

LECTURE 17

MORE CACHES AND PIPELINES

- recall that memory is **virtualized**
- a virtual address \rightarrow hardware address translation is necessary for **every memory access**
- the translation uses a **page table**
- the page table is stored in **memory**
- hence **two** effective memory accesses per memory fetch instruction?

Solution: part of the **page table** is **cached** in the CPU
the “Translation Lookaside Buffer” (TLB)

Caches are used at various levels to hide access latency

	typical latency		
instruction	~0.25 ns	0.25	ns
RAM	~100 ns	100	ns
solid state drive (SSD)	~0.2 ms	200,000	ns
hard disk drive (HDD)	~2 ms	2,000,000	ns
wired ethernet (round-trip)	~1 ms	1,000,000	ns
wifi latency (round-trip)	~10 ms	10,000,000	ns
same-city internet (round-trip)	~5 ms	5,000,000	ns
same-continent internet (round-trip)	~25 ms	25,000,000	ns
transatlantic internet (round-trip)	~100 ms	100,000,000	ns

Examples of caches

- SSDs have internal RAM caches (typically 0-4 GB)
- the operating system caches files in memory
- large content providers (Google, Amazon, Netflix, Cloudflare) have caches all over the world

```
ping canada.ca
```

```
PING canada.ca (205.193.117.159) 56(84) bytes of data.  
64 bytes from 205.193.117.159 (205.193.117.159): icmp_seq=1 ttl=228 time=200 ms  
64 bytes from 205.193.117.159 (205.193.117.159): icmp_seq=2 ttl=228 time=172 ms  
64 bytes from 205.193.117.159 (205.193.117.159): icmp_seq=3 ttl=228 time=148 ms  
64 bytes from 205.193.117.159 (205.193.117.159): icmp_seq=4 ttl=228 time=181 ms
```

```
ping google.com.au
```

```
PING google.com.au (142.251.209.3) 56(84) bytes of data.  
64 bytes from mil04s50-in-f3.1e100.net (142.251.209.3): icmp_seq=1 ttl=115 time=12.2 ms  
64 bytes from mil04s50-in-f3.1e100.net (142.251.209.3): icmp_seq=2 ttl=115 time=14.6 ms  
64 bytes from mil04s50-in-f3.1e100.net (142.251.209.3): icmp_seq=3 ttl=115 time=12.9 ms  
64 bytes from mil04s50-in-f3.1e100.net (142.251.209.3): icmp_seq=4 ttl=115 time=11.8 ms
```

Examples of pipelines

- Storage devices:
 - SSDs typically access data in “pages” of **4096 bytes**
 - **0.2ms** SSD latency would imply a max speed of **20 MB / s**
 - instead SSDs routinely read and write **500 MB / s**
- Networks:
 - Network packets are typically **1500 bytes**
 - **10ms** WiFi latency would imply **150 KB / s**
 - instead most WiFi networks do at least **10 MB / s**
- Browsers:
 - Google Chrome maintains up to **6** connections per domain

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    0m0.777s  
user    0m0.109s  
sys     0m0.668s
```

Variance

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

```
real    0m2.502s  
user    0m2.497s  
sys     0m0.003s
```

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

```
real    0m2.505s  
user    0m2.500s  
sys     0m0.002s
```

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

```
real    0m2.501s  
user    0m2.496s  
sys     0m0.001s
```

Reasons for variance

- power and temperature throttling
(CPU adapts speed to avoid overheating)
- 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

```

> poirrier@lpn:~
 0 [||||| 5.8%] 4 [||||| 0.6%]
 1 [||||| 1.3%] 5 [||||| 0.0%]
 2 [||||| 1.3%] 6 [||||| 3.8%]
 3 [||||| 1.3%] 7 [||||| 0.6%]
Mem[||||| 1.40G/15.3G] Tasks: 126, 511 thr, 144 kthr; 1 running
Swp[||||| 0K/0K] Load average: 0.32 0.15 0.04
Uptime: 9 days, 10:07:17

