Lecture 27 — Liar, Liar, Memory Profiling

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

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2022

Liar, Liar

Sampling Based Profiling

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?

Assumptions

The main assumptions underlying sampling are that:

Samples are “random”; and

The sample distribution approximates the actual time-spent distribution.

In the Criminal Justice System...

Who can we trust?

Part II

Lies from Metrics

Lying perf counters

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/

mfence

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

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

Exploring the lie

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>
```

lock's effects

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

perf for mfence

```
$ 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>
```

mfence's effects

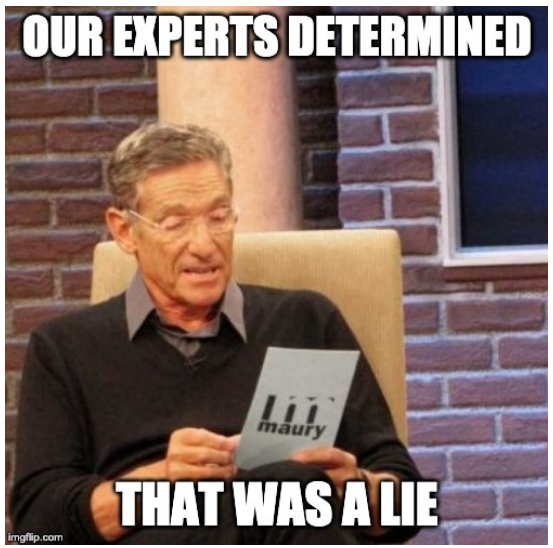
Looks like the reads take 85% of runtime,
while the mfence takes 15% of runtime.

Bigger picture

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.



Conclusions

- mfence underestimated;
- lock overestimated.

Why?

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

Part III

The Long Tail

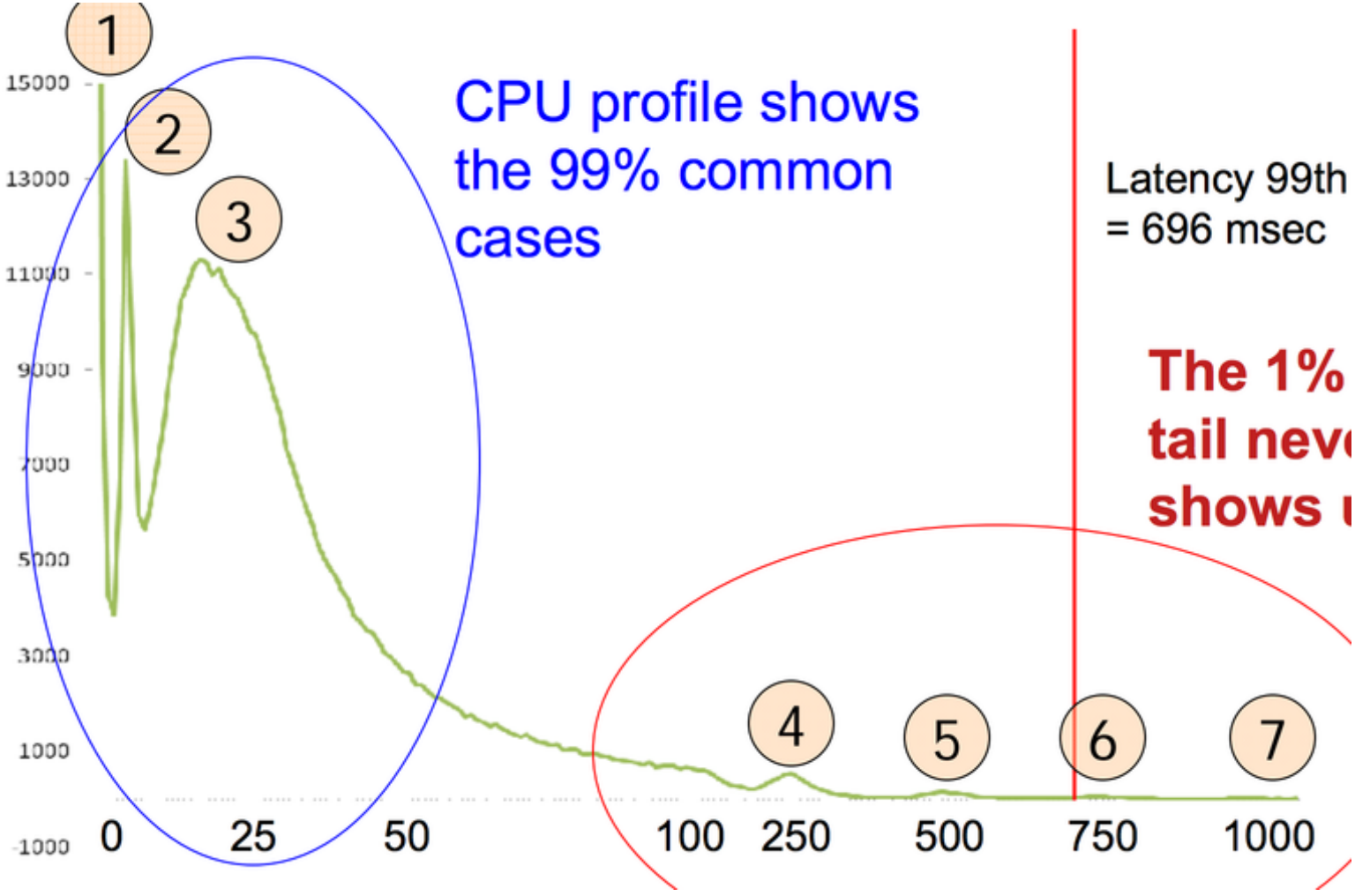
The Long Tail

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.

Grab the Tiger by the Tail



Explaining the Peaks

1. Found in RAM
2. Disk Cache
