Johny’s Software Lab

Why does my program behave like this?

Tools to help you quickly understand the behavior of your program

https://johnysswlab.com, @johnysswlab ivica@johnysswlab.com

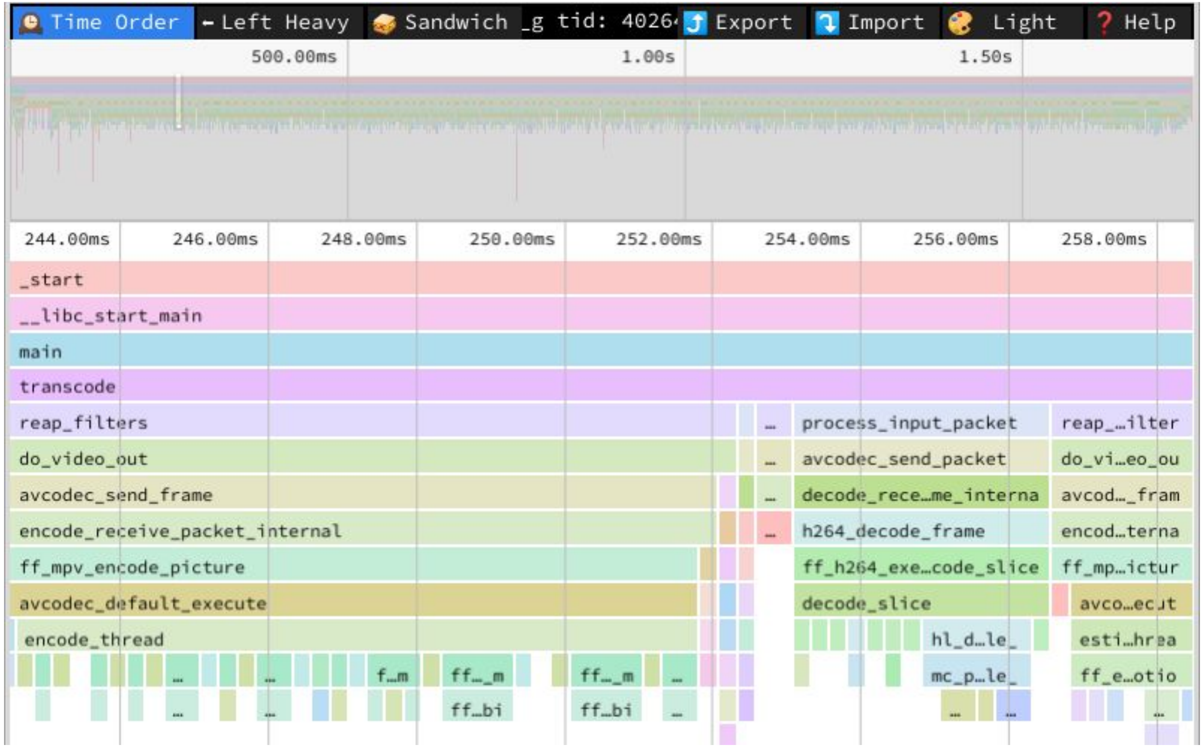
About me

- Ivica Bogosavljevic - application performance specialist
- Professional focus is C/C++ application performance improvement:
  - Better algorithms
  - Better exploiting the underlying hardware
  - Better usage of the standard library
  - Better usage of programming language
  - Better usage of the operating system.
- Work as a an external expert
  - If your software is slow, I can help you make it faster
- Writer for software performance blog: Johny's Software Lab - link in the footer
  - For all the people interested in software performance

Introduction

- Starting a work on unfamiliar codebase
  - Hot and cold functions
  - Timeline
  - Memory usage
  - Hardware efficiency
  - How program interacts with the operating system
- Debugging not covered here
  - Debugging weird behavior
  - Debugging memory issues
  - Debugging thread races
  - Difficult to reproduce bugs
- All examples are available here: https://github.com/ibogosavljevic/johnysswlab/tree/master/talks/tools

