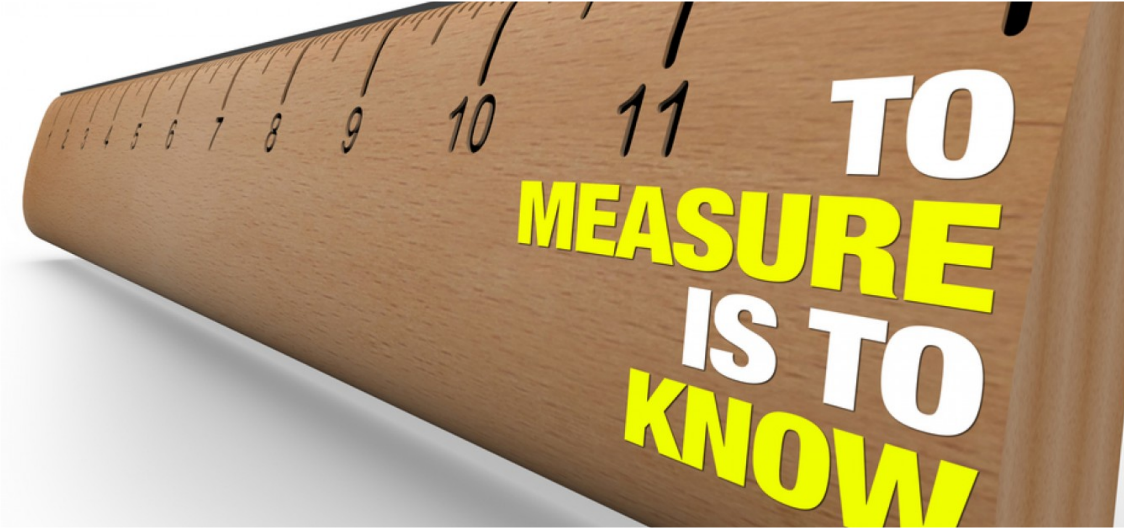
High Performance Co$^{\textit{mputing}}_{\textit{ding}}$ (HPC)

Alberto Dassatti - 2023

What is going on?

You need to measure!

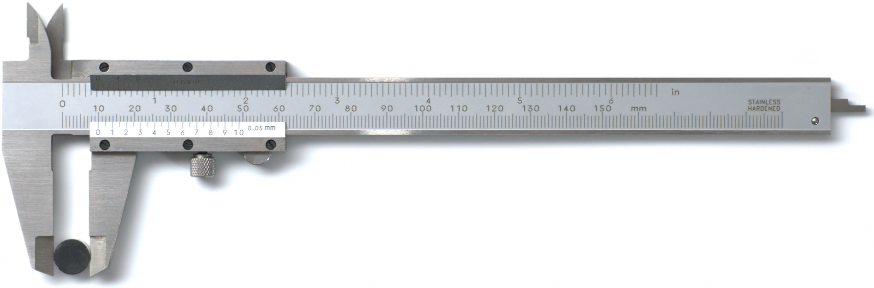

1
2
3
4
5
6
7
8
9
10
11
TO
MEASURE
IS TO
KNOW

“In God we trust. All others must bring data”.

W. Edwards Deming

Improvement

Every measure has its own tool. CS is not different.


0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 mm
1 2 3 4 5 6 in
0 1 2 3 4 5 6 7 8 9 10 0.05 mm
STAINLESS
HARDENED

And measuring correctly is even harder.

Measuring time!

The basics: Time

bash

```
[al@lap ~]# time your_command
[al@lap ~]# <your_command output omitted>
[al@lap ~]# real    0m3.168s
[al@lap ~]# user    0m2.952s
[al@lap ~]# sys     0m0.180s
```

Let’s test hyperfine.