Main I/O
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
1435 poirrier 20 0 1959M 175M 122M S 3.9 1.1 18:58.21 /usr/libexec/Xorg -nolisten tcp -background none -seat
1810 poirrier 9 -11 150M 55768 7804 S 0.0 0.3 6:49.86 /usr/bin/pipewire-pulse
1681 poirrier 9 -11 131M 31496 8824 S 0.0 0.2 4:59.19 /usr/bin/pipewire
1700 poirrier -21 0 131M 31496 8824 S 0.0 0.2 4:58.31 /usr/bin/pipewire
1598 poirrier 20 0 1421M 104M 76760 S 0.0 0.7 3:40.54 /usr/bin/lxqt-panel
1812 poirrier -21 0 150M 55768 7804 S 0.0 0.3 3:03.95 /usr/bin/pipewire-pulse
1897 poirrier 20 0 1922M 93344 54472 S 0.0 0.6 2:40.11 /usr/libexec/evolution-calendar-factory
1454 poirrier 20 0 1959M 175M 122M S 0.6 1.1 2:25.39 /usr/libexec/Xorg -nolisten tcp -background none -seat
 927 root 20 0 324M 21276 17060 S 0.0 0.1 1:30.11 /usr/sbin/NetworkManager --no-daemon
1919 poirrier 20 0 1922M 93344 54472 S 0.0 0.6 1:05.63 /usr/libexec/evolution-calendar-factory
1603 poirrier 20 0 4424 3392 3016 S 0.0 0.0 1:01.44 /usr/bin/xscreensaver -no-splash
1607 poirrier 20 0 780M 54972 40776 S 0.0 0.3 1:00.00 /usr/bin/nm-applet
1930 poirrier 20 0 1922M 93344 54472 S 0.0 0.6 0:49.96 /usr/libexec/evolution-calendar-factory
1560 poirrier 20 0 173M 22176 14352 S 0.0 0.1 0:40.79 /usr/bin/openbox
1841 poirrier 20 0 380M 10084 8868 S 0.0 0.1 0:36.60 /usr/libexec/goa-identity-service
 778 root 20 0 300M 8044 5864 S 0.0 0.1 0:35.18 /usr/libexec/upowerd
1455 poirrier 20 0 973M 82540 67380 S 0.0 0.5 0:33.97 lxqt-session
1861 poirrier 20 0 670M 41128 33056 S 0.0 0.3 0:32.32 /usr/bin/lxqt-powermanagement
311360 poirrier 20 0 105G 263M 161M S 0.6 1.7 0:31.82 /usr/bin/evolution
1851 poirrier 20 0 380M 10084 8868 S 0.0 0.1 0:30.14 /usr/libexec/goa-identity-service
313221 poirrier 20 0 1796M 141M 100M S 0.0 0.9 0:28.89 kate ../documents/plan.md 17_bench.md
1538 poirrier 20 0 973M 82540 67380 S 0.0 0.5 0:23.45 lxqt-session
312905 poirrier 20 0 33.5G 250M 190M S 0.0 1.6 0:20.30 /opt/google/chrome/chrome --incognito build/17_bench.ht
311414 poirrier 20 0 88.8G 175M 130M S 0.0 1.1 0:19.87 /usr/libexec/webkit2gtk-4.1/WebKitWebProcess 13 61
1915 poirrier 20 0 1922M 93344 54472 S 0.0 0.6 0:17.95 /usr/libexec/evolution-calendar-factory
1634 poirrier 20 0 1421M 104M 76760 S 0.0 0.7 0:16.85 /usr/bin/lxqt-panel
1667 poirrier 20 0 780M 54972 40776 S 0.0 0.3 0:15.16 /usr/bin/nm-applet
1594 poirrier 20 0 1356M 105M 81000 S 0.0 0.7 0:13.93 /usr/bin/pcmanfm-qt --desktop --profile=lxqt
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice - F8Nice + F9Kill F10Quit

```

(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

- executable startup is slow
- initialization adds overhead
- input and output are slow

Executable startup is slow

```
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.

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.

Input and output are slow

```
def riemann_zeta(s):  
    r = 0.0  
  
    for i in range(1, 1000000):  
        r = r + 1 / (i ** s)  
  
    return r  
  
#  $\zeta(2) = (\pi^2) / 6$   
print('pi ≈ ', (riemann_zeta(2) * 6) ** 0.5)
```

```
time python zeta.py
```

```
pi ≈ 3.141591698659554
```

