Lecture 28 — Liar, Liar

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 3, 2022

Let's open with a video that illustrates one of the problems with sampling-based profiling:

```
https://www.youtube.com/watch?v=jQDjJRYmeWg
```

Is this fake?

The main assumptions underlying sampling are that:

Samples are "random"; and

The sample distribution approximates the actual time-spent distribution.

Who can we trust?

Part II

Lies from Metrics

While app-specific metrics can lie too,
mostly we'll talk about CPU perf counters.

Reference: Paul Khuong,
`http://www.pvk.ca/Blog/2014/10/19/performance-optimisation-~-writing-an-essay/`

We've talked about `mfence`.
Used in spinlocks, for instance.

Profiles said: spinlocking didn't take much time.
Empirically: eliminating spinlocks = better than expected!

Next step: create microbenchmarks.

Memory accesses to uncached locations,
or computations,

surrounded by store pairs/mfence/locks.

Use perf to evaluate impact of mfence vs lock.

perf for lock

```
$ perf annotate -s cache_misses
[...]
    0.06 :        4006b0:       and     %rdx,%r10
    0.00 :        4006b3:       add     $0x1,%r9
    ;; random (out of last level cache) read
    0.00 :        4006b7:       mov     (%rsi,%r10,8),%rbp
   30.37 :        4006bb:       mov     %rcx,%r10
    ;; foo is cached, to simulate our internal lock
    0.12 :        4006be:       mov     %r9,0x200fbb(%rip)
    0.00 :        4006c5:       shl     $0x17,%r10
    [... Skipping arithmetic with < 1% weight in the profile]
    ;; locked increment of an in-cache "lock" byte
    1.00 :        4006e7:       lock incb 0x200d92(%rip)
   21.57 :        4006ee:       add     $0x1,%rax
    [...]
    ;; random out of cache read
    0.00 :        400704:       xor     (%rsi,%r10,8),%rbp
   21.99 :        400708:       xor     %r9,%r8
    [...]
    ;; locked in-cache decrement
    0.00 :        400729:       lock decb 0x200d50(%rip)
   18.61 :        400730:       add     $0x1,%rax
    [...]
    0.92 :        400755:       jne     4006b0 <cache_misses+0x30>
```

Reads take 30 + 22 = 52% of runtime
Locks take 19 + 21 = 40%.

```
$ perf annotate -s cache_misses
[...]
    0.00 :        4006b0:      and     %rdx,%r10
    0.00 :        4006b3:      add     $0x1,%r9
    ;; random read
    0.00 :        4006b7:      mov     (%rsi,%r10,8),%rbp
   42.04 :        4006bb:      mov     %rcx,%r10
    ;; store to cached memory (lock word)
    0.00 :        4006be:      mov     %r9,0x200fbb(%rip)
    [...]
    0.20 :        4006e7:      mfence
    5.26 :        4006ea:      add     $0x1,%rax
    [...]
    ;; random read
    0.19 :        400700:      xor     (%rsi,%r10,8),%rbp
   43.13 :        400704:      xor     %r9,%r8
    [...]
    0.00 :        400725:      mfence
    4.96 :        400728:      add     $0x1,%rax
    0.92 :        40072c:      add     $0x1,%rax
    [...]
    0.36 :        40074d:      jne     4006b0 <cache_misses+0x30>
```
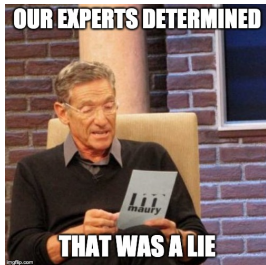
Looks like the reads take 85% of runtime,
while the mfence takes 15% of runtime.

Must also look at total # of cycles.

No atomic/fence: 2.81e9 cycles
lock inc/dec: 3.66e9 cycles
mfence: 19.60e9 cycles

That 15% number is a total lie.

- mfence underestimated;
- lock overestimated.

Why?

mfence = pipeline flush,
costs attributed to instructions being flushed.

Part III

The Long Tail

Suppose we have a task that's going to get distributed over multiple computers (like a search).

If we look at the latency distribution, the problem is mostly that we see a long tail of events.

When we are doing a computation or search where we need all the results, we can only go as the slowest step.