3. Disk
4. and above... very strange!

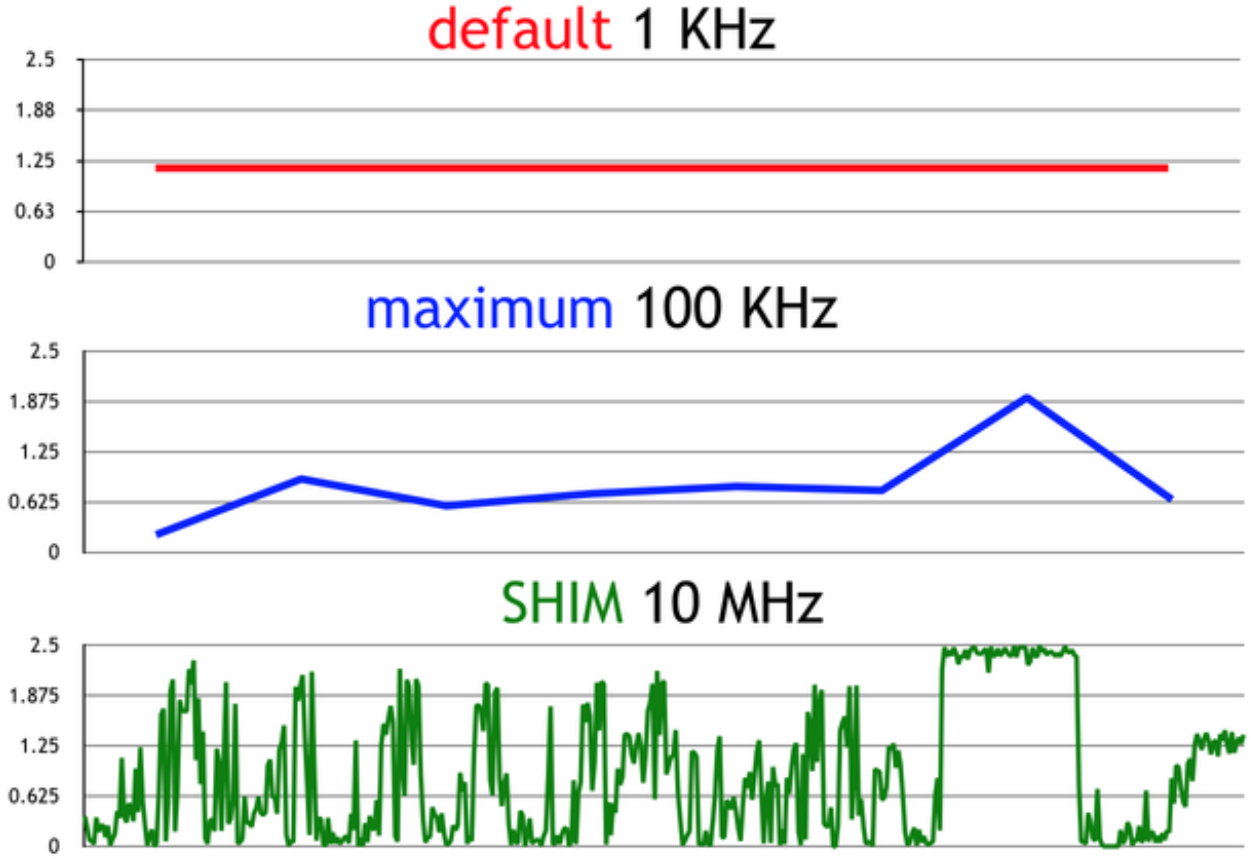
Why 250, 500, 750, 1000?

Answer: CPU throttling!

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

Faster than a Speeding Bullet

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

Counter This

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).

Problem?

Edith Binch @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

Profilers, Lying?

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 said what?

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.

Getting it Wrong

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()`).

Other Tools

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.

Lies summary

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!

Part VI

Memory Profiling

~~Memory Profiling~~ Return to Asgard

So far: CPU profiling.

Memory profiling is also a thing;
specifically heap profiling.

“Still Reachable”: not freed & still have pointers,
but should have been freed?

~~Memory Profiling~~ Return to Asgard

As with queueing theory:

$$\text{allocs} > \text{frees} \implies \text{usage} \to \infty$$

At least more paging, maybe total out-of-memory.

But! Memory isn't really lost: we could free it.

Our tool for this comes from the Valgrind tool suite.

DHAT

DHAT tracks allocation points and what happens to them over a program's execution.

An allocation point is a point in the program which allocates memory.

Leaf Node