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

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

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

    return r

#  $\zeta(2) = (\pi^2) / 6$ 
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

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

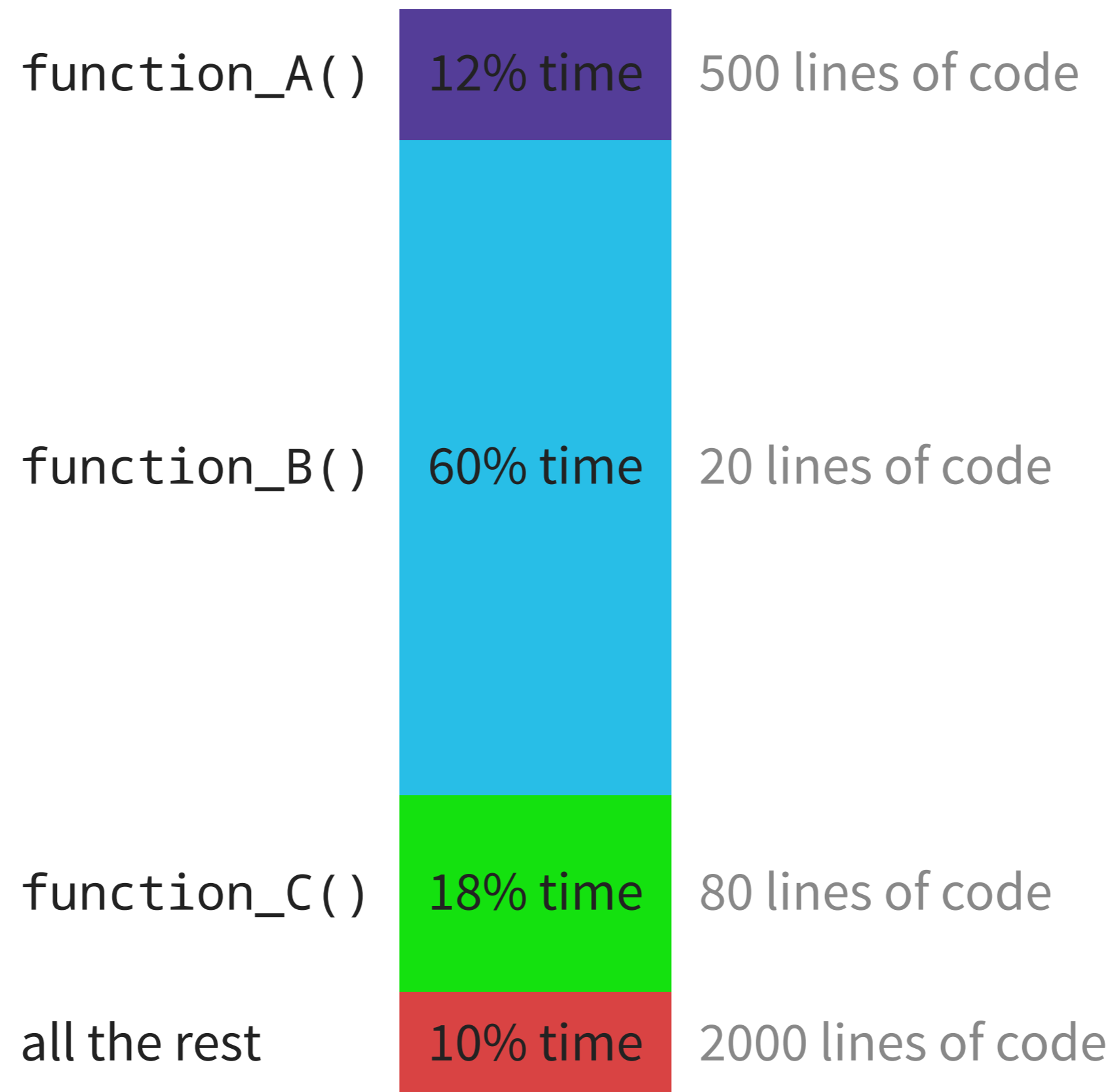
About bottlenecks

“Premature optimization is the root of all evil”

– Donald Knuth

“Structured Programming With GoTo Statements”

1974



time.time()

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

```
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()

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

```
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

```
initialize()

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

cleanup()
```

```
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()
```

Caveat: measuring time takes time!

`time.time()`: ~40 ns (and the actual time fluctuates)

Microbenchmarks

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

```
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()`:

```
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.

% time	cumulative seconds	self seconds	calls	self s/call	total 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

```

index % time  self  children  called  name
[1]    99.7    0.00    5.97     1/1     main [2]
         0.00    5.97     1       tree_build [1]
         3.82    0.42     1/1     tree_dfs [3]
         1.73    0.00     1/1     lut_build [4]
         0.00    0.00     1/10331  aux_d_sort_swapper [5]
         0.00    0.00     1/1     lut_hash_word [12]
         0.00    0.00     1/6     dict_append_file [10]
-----
[2]    99.7    0.00    5.97     <spontaneous>
         0.00    5.97     1/1     main [2]
         0.00    0.00     1/1     tree_build [1]
         0.00    0.00     1/1     solver_connected [13]
-----
[3]    70.8    3.82    0.42     4174     tree_dfs [3]
         3.82    0.42     1/1     tree_build [1]
         0.17    0.25    1+4174  tree_dfs [3]
         0.00    0.00    10330/10331  aux_d_sort_swapper [5]
         0.00    0.00    6715/6715  aux_h_sort [8]
         0.00    0.00    706/706  tree_gc [9]
         0.00    0.00    4174     tree_dfs [3]

```

- **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

STOCHASTIC INSTRUMENTATION

