Benchmarking and instrumentation

Benchmarking

```
time ./application
```

```
real    0m2.501s
user    0m2.498s
sys     0m0.001s
```

- `real`: elapsed “real” (wall-clock) time
- `user`: time spent in user mode (running `./application` code)
- `sys`: time spent in system mode (running OS kernel code)
- `user` + `sys` $\lesssim$ `real` (there may be other applications running)

```
time head -n 1000000 < /dev/random > /dev/null
```

```
real	0m0.444s
user	0m0.066s
sys	0m0.377s
```

Variance

```
time head -n 1000000 < /dev/random > /dev/null
```

```
real	0m0.442s
user	0m0.070s
sys	0m0.371s
```

```
time head -n 1000000 < /dev/random > /dev/null
```

```
real	0m0.448s
user	0m0.066s
sys	0m0.381s
```

```
time head -n 1000000 < /dev/random > /dev/null
```

```
real	0m0.445s
user	0m0.056s
sys	0m0.388s
```

Reasons for variance

- throttling for temperature and power limits

  (CPU adapts speed to avoid overheating or exceeding power supply capabilities)

- interactions with devices

  (OS has in-memory caches for files, storage devices have internal memory caches)

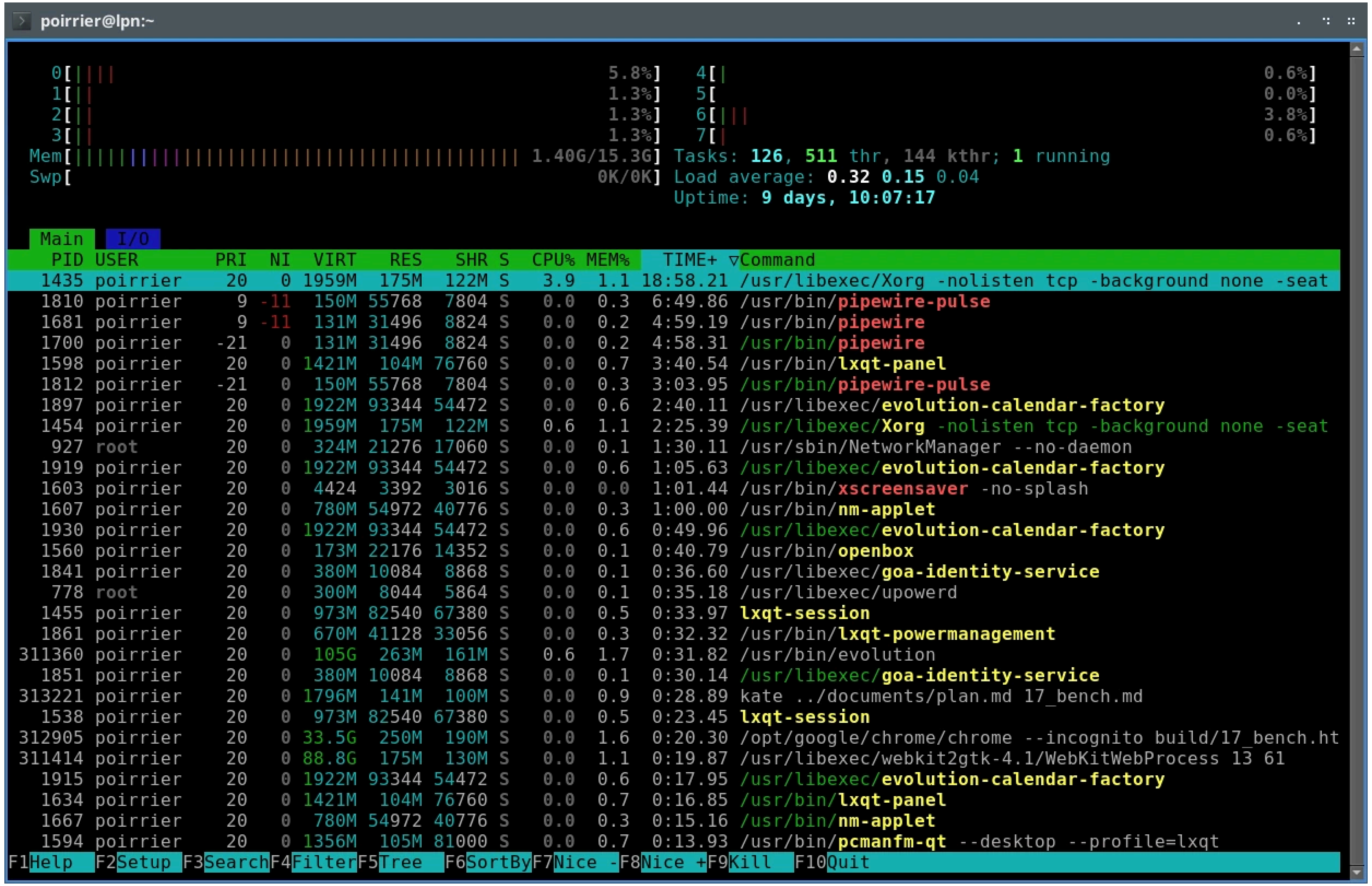
- other processes

  (must share resources)

`top`

`htop`

`ps aux`



(271 processes)

Effect of file caches

```
time md5sum 2GB_file
```

```
860a0023a913fd3fa4b6ad8bfbdd2c62  2GB_file

real	0m5.904s
user	0m4.062s
sys	0m0.560s
```

```
time md5sum 2GB_file
```

```
860a0023a913fd3fa4b6ad8bfbdd2c62  2GB_file

real	0m4.029s
user	0m3.674s
sys	0m0.331s
```

Inaccuracies

1. executable startup is slow
2. initialization adds overhead
3. input and output are slow

1. Executable startup is slow

```c
int main() { return 0; }
```

```
clang -O3 -o main main.c
time ./main
```

```
real    0m0.003s
user    0m0.000s