```
AP 1.1.1.1/2 {
  Total:     31,460,928 bytes (2.32%, 1,565.9/Minstr) in 262,171 blocks (4.41%, 13.05/Minstr
     avg size 120 bytes, avg lifetime 986,406,885.05 instrs (4.91% of program duration)
  Max:       16,779,136 bytes in 65,543 blocks, avg size 256 bytes
  At t-gmax: 0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:     5,964,704 bytes (0.11%, 296.88/Minstr), 0.19/byte
  Writes:    10,487,200 bytes (0.51%, 521.98/Minstr), 0.33/byte
  Allocated at {
    ^1: 0x95CACC9: alloc (alloc.rs:72)
      [omitted]
    ^7: 0x95CACC9: parse_token_trees_until_close_delim (tokentrees.rs:27)
    ^8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
    ^9: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
    ^10: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
    #11: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
    #12: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::
                          StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
  }
}
```

Parents

Going up the tree, DHAT tells you about all of the allocation points that share a call-stack prefix.

The ^ before the number indicates that the line is copied from the parent.

A # indicates that the line is unique to that node.

A sibling of the example above would diverge at the call on line 10.

Shieldmaiden to Thor

What does Massif do?

- How much heap memory is your program using?
- How did this happen?

Next up: example from Massif docs.

Example Allocation Program

```
fn g() {
    let a = Vec::<u8>::with_capacity(4000);
    std::mem::forget(a)
}

fn f() {
    let a = Vec::<u8>::with_capacity(2000);
    std::mem::forget(a);
    g()
}

fn main() {

    let mut a = Vec::with_capacity(10);
    for _i in 0..10 {
        a.push(Box::new([0;1000]))
    }
    f();
    g();
}
```

Send in Sif

After we compile, run the command:

```
plam@amqui ~/c/p/l/l/L/alloc> valgrind --tool=massif target/debug/alloc
==406569== Massif, a heap profiler
==406569== Copyright (C) 2003-2017, and GNU GPL'd, by Nicholas Nethercote
==406569== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==406569== Command: target/debug/alloc
==406569==
==406569==
```

That Was Useful!!!

What happened?

1. The program ran slowly (because Valgrind!)
2. No summary data on the console
   (like memcheck or helgrind or cachegrind.)

Weird. What we got instead was the file `massif.out.[PID]`.

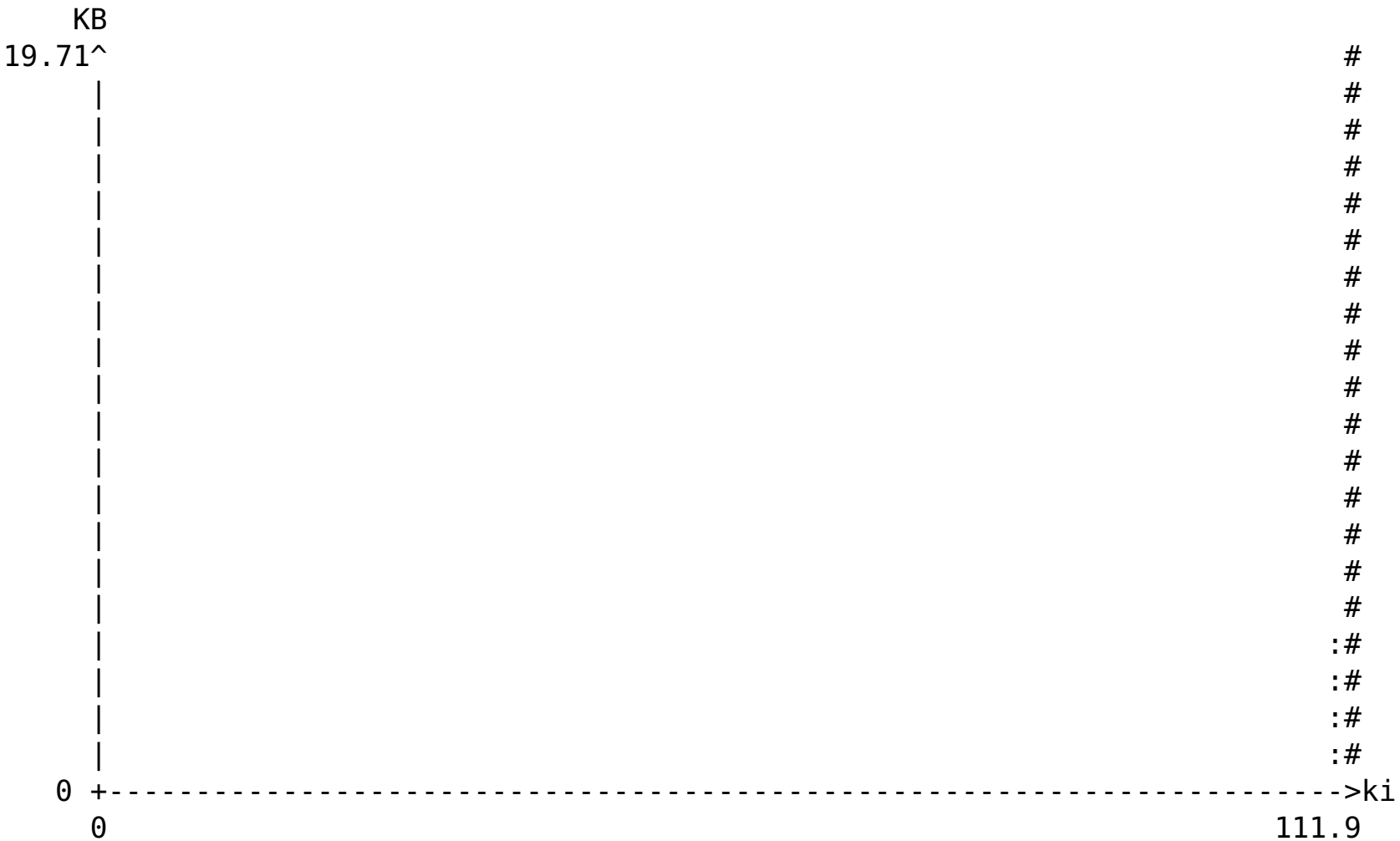
`massif.out.[PID]`:

plain text, sort of readable.

Better: `ms_print`.

Which has nothing whatsoever to do with Microsoft.
Promise.

Post-Processed Output



User Friendly, But Not Useful

For a long time, nothing happens, then…kaboom!

Why? We gave it a trivial program.

We should tell Massif to care more
about bytes than CPU cycles,
with `--time-unit=B`.

Let's try that.

Valgrind (Memcheck) First

Run valgrind (Memcheck) first and make it happy
before we go into figuring out where heap blocks are going with Massif.

Okay, what to do with the information from Massif, anyway?

Easy!

- Start with peak (worst case scenario)
  and see where that takes you (if anywhere).
- You can probably identify some cases where memory is hanging around unnecessarily.

Places to Look with Massif

Memory usage climbing over a long period of time, perhaps slowly, but never decreasing—memory filling with junk?

Large spikes in the graph—why so much allocation and deallocation in a short period?

Other Massif-ly Useful Things

- stack allocation (`--stacks=yes`).
- children of a process
  (anything split off with `fork`) if desired.
- low level stuff: if going beyond `malloc`, `calloc`, new, etc. and using `mmap` or `brk` that is usually missed, can do profiling at page level (`--pages-as-heap=yes`).

Live Demos

As is often the case,
we have examined the tool on a trivial program.

Let’s see if we can do some
live demos of Massif at work.