CPU profile shows the 99% common cases
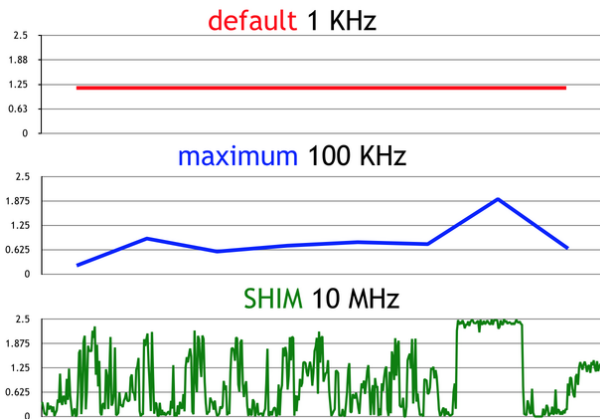
Latency 99th = 696 msec

The 1% tail nev shows
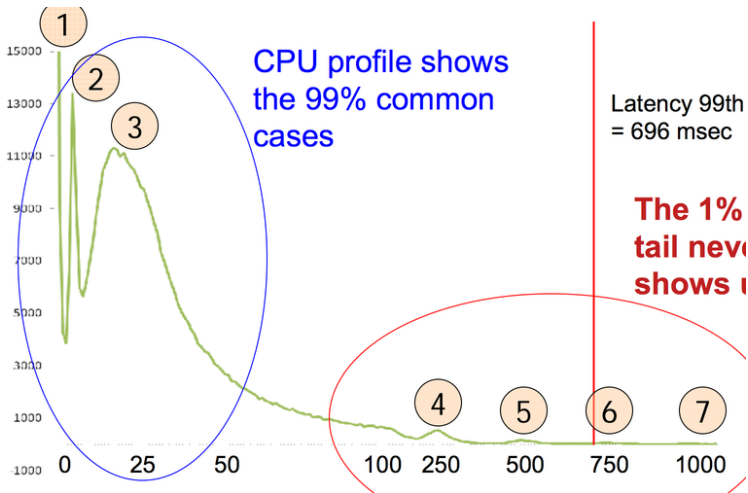
1. Found in RAM
2. Disk Cache
3. Disk
4. and above... very strange!

Answer: CPU throttling!

This was happening on 25% of disk servers at Google, for an average of half an hour a day!

Another problem with sampling, this time from Lucene:

Why is perf limited to 100 KHz?

Answer: perf samples are done with interrupts (slow).

If you crank up the rate of interrupts, before long, you are spending all your time handling the interrupts rather than doing useful work.

SHIM gets around this by being more invasive.

This produces a bunch of data which can be dealt with later.

Part IV

Lies from Counters

Rust compiler hackers were trying to include support for hardware performance counters (what perf reports).

To make counters as deterministic as possible:

- disable Address Space Layout Randomization (randomized pointer addresses affect hash layouts);
- subtract time spent processing interrupts (IRQs);
- profile one thread only (if you can, in your context).

E dith **B** inch @eddyb_r · Nov 4

AMD: we made a really clever speculative thing that keeps executing instructions normally even

me: okay but what do you do when it fails

AMD: 😅it uhhh rolls back time to the point of divergence😅

me: cool, cool, but did you remember you also roll back the perf counters?

AMD: 😬

💬 2   ⟲ 2   ♡ 30   ⬆

Part V

Lies about Calling Context

This part is somewhat outdated now, as it's a pretty specific technical problem that especially arises under the `gprof` tool.

It's still a good example of lying tools, though.

Reference: Yossi Kreinin,
`http://www.yosefk.com/blog/how-profilers-lie-the-cases-of-gprof-and-kcachegrind.html`

gprof uses two C standard-library functions: **profil()** and **mcount()**.

- **profil()**: asks glibc to record which instruction is currently executing ($100\times$/second).
- **mcount()**: records call graph edges; called by -pg instrumentation.

Hence, **profil** information is statistical, while **mcount** information is exact.

gprof can draw unreliable inferences.

If you have a method easy and a method hard, each of which is called once, and hard takes up almost all the CPU time...

gprof might divide total time by 2 and report bogus results.

What's Wrong?

The following results from gprof are suspect (among others):

- contribution of children to parents;
- total runtime spent in self+children;

When are call graph edges right? Two cases:

- functions with only one caller (e.g. `f()` only called by `g()`); or,
- functions which always take the same time to complete (e.g. `rand()`).

Next, we'll talk about callgrind/KCacheGrind.

KCacheGrind is a frontend to callgrind. callgrind gives better information, but imposes more overhead.

KCacheGrind works properly on the earlier hard/easy example, but we can still deceive it with more complicated examples.

Some results are exact;
some results are sampled;
some results are interpolated.

If you understand the tool,
you understand where it can go wrong.

Understand your tools!