THE BASICS: TIME

Can we have a little more visibility?

bash

```
[al@lap ~]# strace -tt
```

bash

```
[al@lap ~]# ltrace -tt
```

The basics: Time

If you have access at the source code and the toolchain, some more options are available.

bash

```
[al@lap ~]# gcc -pg -O3 -Wall source.c -o source
[al@lap ~]# ./source
[al@lap ~]# gprof source gmon.out > analysis.txt
```

With the help of `-pg` there is also the option of uftrace

Manual Instrumentation

Gcc can help.

rdtsc

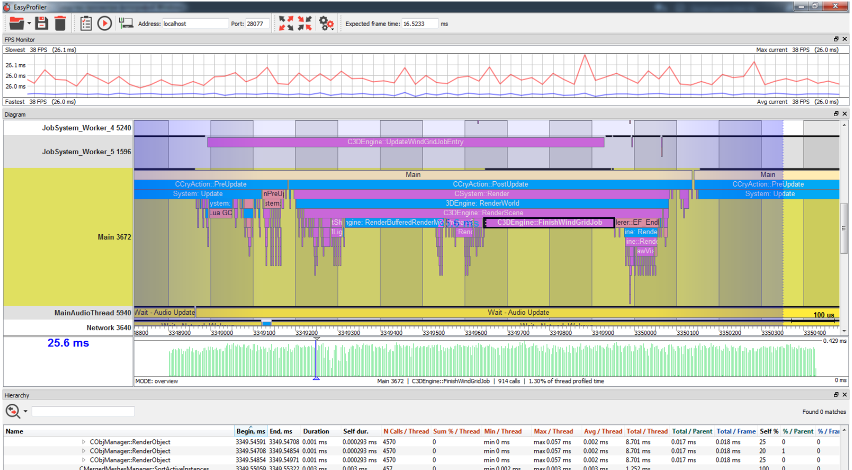
```
static __inline__ unsigned long long rdtsc(void)
{
  unsigned long long int x;
  __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
  return x;
}
```

source

cpucycles is a library that improve a lot on this raw technique.

Manual Instrumentation on steroids

You are not forced to trace all the code, but if you know your critical path you can use an instrumentation library (eg. easy_profiler or Likwid marker API).



Address: localhost Port: 28077 Expected frame time: 16.5233 ms

Slowest 38 FPS (26.1 ms) Max current 38 FPS (26.0 ms)

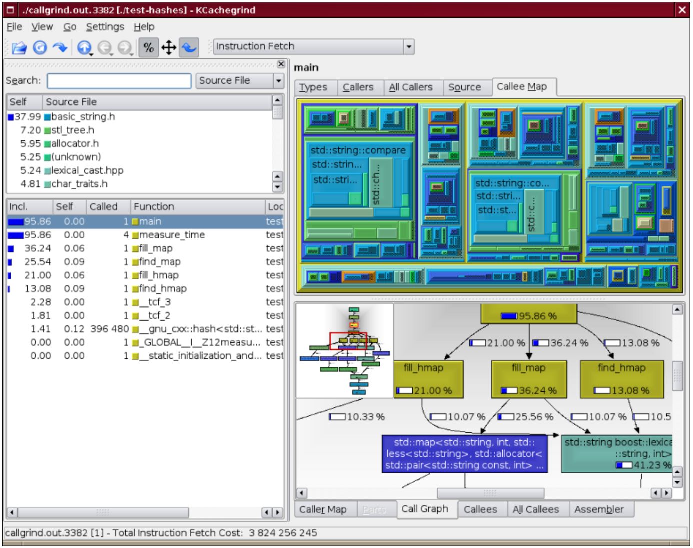
Fastest 38 FPS (26.0 ms) Avg current 38 FPS (26.0 ms)

MODE: overview Main 3672 | C3DEngine::FinishWindGridJob | 914 calls | 1.30% of thread profiled time

