GUI

David Croft

December 15, 2015

Abstract

This week we are going to be looking at profiling and how it relates to writing good code.

1 Introduction

Last semester we talked about $O()$ notation and how it can be used to give us an idea of the performance and time required for an algorithm to run.

Optimising your software trades code clarity (how easy it is to understand and maintain your code) for greater code performance (how fast your code is going to run). Being able to optimise you code is important if you want fast, reactive software. It can, however, do more harm that good if you optimise you code so much that it becomes difficult or impossible to understand

The most important factor in software optimisation is knowing where to optimise you code and this is where software profiling is important. Profiling is used to measure the performance of our software. It allows us to identify what parts of our code are using the most resources and so where we need to focus our attention if we want to improve the performance of our code.

2 Crude profiling

The simplest way to profile our software is to simply time how long it takes to run. Windows doesn't have a good, built-in way of doing this but on Linux it can be done using the **time** command.

Which will produce something like this...

```
real 0m4.214s
user 0m3.767s
sys  0m0.016s
```

`real` is the total, clock-on-the-wall time that the program took to run. As your code is running on a machine with other pieces of software the real time that it takes to run is dependent what else is happening on that machine. If you try and run your code at the same time and playing a game or processing video, it's going to take longer to run in real time.

The `user` time is the amount of time that was spent running the code that you wrote, the `sys` time is the amount of time that was spent doing system calls on behalf of your code. This is not affected by other programs running on the machine at the same time.

The total amount of processor time that you are responsible for is, therefore, user + sys. Interestingly for multi-threaded code the real time spent running your software can be less than the user + sys time

2.1

The second alternative is to manually add timing code to your software. This is normally only required while you are developing and should be removed once the code is finished.

```python
import sys, time

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():
    slow_math_function(42, 69)

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

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

Listing 1: timingcode.py

```python
import sys, time
from datetime import datetime

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():
    start = datetime.now()
    slow_math_function(42, 69)
    delta = datetime.now() - start
    print( 'Slow math took %fs' % \
        delta.total_seconds() )

    start = datetime.now()
    for i in range(100000):
        fast_math_function(42,69)
    delta = datetime.now() - start
    print( 'Fast math took %fs' % \
        delta.total_seconds() )

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

Listing 2: timingcode.py

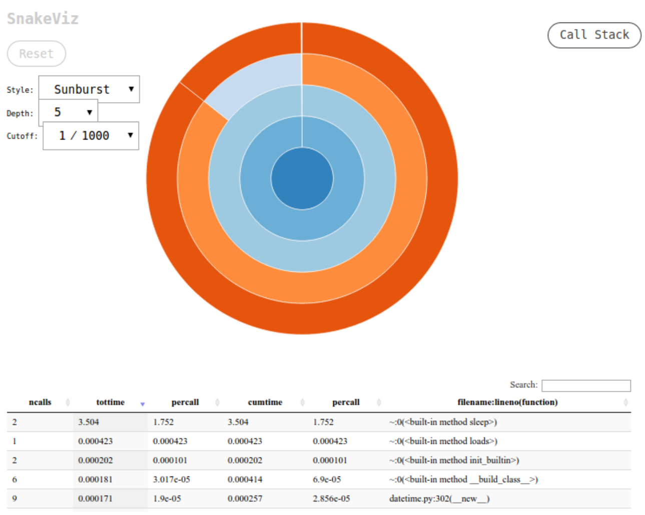
This method lets us test how long individual portions of out code are taking the run but only measures the real time that the code took and, as we saw in the previous section, that's unreliable.

3 Deterministic profiling

Finally we come to deterministic profiling. This means using software to externally monitor precisely how long each portion of your code takes to run. Deterministic profiling is not perfect, running the same code multiple times will likely produce slightly different results each time. Despite this it remains one of the most powerful tools for examining software performance.

There are a number of different Python profilers available but we are going to use cProfile as it's the most reliable built in profiler.

cProfile does not require that you modify you code in anyway. You can profile your existing



Style: Sunburst
Depth: 5
Cutoff: 1 / 1000

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2	3.504	1.752	3.504	1.752	~:0(<built-in method sleep>)
1	0.000423	0.000423	0.000423	0.000423	~:0(<built-in method loads>)
2	0.000202	0.000101	0.000202	0.000101	~:0(<built-in method init_builtin>)
6	0.000181	3.017e-05	0.000414	6.9e-05	~:0(<built-in method __build_class__>)
9	0.000171	1.9e-05	0.000257	2.856e-05	datetime.py:302(__new__)

