Performance Profiling in Python

Network vs CPU Bottlenecks Analysis

Karen Cardiel Olea
Universidad Politécnica de Yucatán
Email: 22090239@upy.edu.mx

***Abstract*—This technical report presents a case study on Python profiling techniques to identify and distinguish between I/O-bound and CPU-bound performance bottlenecks. Two scenarios were analyzed, downloading weather station data from NOAA servers (I/O-bound) and computing distances between geographic stations (CPU-bound). Using tools such as cProfile, SnakeViz, and line_profiler, to demonstrate how proper profiling enables targeted optimization strategies.**

I Section 2.1: Network / I-O Profiling

I-A What I Did

I executed a Python script that downloads weather data from NOAA servers for multiple stations and years. I compared two versions:

- **Without cache** (`load.py`): downloads data every time from NOAA servers.
- **With cache** (`load_cache.py`): checks if file exists before downloading using `os.path.exists()`, reusing previously downloaded files.

Commands executed:

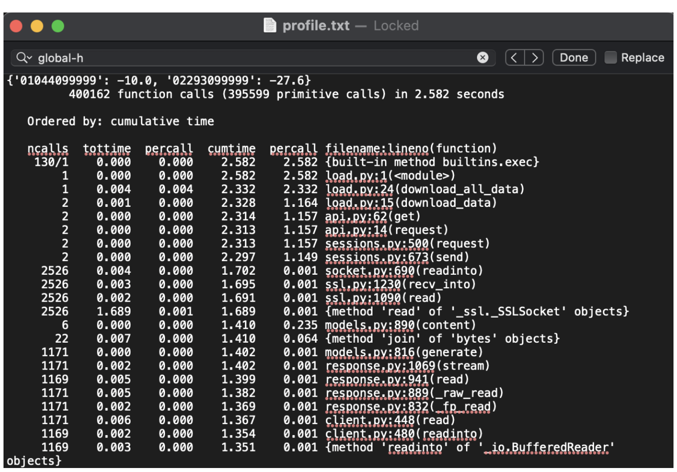
First run (without cache):

```
python3 -m cProfile -s cumulative load.py
    01044099999,02293099999 2021 2021 > profile.txt
```

Second run (with cache):

```
python3 -m cProfile -s cumulative load_cache.py
    01044099999,02293099999 2021 2021 > profile_cache.
    txt
```

I-B Results

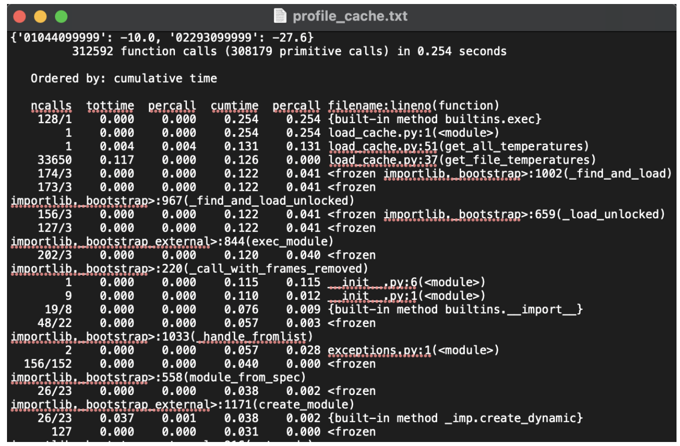


```
profile.txt — Locked
global-h   Done   Replace
{'01044099999': -10.0, '02293099999': -27.6}
         400162 function calls (395599 primitive calls) in 2.582 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    130/1    0.000    0.000    2.582    2.582 {built-in method builtins.exec}
        1    0.000    0.000    2.582    2.582 load.py:1(<module>)
        1    0.004    0.004    2.332    2.332 load.py:24(download_all_data)
        2    0.001    0.000    2.328    1.164 load.py:15(download_data)
        2    0.000    0.000    2.314    1.157 api.py:62(get)
        2    0.000    0.000    2.313    1.157 api.py:14(request)
        2    0.000    0.000    2.313    1.157 sessions.py:500(request)
        2    0.000    0.000    2.297    1.149 sessions.py:673(send)
     2526    0.004    0.000    1.702    0.001 socket.py:690(readinto)
     2526    0.003    0.000    1.695    0.001 ssl.py:1230(recv_into)
     2526    0.002    0.000    1.691    0.001 ssl.py:1090(read)
     2526    1.689    0.001    1.689    0.001 {method 'read' of '_ssl._SSLSocket' objects}
        6    0.000    0.000    1.410    0.235 models.py:890(content)
       22    0.007    0.000    1.410    0.064 {method 'join' of 'bytes' objects}
     1171    0.000    0.000    1.402    0.001 models.py:816(generate)
     1171    0.002    0.000    1.402    0.001 response.py:1069(stream)
     1169    0.005    0.000    1.399    0.001 response.py:941(read)
     1171    0.005    0.000    1.382    0.001 response.py:889(_raw_read)
     1171    0.002    0.000    1.369    0.001 response.py:832(_fp_read)
     1171    0.006    0.000    1.367    0.001 client.py:448(read)
     1169    0.002    0.000    1.354    0.001 client.py:480(readinto)
     1169    0.003    0.000    1.351    0.001 {method 'readinto' of '_io.BufferedReader'
objects}
```