Found 0 matches

Name	Begin, ms	End, ms	Duration	Self dur.	N Calls / Thread	Sum % / Thread	Min / Thread	Max / Thread	Avg / Thread	Total / Thread	Total / Parent	Total / Frame	Self %	% / Parent	% / Frai
CObjManager::RenderObject	3349.54591	3349.54708	0.001 ms	0.000293 ms	4570	0	min 0 ms	max 0.057 ms	0.002 ms	8.701 ms	0.017 ms	0.018 ms	25	0	0
CObjManager::RenderObject	3349.54708	3349.54854	0.001 ms	0.000293 ms	4570	0	min 0 ms	max 0.057 ms	0.002 ms	8.701 ms	0.017 ms	0.018 ms	20	1	0
CObjManager::RenderObject	3349.54854	3349.54971	0.001 ms	0.000293 ms	4570	0	min 0 ms	max 0.057 ms	0.002 ms	8.701 ms	0.017 ms	0.018 ms	25	0	0
CMergedMeshesManager::SortActiveInstances	3349.55059	3349.55322	0.003 ms	0.003 ms	457	0	min 0.002 ms	max 0.003 ms	0.003 ms	1.252 ms			100	0	0

Run it on an emulator: Callgrind, cachegrind



./callgrind.out.3382 [./test-hashes] - KCachegrind

Search: Source File

Self	Source File
37.99	basic_string.h
7.20	stl_tree.h
5.95	allocator.h
5.25	(unknown)
5.24	lexical_cast.hpp
4.81	char_traits.h

Incl.	Self	Called	Function	Loc
95.86	0.00	1	main	test
95.86	0.00	4	measure_time	test
36.24	0.06	1	fill_map	test
25.54	0.09	1	find_map	test
21.00	0.06	1	fill_hmap	test
13.08	0.09	1	find_hmap	test
2.28	0.00	1	__tcf_3	test
1.81	0.00	1	__tcf_2	test
1.41	0.12	396 480	__gnu_cxx::hash<std::st...	test
0.00	0.00	1	_GLOBAL__I__Z12measu...	test
0.00	0.00	1	__static_initialization_and...	test

callgrind.out.3382 [1] - Total Instruction Fetch Cost: 3 824 256 245

a good tutorial

Warning and lies

Understand the numbers provided by the tools is not always easy. Please read this article and this discussion to be sure you get the most accurate info from your tool. I strongly reccomend you this fun and instructive video.

What our system is doing?

What our system is doing

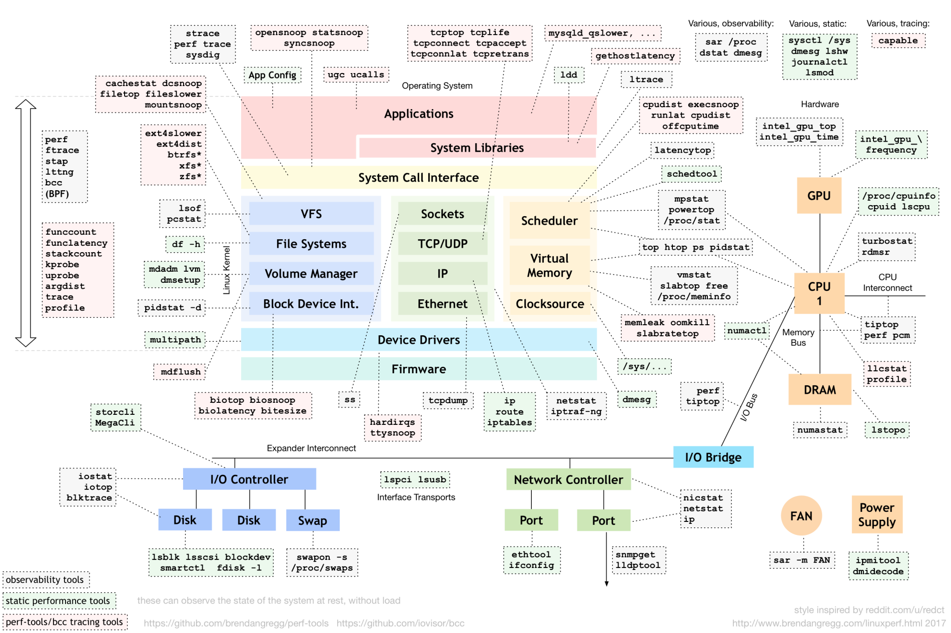
For a first look we may use

bash

```
[al@lap ~]# top
```

`htop` is a feature packed alternative

Linux Performance Tools



these can observe the state of the system at rest, without load

https://github.com/brendangregg/perf-tools https://github.com/iovisor/bcc

style inspired by reddit.com/u/redct

http://www.brendangregg.com/linuxperf.html 2017

Up to date tools catalog can be found here and here.

