122COM: Profiling

David Croft

Coventry University

david.croft@coventry.ac.uk

2017

Overview

Efficiency

C

When writing software think about its efficiency.

- Time.
- Memory.
- Time vs Memory.
  - Can you trade one for the other
  - I.e. data stored in RAM costs memory but saves time.
  - I.e. data stored on hard drive saves memory but costs time.
- Optimization makes software run faster/leaner/better.

Optimization

C

“Premature optimization is the root of all evil”

–Donald Knuth

For any large piece of code you should:

- Write clear, easily understood code. Focus on getting the behaviour right, not on performance.
- Test the performance.
  - It may be fine.
- Profile your code to get the baseline performance.
  - So that you know if you are making things better or worse.
- Focus your efforts on the code that is consuming all the time.
  - E.g. small pieces of code that get called multiple times.

Profiler types

A

Profiling is a method of analysing your code to identify the impact of the different functions/classes/sections etc.

Instrumentation profilers

- Add extra bits of code to track time/memory/function calls.
    - Can be done manually.
    - But automatic is better.
- Accurate.
    - But slows things down.

Statistical profilers

- Regularly checks the software state.
- Accurate-ish.
    - Based on statistical sampling.
    - Doesn’t slow things down.

Example

A

In this example which function takes the most time?

- `fast_math_function()` or `slow_math_function()`?
- Why don't we just profile it and find out?

```python
def fast_math_function(a, b):
    time.sleep(0.00001)
    return a + b

def slow_math_function(a, b):
    time.sleep(3)
    return a + b

def main():
    for i in range(int(1.0000)):
        slow_math_function(42, 69)

    for i in range(int(100000)):
        fast_math_function(42,69)

if __name__ == '__main__':
    sys.exit(main())
```

`lec_functions.py`

Profiler results

A

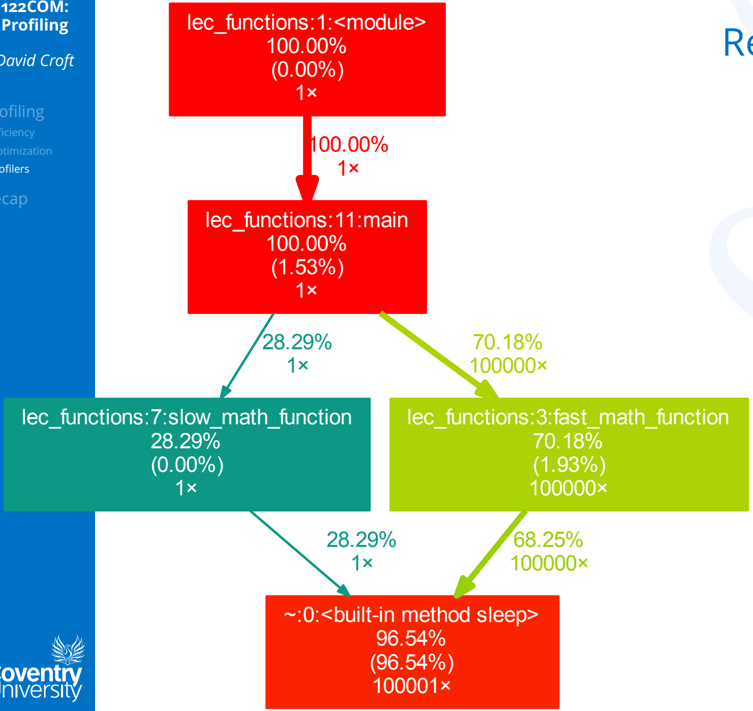
```
» python3 -m cProfile lec_functions.py

      200007 function calls in 10.362 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.000   0.000  10.362  10.362 lec_functions.py:1(<module>)
     1   0.137   0.137  10.362  10.362 lec_functions.py:11(main)
100000   0.171   0.000   7.222   0.000 lec_functions.py:3(fast_math_function)
     1   0.000   0.000   3.003   3.003 lec_functions.py:7(slow_math_function)
     1   0.000   0.000  10.362  10.362 {built-in method exec}
     1   0.000   0.000   0.000   0.000 {built-in method exit}
100001  10.054   0.000  10.054   0.000 {built-in method sleep}
     1   0.000   0.000   0.000   0.000 {method 'disable' of '_lsprof.Profiler' obje
```

Things to note:

- Total time - time spent in each function.
- Cumulative time - time spent in each function AND the functions it calls.

Results visualised

A



Results passed through Graphviz/gprof2dot.

- A profiling visualisation tool.

Why do I care?

- Everyone
  - Without *O*() notation we can't discuss how algorithms compare.
  - Without *O*() can't discuss why some tasks are effectively impossible (travelling salesman).
  - You should be trying to write good, efficient code. Profiling helps you do this.
- Ethical Hackers - *O*() important in discussing password security.
- Games Tech - *O*() explains the need for path finding and graphics work arounds.

Recap

Profiling help determines the actual performance of your code.

- Statistical profilers.
  - Accurate-ish
- Instrumental profilers.
  - Insert additional instructions.
  - Accurate but slows things down.

The End