Fig. 1. cProfile results without caching - showing high requests.get time



```
profile_cache.txt
{'01044099999': -10.0, '02293099999': -27.6}
         312592 function calls (308179 primitive calls) in 0.254 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    128/1    0.000    0.000    0.254    0.254 {built-in method builtins.exec}
        1    0.000    0.000    0.254    0.254 load_cache.py:1(<module>)
        1    0.004    0.004    0.131    0.131 load_cache.py:51(get_all_temperatures)
    33650    0.117    0.000    0.126    0.000 load_cache.py:37(get_file_temperatures)
    174/3    0.000    0.000    0.122    0.041 <frozen importlib._bootstrap>:1002(_find_and_load)
    173/3    0.000    0.000    0.122    0.041 <frozen
importlib._bootstrap>:967(_find_and_load_unlocked)
    156/3    0.000    0.000    0.122    0.041 <frozen importlib._bootstrap>:659(_load_unlocked)
    127/3    0.000    0.000    0.122    0.041 <frozen
importlib._bootstrap_external>:844(exec_module)
    202/3    0.000    0.000    0.120    0.040 <frozen
importlib._bootstrap>:220(_call_with_frames_removed)
        1    0.000    0.000    0.115    0.115 __init__.py:6(<module>)
        9    0.000    0.000    0.110    0.012 __init__.py:1(<module>)
     19/8    0.000    0.000    0.076    0.009 {built-in method builtins.__import__}
    48/22    0.000    0.000    0.057    0.003 <frozen
importlib._bootstrap>:1033(_handle_fromlist)
        2    0.000    0.000    0.057    0.028 exceptions.py:1(<module>)
  156/152    0.000    0.000    0.040    0.000 <frozen
importlib._bootstrap>:558(module_from_spec)
    26/23    0.000    0.000    0.038    0.002 <frozen
importlib._bootstrap_external>:1171(create_module)
    26/23    0.037    0.001    0.038    0.002 {built-in method _imp.create_dynamic}
      127    0.000    0.000    0.031    0.000 <frozen
```

Fig. 2. cProfile results with caching - showing reduced network calls

I-C Observations and Interpretation

Most of the time was spent in:

- `requests.get()` - HTTP requests to NOAA servers
- SSL socket reads and network I/O operations

After implementing caching:

- Execution time decreased significantly on subsequent runs
- Network calls reduced to zero for already-downloaded files

This demonstrates a clear **I/O-bound process**, where performance depends primarily on network speed and latency rather than computational capacity. The simple file existence check (`os.path.exists()`) eliminated redundant downloads and dramatically improved performance.

II Section 2.2: CPU Profiling

II-A What I Did

I cloned the code repository from the book:

```
git clone https://github.com/tiagoantao/python-
    performance
```

The relevant code is located in `02-python/sec2-cpu/`.

I computed pairwise distances between weather stations using geographic coordinates with the Haversine formula. This operation has O(n²) complexity, making it computationally intensive.