Flamegraphs - e.g. speedscope

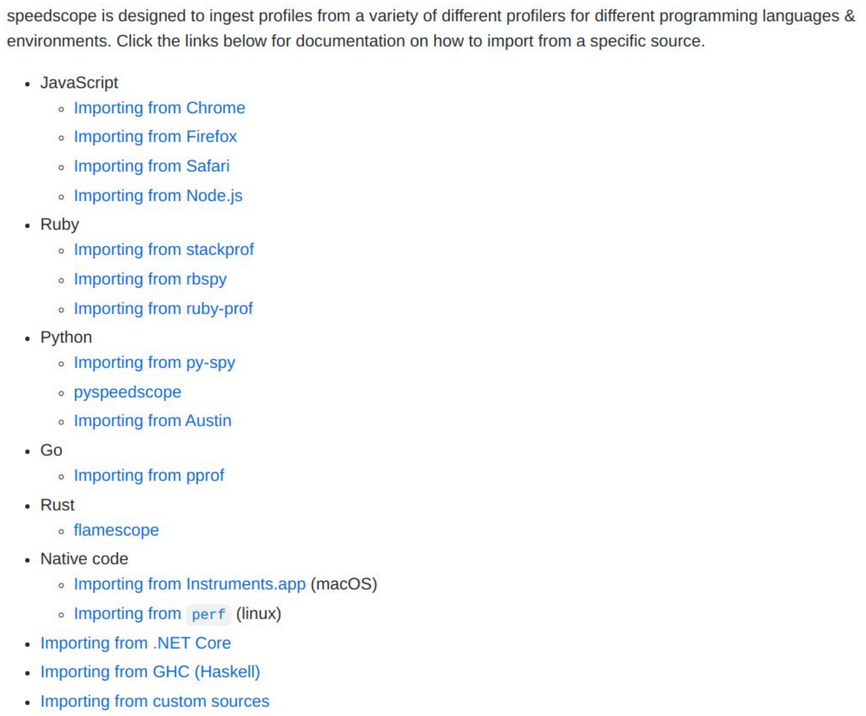


- FFMPEG Example: https://www.speedscope.app/#profileURL=https%3A%2F%2Fraw.githubusercontent.com%2Fibogosavljevic%2Fjohnysswlab%2Fmaster%2F2021-03-speedscope%2Fspeedscope-ffmpeg.txt
-

Speedscope

- Hands-on in terminal
- Reference
  - `perf record --call-graph dwarf -F 99 ./my_app`
  - Parameter -F is the sampling frequency, increase for short running processes, decrease for long running processes
  - The app needs to be compiled with debug symbols (`-g` for GCC and CLANG)
  - perf script | speedscope -
- More information about supported platforms and installation: https://www.speedscope.app/
- More information on how to collect data using perf:
- https://johnysswlab.com/speedscope-visualize-what-your-program-is-doing-and-where-it-is-spending-time/

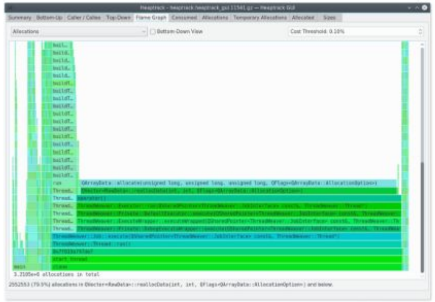
Speedscope



speedscope is designed to ingest profiles from a variety of different profilers for different programming languages & environments. Click the links below for documentation on how to import from a specific source.

- JavaScript
  - Importing from Chrome
  - Importing from Firefox
  - Importing from Safari
  - Importing from Node.js
- Ruby
  - Importing from stackprof
  - Importing from rbspy
  - Importing from ruby-prof
- Python
  - Importing from py-spy
  - pyspeedscope
  - Importing from Austin
- Go
  - Importing from pprof
- Rust
  - flamescope
- Native code
  - Importing from Instruments.app (macOS)
  - Importing from `perf` (linux)
- Importing from .NET Core
- Importing from GHC (Haskell)
- Importing from custom sources

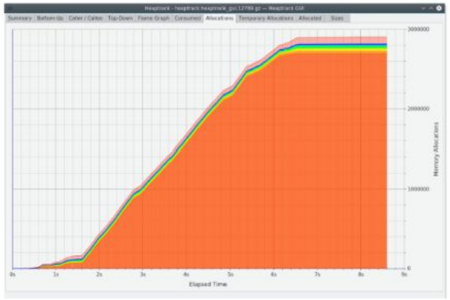
Heap profilers, e.g. Heaptrack



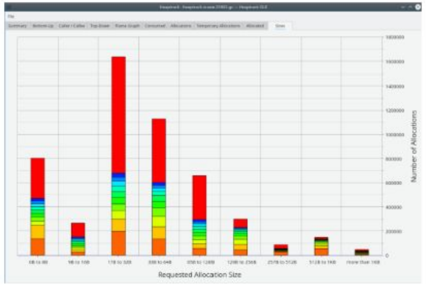
3.2105e+6 allocations in total

2552553 (79.5%) allocations in QVector<RawData>::reallocData(int, int, QFlags<QArrayData::AllocationOption>) and below.