OK, BUT WHY IT IS TAKING SO LONG?

Workload Characterization!

micro Benchmarking

Isolate a small kernel or a function and analyze it alone

vs

System Benchmarking

Looking at the performance of a system fully in service

Statistical

Periodically stop the system and look at its state. If you collect enough data you can have a good picture of the situation.

- Advantage: do not need any code modification
- Warning: can be missleading, use long runs
- Example: `perf`

vs

Instrumentation

Modify your code manually or automatically to emit `events` and then process them

- Advantage: tracing can be very precise
- Warning: you cannot see what's outside of your code or what your input is not stimulating
- Example: `gprof`

Time-based sampling (TBS)

Program a timer. When the timer ends look at the `pc` and at the *stack* to get useful information.

- It's useful to have the debug symbols (-g)
- Trade sampling frequency for accuracy (limit the overhead)
- Some more ideas (`-fno-exceptions`, `-fno-rtti`, `-fno-omit-frame-pointer`)

vs

Event-based sampling (EBS)

Count some events (HW or SW generated) and try to get what's going on.

- Low overhead (Intel $< 2\%$ using performance monitor counters)
- Use event ratios (cache miss/ cache access)

Performance Counters

oprofile, perf, VTune

- Total instruction count and mix
- Branch events
- Load/store events
- L1/L2 cache events
- Prefetching events
- TLB events
- Multicore events

Use event ratios

```
sudo perf list
```

Perf Tutorial

blog, blog, pdf, Cern

Perf goodies

- Pmc-cloud-tools
- HotSpot
- Flamegraph

Perf is awsome

but, what if what you are interested in is not already an event or you want more visibility?

Dynamic instrumentation is the answer!

UPROBE AND KPROBE

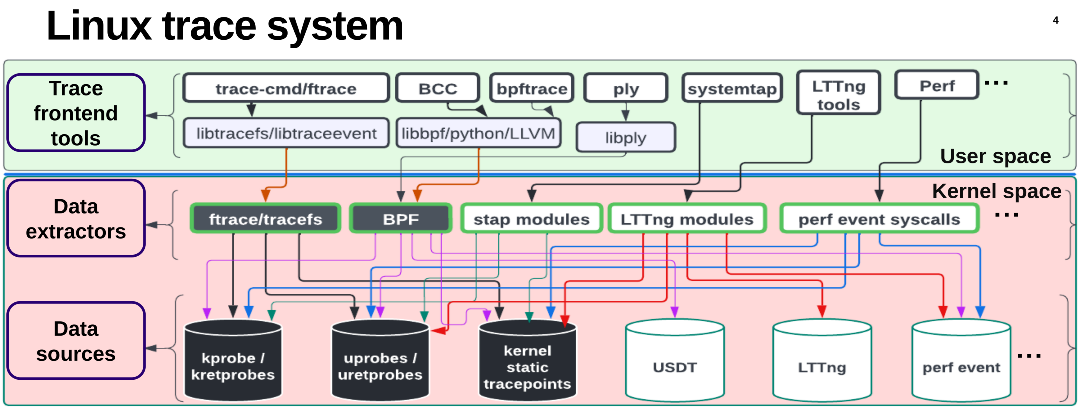
Since long time (3.15 at least), Linux has the infrastructure to dynamically add trace point in both kernel space (kprobes) and user space (uprobe).

They are not usually used directly. `perf probe` can help.

Tracing

Bootlin Debugging-slides from pg 144

Behind the scene



Linux trace system

4

From Here.

TRACING

ftrace
perf_events
eBPF
SystemTap
LTTng
ktap
dtrace4linux
OEL DTrace
sysdig

[read this]

To know more:

- have a look here
- read carefully and understand these Methods
- dig this mine of information

Questions

?