Tools used:

- `cProfile`: High-level performance overview
- `SnakeViz`: Visualization of profiling results
- `line_profiler`: Per-line performance analysis

Execution steps:

Step 1: Generate profiling file

```
python3 -m cProfile -o distance_cache.prof
    distance_cache.py
```

Step 2: Visualize with SnakeViz

```
python3 -m snakeviz distance_cache.prof
```

Step 3: Line profiling

```
python3 -m kernprof -l lprofile_distance_cache.py
python3 -m line_profiler lprofile_distance_cache.py.
    lprof
```

II-B Results

Fig. 3. Terminal after running cProfile

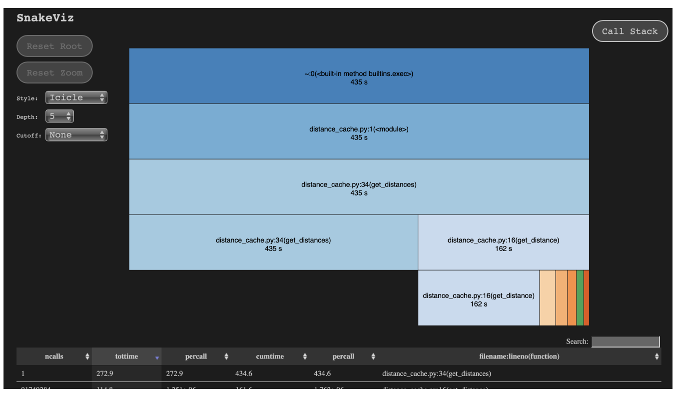


Fig. 4. SnakeViz visualization showing time distribution

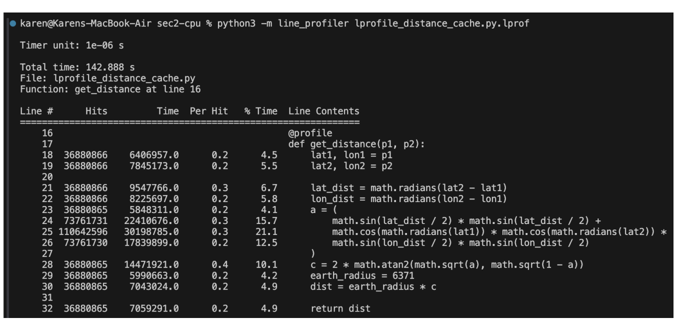


Fig. 5. Line profiler output showing bottleneck lines

Key metrics:

- Total execution time: 142.888 seconds
- Function analyzed: `get_distance()`
- Total calls per line: ~36.8 million
- Algorithm complexity: O(n²)

TABLE I
MOST EXPENSIVE LINES IN GET_DISTANCE()

Line	Operation	% Time
24	`math.cos(radians(lat1)) * math.cos(...)`	21.1%
23	`math.sin(lat_dist / 2) * math.sin(...)`	15.7%
25	`math.sin(lon_dist / 2) * math.sin(...)`	12.5%
28	`math.atan2(...)`	10.1%

II-C Observations and Interpretation

Majority of execution time spent in mathematical operations:

- Trigonometric functions (`math.sin`, `math.cos`)
- Repeated `math.radians()` conversions
- `math.atan2()` calculations

Lines 23-25 combined account for nearly 50% of total execution time. The bottleneck is heavy computation repeated millions of times. Each operation is fast (approximately 0.2-0.4 microseconds), but 36+ million calls create significant overhead.

This confirms a **CPU-bound problem**. Unlike Section 2.1, this slowdown is not related to waiting for external resources, but to:

- Pure computation with intensive mathematical operations
- Algorithmic complexity: O(n²) means calculations grow quadratically
- Pattern: *Small cost* × *Massive repetition* = *Major bottleneck*

III Conclusions

Key findings:

- Profiling tools are essential to identify bottleneck origins before optimization
- I/O-bound tasks benefit from caching and reducing redundant operations (Section 2.1)
- CPU-bound tasks require algorithmic or implementation-level optimization (Section 2.2)
- Visual tools (SnakeViz) and line profiling provide targeted insights
- Scaling up is mostly about swapping slow O(n²)) logic for faster (O(n) solutions."

References

[1] T. Antao, *Fast Python: High Performance Techniques for Large Datasets*. Manning Publications, 2023, ch. 2, pp. 18-28.