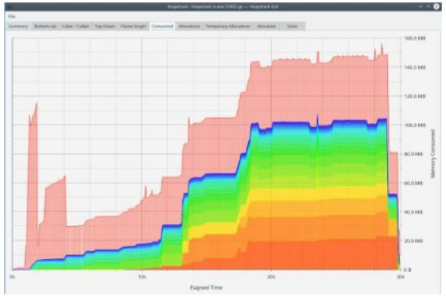
Flamecharts



Cumulated allocations



Sizes



Consumed

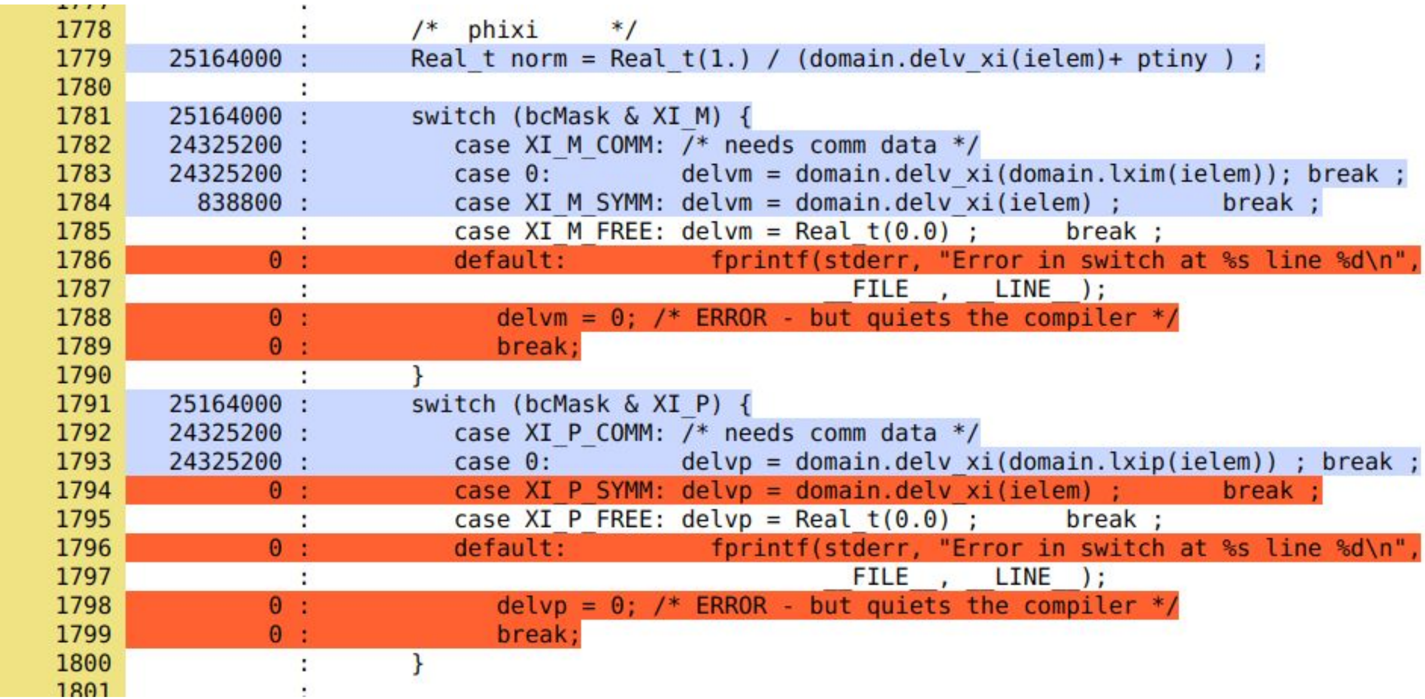
- Debugging memory consumption issues

https://johnysswlab.com, @johnysswlab ivica@johnysswlab.com

Heaptrack

- Hands-on in terminal
- Installation through packet manager
- Reference
  - Compile with debug symbols (`-g` on GCC and CLANG)
  - Recording information: `heaptrack ./my_app`
  - Visualization: `heaptrack --analyze heaptrack*.gz`
- Additional information
  - https://www.youtube.com/watch?v=ZXTI5iWHhrg
  - https://github.com/adesitter/accu_presentations/blob/master/ReducingMemoryAllocations_CppOnSea_2020.pdf