Figure 1: Snakeviz window

software using the following command.

```
>> python -m cProfile nameofyoursoftware.py
```

The output of cProfile is not that easier to understand so we can make use of profiler visualisation tools. As with everything Python there are a wide range of different visualisation tools available. I personally recommend and gprof2dot which also works with C and C++ but we are going to be using snakeviz as it's simpler to run and works on all platforms. To use snakeviz enter the following commands

```
>> python -m cProfile -o profile nameofyoursoftware.py
```

```
>> snakeviz profile
```

The first command runs our software with cProfile and saves the profiling results in a file called 'profile'. The second command runs snakeviz using the profiling results that we just generated. Snakeviz should now open a window in your default browser that looks like figure 1.

Pre-lab work: Snakeviz

Snakeviz provides a number of different viewing and sorting options so experiment with them until you find the settings you prefer.

There are two factors to consider when looking at the performance of a function. The first of these is the per-call running time, this is the average amount of time that the function takes to run each time that it is called. The second is the cumulative running time, this is the total amount of time spent running the function.

`fast_math_function` takes $\approx$ 0.0001 seconds to run compared to `slow_math_function` which takes $\approx$ 3 seconds. So if we want to speed our software up it would appear that we should fix `slow_math_function` first. However, because `fast_math_function` gets called so much more often than `slow_math_function`, even a small improvement in it can have a much bigger overall effect.

When optimising your code make sure that the benefits are worth the time and effort that it's going to require.

4

In our previous example it's obvious how we can speed our code up, just remove all the `time.sleep()` calls. It's unlikely to be that easy in your code. The functions listing 3 on the next page shows two functions that calculate the first n prime numbers. But one is much more efficient than the other. You could spend time reading through all the code and figuring out $O()$ complexities OR you could just profile the code.

Pre-lab work: aa

Figure out which of the functions shown in listing 3 on the following page is fastest.

Extended work: aa

See if you can improve the faster function.

```
import sys

def find_first_n_version1(self, n):
    primes = [2]

    counter = 3
    while len(self.primes) < n:
        prime = True

        for i in range(3,counter):
            if counter % i == 0:
                prime = False

        if prime:
            self.primes.append(counter)

        counter += 1

    return primes

def find_first_n_version2(self, n):
    primes = []

    counter = 2
    while len(self.primes) < n:
        prime = True

        for i in self.primes:
            if counter % i == 0:
                prime = False

        if prime:
            self.primes.append(counter)

        counter += 1

    return primes

def main():
    primesToFind = 1000

    find_first_n_version1( primesToFind )
    find_first_n_version2( primesToFind )

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

Listing 3: profiling_b.py

5 C++ profiling

Unfortunately profiling C++ code is a more complex process. As with Python there are range of different profilers available and matters are made more difficult as not all profilers and visualisers are available on all Operating Systems (OSs).

The professional version of Microsoft Visual Studio for example has a built in profiler, if you are using the free version then you will need to rely on 3rd party tools such as VerySleepy.

We are going to be using gprof for the profiling as this is a standard g++ profiler. As a compiled language we can't just run our code to profile it, firstly we have to recompile our code with additional profiling information. This is done by adding the `-pg` option when compiling.

```
>> g++ -std=c++11 -pg programname.cpp -o programname
```

Linux/Mac

```
>> ./programname

>> gprof programname
```

Windows in Cygwin

```
>> ./programname.exe

>> gprof programname.exe
```

```
>> gprof programname | grof2dot | dot -Tpdf -o programname.pdf
```

The results of this are similar to those of cProfile.

5.1 Graphviz

As with Python and snakeviz we may want to visualise the results of our profiling. There are multiple options for viewing C++ profiling results. We are going to being looking at how to use gprof2dot to create profiling visualisations. The reason that we are using this approach as opposed to one of the alternatives is that gpro2dot is platform independent and also works with Python.

```
>> python -m cProfile -o results.prof [YOUR_PROGRAM_NAME].py
>> python gprof2dot.py -f pstats results.prof | dot -Tpdf results.pdf
```

Your Python program is run using cProfile to profile it, the results of the profiling are saved in the file results.prof. The results.prof file is read in by the grof2dot program which creates and outputs a Graphviz file, this Graphviz file is fed into the dot program which turns it into a pdf. As this is all horribly complex, I have created a program that does all of this for you.

Download @@@ from moodle, create your .prof file as normal then just run it through the profile_python program as shown below to create a lovely visualisation of your profiling results.

```
>> profile_python results.prof
```

We are going to be using grof as this is a standard profiler to g++.