sys     0m0.002s
```

```
time python -c 'exit(0)'
```

```
real    0m0.030s
user    0m0.023s
sys     0m0.008s
```

→ we cannot accurately benchmark application that only take a few milliseconds.

2. Initialization adds overhead

```
time glpsol LP_576x18380.mps
```

```
real    0m0.256s
user    0m0.246s
sys     0m0.010s
```

```
time glpsol --check LP_576x18380.mps
```

```
real    0m0.168s
user    0m0.161s
sys     0m0.008s
```

What are we really measuring?

The speed of the MPS file parser, not the simplex algorithm.

3. Input and output are slow

$$\pi = \sqrt{6\,\zeta(2)} \quad \text{where} \quad \zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

```
def riemann_zeta(s):
    r = 0.0

    for i in range(1, 1000000):
        r += 1 / (i ** s)

    return r

# ζ(2) = (pi ** 2) / 6
print('pi ≈ ', (riemann_zeta(2) * 6) ** 0.5)
```

```
time python zeta.py
```

```
pi ≈  3.141591698659554

real    0m0.124s
user    0m0.118s
sys     0m0.006s
```

```python
def riemann_zeta(s):
    r = 0.0

    for i in range(1, 1000000):
        r = r + 1 / (i ** s)
        print('r = ', r)

    return r

# ζ(2) = (pi ** 2) / 6
print('pi ≈ ', (riemann_zeta(2) * 6) ** 0.5)
```

```
time python zeta.py
```

```
r =  1.0
r =  1.25
r =  1.3611111111111112
r =  1.4236111111111112
r =  1.4636111111111112
r =  1.4913888888888889
r =  1.511797052154195

       [...]

r =  1.6449330668467699
r =  1.64493306684777
pi ≈  3.141591698659554

real    0m3.768s
user    0m2.516s
sys     0m0.999s
```

Aggregate measures

- if we benchmark our code on different inputs, we may want to use
  - total time / average time
  - geometric mean
  - or other aggregate measures
  - or some visualization (bar graphs, performance profiles, etc.)

