GUI

David Croft

February 5, 2016

Abstract

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

1 Introduction

Optimising your software trades code clarity (how easy it is to understand and maintain your code) for greater performance (how fast your code is going to run, how much memory it will require). Being able to optimise you code is important if you want fast/small, 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 your 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 and Macs 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 User code

The second alternative is to manually add timing code to your software. This is normally only required while you are developing and would 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: pre_no_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: pre_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 start with cProfile as it's a reliable built in profiler.

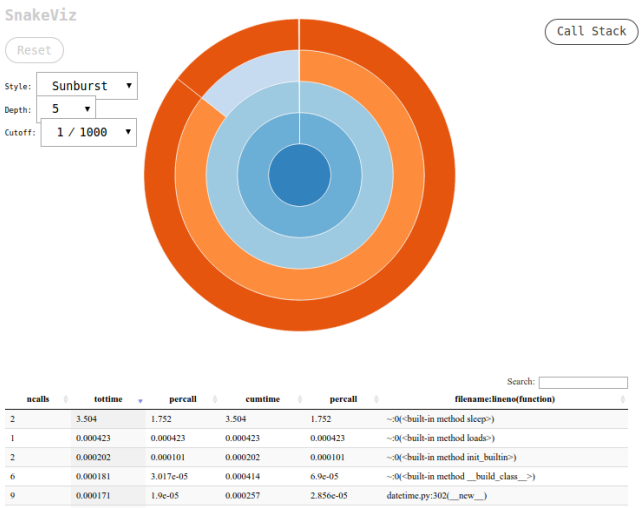

Figure 1: Snakeviz window

cProfile does not require that you modify you code in anyway. You can profile your existing software using the following command.

```
>> c:\python 34\python -m cProfile NAMEOFYOURPROGRAM.py
```

This provides a lot of valuable information but is not that easier to understand. To make things easier we can make use of profiler visualisation tools, these convert the raw profiling information into pretty pictures that make our lives much easier.

Pre-lab work:

1. Try profiling pre_no_timingcode.py or pre_no_timingcode.py
   - Try and interpret the profiling results.

3.1 Visualisation tools

We are going to be starting with snakeviz as it's simple to run and works on all platforms. To use snakeviz enter the following commands

```
>> c:\python 34\python -m cProfile -o profile NAMEOFYOURPROGRAM.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.

3.2 Testing

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. pre_profiling contains two functions that calculate the first n prime numbers. But one is much faster efficient than the other. You could spend time reading through all the code and figuring out which is better OR you could just profile the it.

Pre-lab work:

1. Profile pre_profiling.py
2. Which of the two functions is contains is faster?

Extended work:

1. See if you can write your own function with a better performance than `find_first_n_version1()` and `find_find_n_version2()`.
2. Make sure to profile your code to prove it's superior.

4 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). We're still going to be doing deterministic profiling but we are going to have to use different tools.

We are going to be using gprof for the profiling as this is a standard g++ profiler. Since C++ is 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

```
>> PROGRAMNAME.exe
```

```
>> gprof PROGRAMNAME
```

The results of this are similar to those of cProfile.

Lab work:

3. Try profiling pre_profiling.cpp
4. See how the results compare to those of pre_profiling.py

4.1 Visualisation

Just like with Python there are visualisation tools available for C++. We're going to be discussing gprof2dot and graphviz.

Graphviz (or dot) is a graph drawing program which takes in a specially formatted "dot" file and draws a corresponding graph while handelling the layout for you.

Gprof2dot is a program that converts the profiling results from gprof into the format that graphviz requires.

When combined (as shown below) together they can produce a profiling graph for your C++ programs.

```
>> gprof PROGRAMNAME | gprof2dot | dot -Tpdf -o PROGRAMNAME.pdf
```

Graphviz (dot) and gprof2dot have a wide range of options that can be tuned and configured but you can explore those in your own time.

Lab work:

5. Try visualising pre_profiling.cpp
   - On the university machines you will need make sure that graphviz is installed using the software portal first.
     Just search for graphviz and click 'launch'.
   - Because of where the software portal installs graphviz, things are not as straight forward as the instructions I have given you above.
   - In order to make things easier use the cppprof.bat file provided in the github repo instead.

     ```
     >> cppprof.bat YOURPROGRAM.cpp
     ```

     or

     ```
     >> cppprof.bat YOURPROGRAM.cpp VISUALISATION.pdf
     ```
   - The profiling graph will be saved to out.pdf by default.

4.1.1 Python

Graphviz and gprof2dot also works for Python but this wasn't mentioned earlier since snakeviz is a more straightforward solution

The steps for using graphviz and gprof2dot with python are shown below.

```
>> python -m cProfile -o results.prof YOURPROGRAM.py
>> python gprof2dot.py -f pstats results.prof | dot -Tpdf -o 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 dot file, this file is fed into the graphviz (dot) program which turns it into a pdf.

On the university machines things are made more complicated by the graphviz installation location so use the pythonprof.bat file off the github repo instead. Usage is the same as cpp-prof.bat.