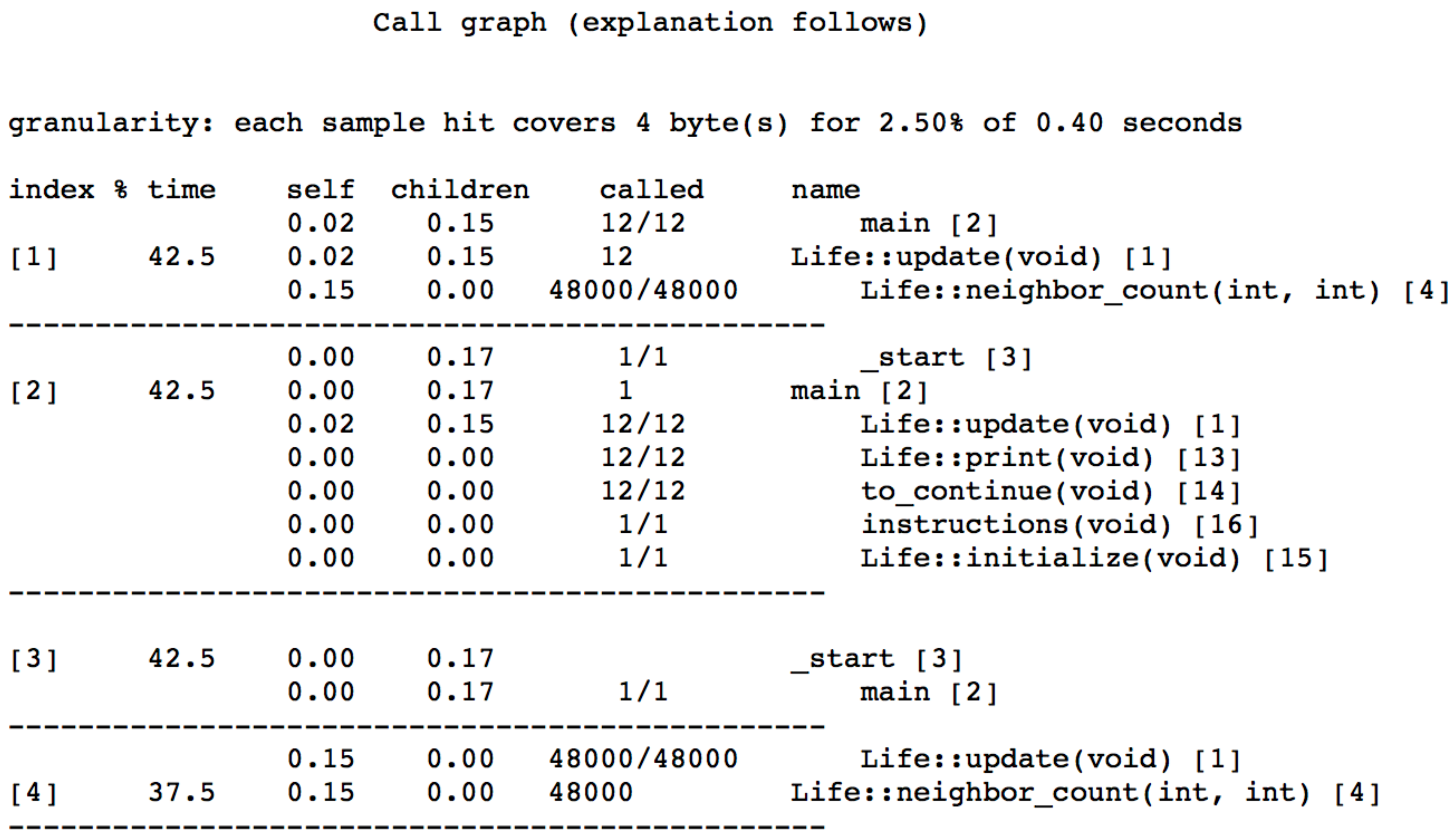
number of times
func is called

text src: http://web.eecs.umich.edu/~sugih/pointers/gprof_quick.html

Optimising Things

- Short, frequently called utility functions
  - consider **inlining**
- Long functions
  - look at code - O(n^2)+?
  - can it be simplified?
- Too many tiny function calls
  - hard to analyse and add up - look at **call graph**
  - high overhead - longer functions or recursion->loop?

Results: Call Graph



```
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 2.50% of 0.40 seconds

index % time    self  children    called     name
                0.02    0.15      12/12          main [2]
[1]     42.5    0.02    0.15      12         Life::update(void) [1]
                0.15    0.00   48000/48000       Life::neighbor_count(int, int) [4]
------------------------------------------------
                0.00    0.17       1/1           _start [3]
[2]     42.5    0.00    0.17       1         main [2]
                0.02    0.15      12/12          Life::update(void) [1]
                0.00    0.00      12/12          Life::print(void) [13]
                0.00    0.00      12/12          to_continue(void) [14]
                0.00    0.00       1/1           instructions(void) [16]
                0.00    0.00       1/1           Life::initialize(void) [15]
------------------------------------------------

[3]     42.5    0.00    0.17                 _start [3]
                0.00    0.17       1/1           main [2]
------------------------------------------------
                0.15    0.00   48000/48000       Life::update(void) [1]
[4]     37.5    0.15    0.00   48000         Life::neighbor_count(int, int) [4]
------------------------------------------------
```

- [1],[2],etc. - start of entry
- lines above - function that called this function
- lines below - functions called by this function
- costs include cost of child functions here

Optimising Things

- A **library** or driver is sucking up all the time
  - "*Things That Make You Go Hmmm*"
  - Can it be replaced?
  - Maybe this wheel should be reinvented…
- Generic code is expensive / debug build is too slow
  - Do you really need those templates/inheritance/virtual functions?
  - Turn on compiler optimisations with **`-O`** or **`-O3`**

    NB: this produces carbon!

Optimising Things

- Read literature and ask experts
  - is there a data structure or algorithm for this?
  - e.g. O(n^2) -> find O(log n)
    - may require some creative adaptation
- Know how the hardware works (and what it likes)
  - Look at **assembled** code for critical functions
  - are we misusing the **cache** or causing **page faults**

Optimising Things

- Profile again after trying things
  - usually you've made it *worse*
  - optimisation is hard but worth reasoning at this level about your work
  - try it on different computers

Optimising Things

- sometimes the answer is **no**
  - lose useful features/good work
  - lose clarity/simplicity
  - gains are too small to justify amount of work
  - optimised versions are too hardware-specific
- *engineering decisions*…
  - what - are the target machines?
  - who - is using this code?
  - when - quality vs deadlines or product turn-around time