- but beware: all aggregate measures are biased

	Input 1		Input 2		Input 3		Average
Version A	2530s		2300s		12s		1614s
Version B	2535s	1.002x	2304s	1.002x	6s	0.5x	1615s

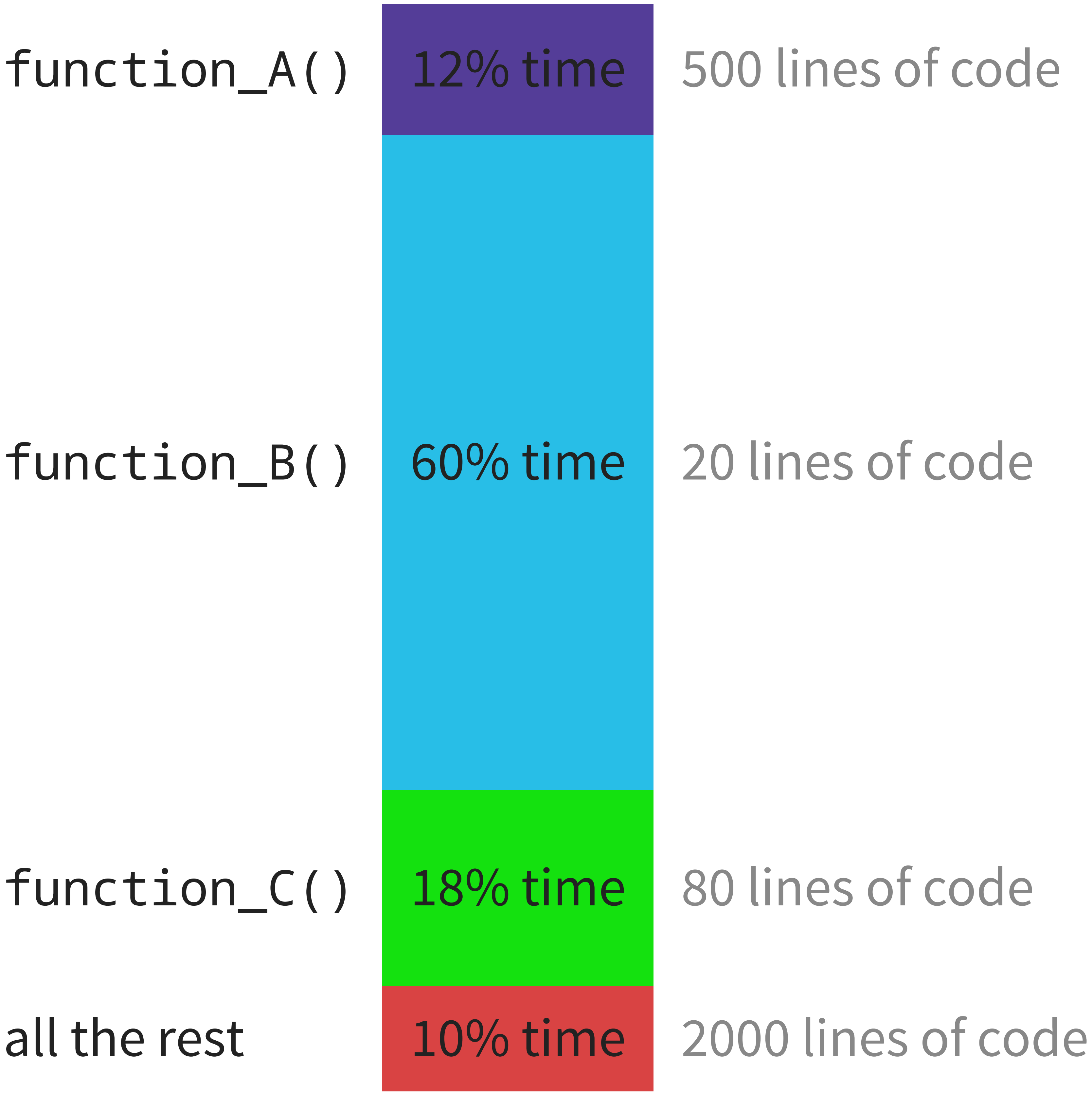
(geometric means: Version A 411.8 Version B 327.2)

