Python & Profiling
par serge-sans-paille

- Ingénieur R&D en compil' à Quarkslab
- Chercheur associé à (feu) Télécom Bretagne
- Core dev Pythran
Premature optimization is the root...

*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.* -- Hoare
The Fallacy of Premature Optimization

By Randall Hyde [full article](http://ubiquity.acm.org/article.cfm?id=1513451)

1. != Optimization is the root of all evil
2. engineers do not consider application performance during the design of the software
3. Pareto principle is OK, but only if performance bottleneck is local
4. Moore's law is dead
5. engineer don't profile (correctly)
6. engineer time << user time
7. optimization delay delivery but improve user experience
8. better algorithm don't solve all solutions
9. substituting algorithm is not always easy
Myth: Python is slow

from the computer language benchmark game
(http://benchmarksgame.alioth.debian.org/)

binary-trees

<table>
<thead>
<tr>
<th>source</th>
<th>secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python 3</td>
<td>152.06</td>
</tr>
<tr>
<td>C++ g++</td>
<td>6.98</td>
</tr>
</tbody>
</table>
Reality: CPython can be fast

from the computer language benchmark game (http://benchmarksgame.alioth.debian.org/), pidigits benchmark

×   source  secs
1.0 Pascal  1.73
1.0 C gcc   1.73
1.0 Rust    1.74
1.1 Fortran 1.92
1.3 Python3 2.20
1.3 C++ g++  2.29
Reality: numeric computations can be slow

spectral-norm benchmark, that uses list and scalars

<table>
<thead>
<tr>
<th>x</th>
<th>source</th>
<th>secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>C gcc</td>
<td>1.98</td>
</tr>
<tr>
<td>2.1</td>
<td>Java</td>
<td>4.26</td>
</tr>
<tr>
<td>8.0</td>
<td>Node.js</td>
<td>15.77</td>
</tr>
<tr>
<td>126</td>
<td>Python3</td>
<td>250.12</td>
</tr>
</tbody>
</table>
But Python != CPython

$ gcc sn.c -o sn -O3
$ time ./sn 5500
./sn 5500  4.86s user 0.00s system 99% cpu 4.864 total
$ pythran sn.py
$ python -m timeit 'import sn' 'sn.main(5500)'
10 loops, best of 3: 4.79 sec per loop
Understanding CPython Performance

In [1]:
import dis
code = lambda x, y : x + y
dis.dis(code)

2           0 LOAD_FAST                0 (x)
3 LOAD_FAST                1 (y)
6 BINARY_ADD
7 RETURN_VALUE

foo:
leal (%rdi,%rsi), %eax
ret
Glimpses of explanations #1

Indirections Everywhere

- LOAD_FAST $\Rightarrow$ array lookup
- BINARY_ADD $\Rightarrow$ dict lookup

Function Call Overhead

- Suspend current frame
- Create a new frame
- Push it on the stack
Almost no optimizations

$ gcc sn.c -o sn -O0 -fsanitize=address -fsanitize=null -fsanitize=signed-integer-overflow -fsanitize=integer-divide-by-zero
$ time ./sn 5500
./a.out 5500   11.04s user 0.02s system 99% cpu 11.053 total
Glimpses of explanations #3

Poor Parallelism Support

- Global Interpreter Lock ⇒ few parallelism gain (except io/native calls)
- Do not speak about vectorization
- Generally no direct hardware access

Poor Memory Locality

How many allocations in

```python
[x**y for x, y in enumerate(range(100, 200))]
```
Glimpses of Hope

- Efficient dictionary
- Cached String hashing
- BigInt comparable to GMP
- Many Native Library Wrappers

Python was designed as a wrapping language
Profiling tools

Profiling = "sampling" or "instrument"

Official tools:

- cProfile
- profile
- -hotspot-
cProfile

In [2]: !python -m cProfile -h

Usage: cProfile.py [-o output_file_path] [-s sort] scriptfile [arg] ...

Options:
-h, --help           show this help message and exit
-o OUTFILE, --outfile=OUTFILE
                   Save stats to <outfile>
-s SORT, --sort=SORT Sort order when printing to stdout, based on
                   pstats.Stats class
cProfile Example

cumulated time, saved to a file
python -m cProfile -o myscript.prof -s cumtime myscript.py arg0 arg1

total time, printed to stdout
python -m cProfile -s tottime myscript.py arg0 arg1
cProfile Limitations

no mutithreading support

- Only main thread is profiled
- Still possible to use the API to manually collect and merge stats

function level granularity

- no line information

text output

- difficult to sort relevant information for large applications
pip install yappi

Pro

- multithread support
- callgrind or pstat output

Contra

- only supports Python < 3.5
pip install pprofile

Pro

- multithread support
- callgrind or text output
- statistic profiling

Contra

- relatively slow
pip install snakeviz

Name: filter
Cumulative Time: 0.000294 s (31.78 %)
File: fnmatch.py
Line: 48
pip install pyprof2calltree+callgrind
Benchmarking

Standard tool
- timeit

Extension tool
- perf
timeit

In [3]: %timeit [1 for _ in range(100)]
   2.63 µs ± 61.5 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [4]: %timeit -r 3 -n 100 [1 for _ in range(100)]
   2.9 µs ± 74.9 ns per loop (mean ± std. dev. of 3 runs, 100 loops each)

In [5]: !python -m timeit -s "x=list(range(100))" "[1 for _ in x]"
   100000 loops, best of 3: 2.08 usec per loop
pip install perf

In [6]: `!python -m timeit -s "x=list(range(100))" "[1 for _ in x]"
   
100000 loops, best of 3: 2.12 usec per loop

In [7]: `!python -m perf
   
usage: -m perf [-h]
   {show, hist, compare_to, stats, metadata, check, collect_metadata, timeit, system, convert, dump, slowest, command}
   ...`
perf usage

In [8]: python -m perf timeit -o perf.json "[1 for _ in range(10)]"

ERROR: The JSON file 'perf.json' already exists

In [9]: python -m perf hist perf.json

568 ns:  3 ################
572 ns:  6 ##########################
576 ns:  2 ################
579 ns:  3 ##########################
583 ns:  9 ##########################
587 ns: 13 ##########################
591 ns:  5 ##########################
595 ns:  3 ##########################
598 ns:  5 ##########################
602 ns:  2 ##########################
606 ns:  0 |
610 ns:  2 ##########################
614 ns:  2 ##########################
617 ns:  0 |
621 ns:  1 ####
625 ns:  1 ####
629 ns:  1 ####
633 ns:  0 |
636 ns:  0 |
640 ns:  0 |
644 ns:  1 ####
648 ns:  1 ####
In [10]: !python -m perf command ls

.................
command: Mean +- std dev: 980 us +- 29 us
perf again
$ perf record cmd arg0 arg1

(may require root privilege)

$ perf report
perf visu

The image contains a performance visualization, likely from a tool like `perf`, which is used for profiling and measuring system performance. The page displays a list of processes and system calls, with their respective overhead, cycles, and event counts. The text is technical and related to system performance monitoring, showing specific system calls and their associated performance metrics.
Conclusion

- Profile before you optimize
- Think before you profile
- Plenty of supporting tools