Coverage tools, e.g. lcov

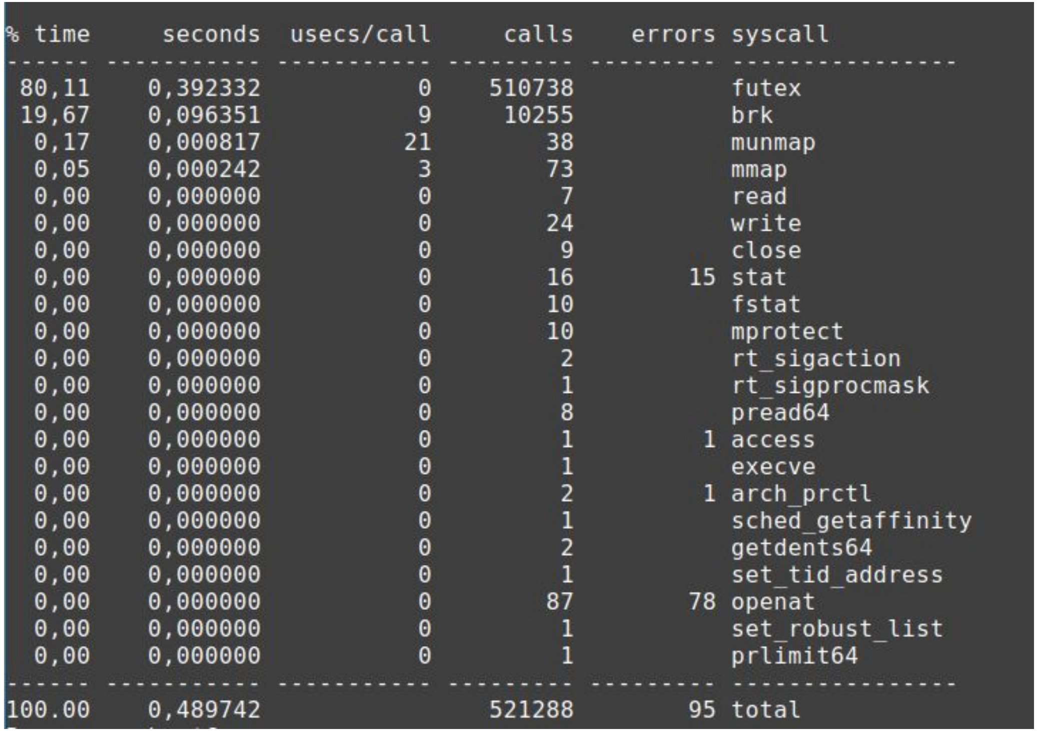


```
    1777            :
    1778            :         /*  phixi     */
    1779   25164000 :         Real_t norm = Real_t(1.) / (domain.delv_xi(ielem)+ ptiny ) ;
    1780            :
    1781   25164000 :         switch (bcMask & XI_M) {
    1782   24325200 :            case XI_M_COMM: /* needs comm data */
    1783   24325200 :            case 0:         delvm = domain.delv_xi(domain.lxim(ielem)); break ;
    1784     838800 :            case XI_M_SYMM: delvm = domain.delv_xi(ielem) ;       break ;
    1785            :            case XI_M_FREE: delvm = Real_t(0.0) ;      break ;
    1786          0 :            default:          fprintf(stderr, "Error in switch at %s line %d\n",
    1787            :                                      __FILE__, __LINE__);
    1788          0 :               delvm = 0; /* ERROR - but quiets the compiler */
    1789          0 :               break;
    1790            :         }
    1791   25164000 :         switch (bcMask & XI_P) {
    1792   24325200 :            case XI_P_COMM: /* needs comm data */
    1793   24325200 :            case 0:         delvp = domain.delv_xi(domain.lxip(ielem)) ; break ;
    1794          0 :            case XI_P_SYMM: delvp = domain.delv_xi(ielem) ;       break ;
    1795            :            case XI_P_FREE: delvp = Real_t(0.0) ;      break ;
    1796          0 :            default:          fprintf(stderr, "Error in switch at %s line %d\n",
    1797            :                                      __FILE__, __LINE__);
    1798          0 :               delvp = 0; /* ERROR - but quiets the compiler */
    1799          0 :               break;
    1800            :         }
    1801            :
```

lcov

- Hands-on in terminal
- Installation from the repositories
- Reference
  - Compile and link with coverage enabled (`--coverage` and `-g` with GCC and CLANG)
  - Run the binary
  - `lcov --capture --directory my_dir --output-file coverage.info`
  - `genhtml coverage.info --output-directory out`
  - `google-chrome out/index.html`
- More info: http://ltp.sourceforge.net/coverage/lcov.php