Static instrumentation

- we may want to benchmark specific parts of our code
    - to circumvent executable startup, initialization, and input/output
    - to benchmark parts of the code that run quickly
    - to find bottlenecks

- for that, we need to add timing instrumentation to our code

Bottlenecks



time.time()

```
initialize()
function_A()
function_B()
function_C()
cleanup()
```

```python
import time

t0 = time.time()
initialize()

t1 = time.time()
function_A()

t2 = time.time()
function_B()

t3 = time.time()
function_C()

t4 = time.time()
cleanup()

t5 = time.time()

print(f'total time: {t5 - t0:16.6f}')

print(f'function_A: {t2 - t1:16.6f}')
print(f'function_B: {t3 - t2:16.6f}')
print(f'function_C: {t4 - t3:16.6f}')

print(f'      rest: {(t5 - t0) - (t4 - t1):16.6f}')
```

clock_gettime()

```c
int main()
{
    initialize();
    function_A();
    function_B();
    function_C();
    cleanup();
    return 0;
}
```

```c
int main()
{
    struct timespec t0, t1, t2, t3, t4, t5;

    clock_gettime(CLOCK_MONOTONIC, &t0);
    initialize();

    clock_gettime(CLOCK_MONOTONIC, &t1);
    function_A();

    clock_gettime(CLOCK_MONOTONIC, &t2);
    function_B();

    clock_gettime(CLOCK_MONOTONIC, &t3);
    function_C();

    clock_gettime(CLOCK_MONOTONIC, &t4);
    cleanup();

    clock_gettime(CLOCK_MONOTONIC, &t5);

    print_all_clocks(&t0, &t1, &t2, &t3, &t4, &t5);
    return 0;
}
```

Cumulative time

```python
initialize()

for i in range(1000000):
    function_A()
    function_B()
    function_C()

cleanup()
```

```python
import time.time

initialize()
tA, tB, tC = 0

for i in range(1000000):
    t0 = time.time()
    function_A()

    t1 = time.time()
    function_B()

    t2 = time.time()
    function_C()

    t3 = time.time()

    tA += (t1 - t0)
    tB += (t2 - t1)
    tC += (t3 - t2)

cleanup()
```

Caveat: measuring time takes time!

`time.time()`: ~40 ns (and this value fluctuates!)

Microbenchmarks

What do we do if `function_A()` takes much less time than `time.time()`?

```python
import time.time

initialize()
tA, tB, tC = 0

for i in range(1000000):
    t0 = time.time()
    function_A()

    t1 = time.time()
    function_B()

    t2 = time.time()
    function_C()

    t3 = time.time()

    tA = tA + (t1 - t0)
    tB = tB + (t2 - t1)
    tC = tC + (t3 - t2)

cleanup()
```

Microbenchmark for `function_A()`:

```python
import time.time

initialize()

t0 = time.time()

for i in range(50000000):
    function_A()

t1 = time.time()

cleanup()
```

Microbenchmarks limitations

- It may not make sense to call `function_A()` in isolation
  - Take `sin(x)` for example: which value of x do we choose?
  - Always the same?
    - Are we sure `sin(0)` takes as much time as `sin(0.1)`?
  - A random value for `x`?
    - What if generating pseudo-random values takes more time than `sin()`?
- What about caches?
  - Caches will be “hot” (already filled with relevant data)
  - Microbenchmarking presents an over-optimistic picture of memory access times

Automated instrumentation: Profilers

gprof

Add “-pg” to gcc/clang parameters

```
gcc -O3 -o app app.c -pg
```

Run the application

```
./app
```

Generate report

```
gprof app
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 63.77      3.82     3.82        1     3.82     4.24  tree_dfs
 28.88      5.55     1.73        1     1.73     1.73  lut_build
  4.17      5.80     0.25  1523737     0.00     0.00  aux_h_merge
  2.84      5.97     0.17    10331     0.00     0.00  aux_d_sort_swapper
  0.33      5.99     0.02                             tree_prune
  0.00      5.99     0.00     6715     0.00     0.00  aux_h_sort
  0.00      5.99     0.00      706     0.00     0.00  tree_gc
  0.00      5.99     0.00        6     0.00     0.00  dict_append_file
  0.00      5.99     0.00        1     0.00     0.00  dict_filter_dupes
  0.00      5.99     0.00        1     0.00     0.00  lut_hash_word
  0.00      5.99     0.00        1     0.00     0.00  solver_connected
  0.00      5.99     0.00        1     0.00     5.97  tree_build
```

- Pros
  - Easy to use
  - Exhaustive profile information
  - Generally low overhead
- Cons
  - Overhead increases when bottlenecks are in small, short functions (up to 2x runtime)
  - Limited accuracy

Hardware performance counters

The simplest hardware-aided performance-measuring tool is:

the time stamp counter (TSC)

- Introduced by Intel with the Pentium architecture (1993)
- Similar feature available on ARM since ARMv7 (1996)
- Special integer register
- Incremented by one at a constant rate (e.g. every clock cycle)
- Reading this register has high latency (>10 cycles)
- Useful for microbenchmarks and instrumentation
- `time.time() / clock_gettime()` use this internally

More complex performance counters

Since then, Intel and ARM have added many more performance counters:

- executed ("retired") instructions
- branches
  - successfully predicted
  - mispredicted branches
- memory accesses
  - found in L1 cache
  - L1 misses, found in L2 cache
  - L2 misses, found in (last-level) L3 cache
  - L3 misses, found in main memory
  - TLB (page table cache) hits
  - TLB misses

- Pros
  - always measured
  - no performance penalty
  - no interference with normal execution
- Cons
  - only an aggregate measure (totals)

Linux perf

```
perf stat ./application
```

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

          3,216.90 msec task-clock                       #    1.000 CPUs utilized
                 8      context-switches                 #    2.487 /sec
                 1      cpu-migrations                   #    0.311 /sec
             6,205      page-faults                      #    1.929 K/sec
     9,442,508,623      cycles                           #    2.935 GHz                         (52.90%)
     7,596,331,032      instructions                     #    0.80  insn per cycle              (58.81%)
     1,086,117,213      branches                         #  337.629 M/sec                       (58.84%)
         1,085,287      branch-misses                    #    0.10% of all branches             (58.87%)
     2,162,685,901      L1-dcache-loads                  #  672.289 M/sec                       (58.87%)
     1,079,393,101      L1-dcache-load-misses            #   49.91% of all L1-dcache accesses   (58.88%)
     1,069,062,732      LLC-loads                        #  332.327 M/sec                       (58.87%)
         6,537,301      LLC-load-misses                  #    0.61% of all L1-icache accesses   (23.50%)
     2,161,850,109      dTLB-loads                       #  672.029 M/sec                       (23.50%)
           896,301      dTLB-load-misses                 #    0.04% of all dTLB cache accesses  (23.50%)
         9,051,173      dTLB-stores                      #    2.814 M/sec                       (23.50%)
            81,624      dTLB-store-misses                #   25.374 K/sec                       (23.50%)