Kernel call tracing, e.g. strace



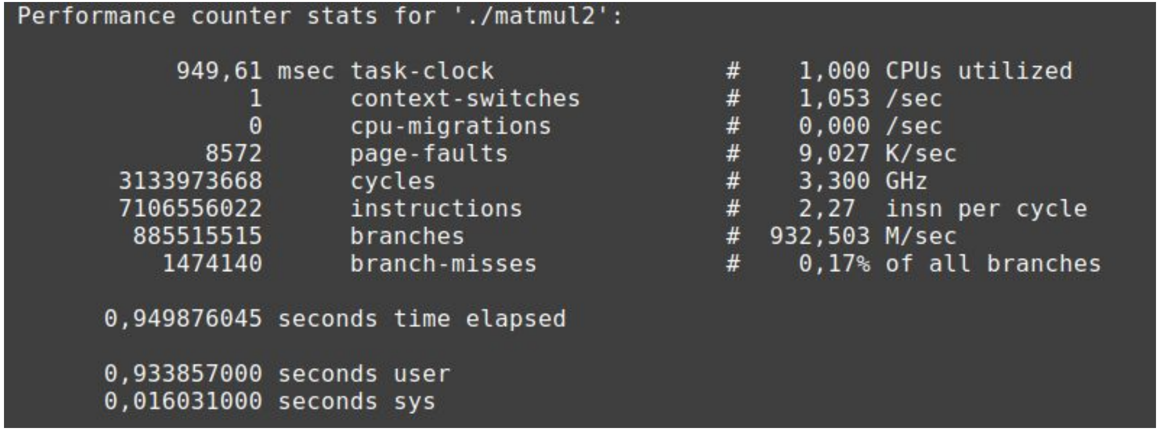
```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 80,11    0,392332           0    510738           futex
 19,67    0,096351           9     10255           brk
  0,17    0,000817          21        38           munmap
  0,05    0,000242           3        73           mmap
  0,00    0,000000           0         7           read
  0,00    0,000000           0        24           write
  0,00    0,000000           0         9           close
  0,00    0,000000           0        16        15 stat
  0,00    0,000000           0        10           fstat
  0,00    0,000000           0        10           mprotect
  0,00    0,000000           0         2           rt_sigaction
  0,00    0,000000           0         1           rt_sigprocmask
  0,00    0,000000           0         8           pread64
  0,00    0,000000           0         1         1 access
  0,00    0,000000           0         1           execve
  0,00    0,000000           0         2         1 arch_prctl
  0,00    0,000000           0         1           sched_getaffinity
  0,00    0,000000           0         2           getdents64
  0,00    0,000000           0         1           set_tid_address
  0,00    0,000000           0        87        78 openat
  0,00    0,000000           0         1           set_robust_list
  0,00    0,000000           0         1           prlimit64
------ ----------- ----------- --------- --------- ----------------
100.00    0,489742                521288        95 total
```

- You can deduce a lot of information about program by observing how it talks with the OS
- Also important if the program is spending a lot of time in the system mode

strace

- Hands on in the terminal
- Installed on most Linux’
- Reference
  - `` `strace ./my_app` `` prints all system calls
  - `` `strace -k ./my_app` `` prints all system calls with complete stack trace
  - `` `strace -c .my_app` `` statistical information about system calls, useful for debugging performance issues
- More info: https://johnysswlab.com/lessons-in-debugging-observe-how-programs-interact-with-the-linux-kernel-with-strace/
- For girls: https://jvns.ca/strace-zine-v2.pdf

Hardware counters and event counters - perf



```
Performance counter stats for './matmul2':

          949,61 msec task-clock                #    1,000 CPUs utilized
               1      context-switches          #    1,053 /sec
               0      cpu-migrations            #    0,000 /sec
            8572      page-faults               #    9,027 K/sec
      3133973668      cycles                    #    3,300 GHz
      7106556022      instructions              #    2,27  insn per cycle
       885515515      branches                  #  932,503 M/sec
         1474140      branch-misses             #    0,17% of all branches

     0,949876045 seconds time elapsed

     0,933857000 seconds user
     0,016031000 seconds sys
```

- Perf is a very powerful tool, here we limit ourselves only to the most basic information

perf

- Hands on in terminal
- Reference
  - `perf stat ./my program` - lists the basic events
  - User and system - information about time spent in user and system mode
  - Cycles - Information about spent cycles, corresponding to execution time
  - Instruction - Information about executed instructions
  - Instruction per cycle - ideally 4, but very rarely seen that number
  - List all events `perf list`
- In general: efficient instructions result in smaller cycle count, but increasing instructions increases cycle count
- For measuring on a piece of code instead of the whole program, see LIKWID: https://johnysswlab.com/hardware-performance-counters-the-easy-way-quickstart-likwid-perfctr/

Join the community!

- cppserbia.slack.com
    - Goran Aranđelović
    - Ivica Bogosavljević