       3.217829387 seconds time elapsed

       3.167788000 seconds user
       0.022723000 seconds sys
```

Stochastic instrumentation

Previous limitations

- Static instrumentation is expensive (and affects accuracy)
- With performance counters:
  - How could we find hot spots?

    (small groups of instructions that the application spends a lot of time running)
  - What about performance counts (cache misses, mispredicted branches,...) at those hot spots?

Solution

Stochastic instrumentation:

- every N cycles (e.g. every 1,000,000th cycle / every 0.1ms), a sample is taken
- the sample records:
  - which instruction is currently being executed
  - optionally, what it is waiting for (instr. decoding, pipeline bubble, memory access, …)
  - optionally, instruction addresses of the last few branches
  - optionally, whether those branches were successfully predicted

Stochastic instrumentation

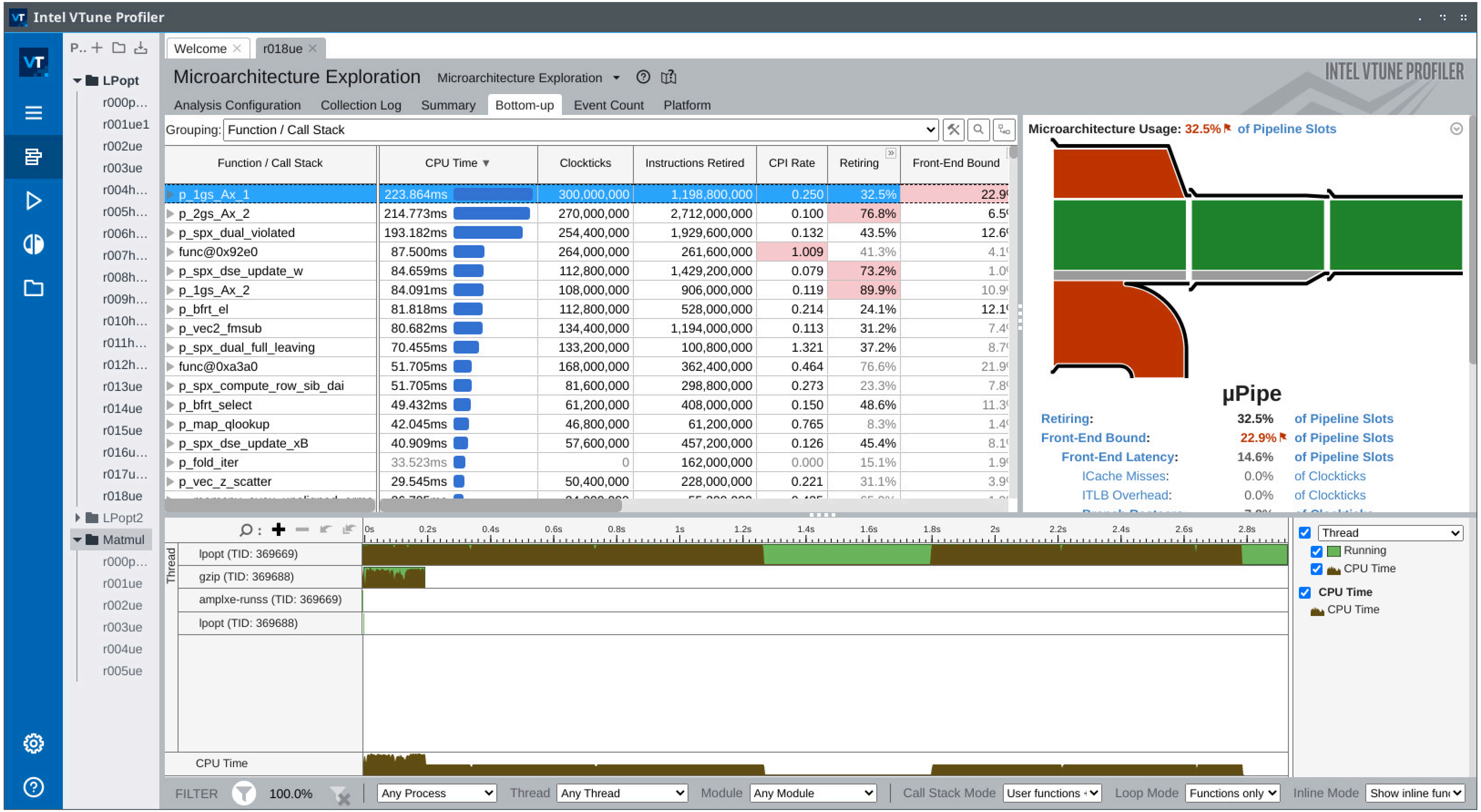
- Pros
  - no performance penalty
  - no interference with normal execution
  - accuracy naturally increases on hotspots

- Cons
  - like performance counters, needs hardware support

Analysis applications

- Linux
  - Linux perf: `perf record / perf report`
  - `KDAB hotspot`
- MacOS: Apple XCode Instruments
- Windows:
  - Visual Studio Performance Profiler
  - Windows Performance Toolkit
- Intel-specific: vTune
- AMD-specific: uProf

Bottom-up analysis



Flame graphs