Lecture 28 — Causal Profiling, Memory Profiling

Patrick Lam & Jeff Zarnett

`patrick.lam@uwaterloo.ca` `jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

December 9, 2022

If we have more than one slow thing, which do we want to optimize?

Scientific approach: do an experiment, see the impact, re-evaluate.

Causal profiling lets us do what-if without code changes!

Example we'll discuss today: Coz.

Simple approach: look at the time and calculate what happens if `work()` runtime is reduced by 10%.

This model is okay but it's not very realistic.

We actually want a simulation!

Key observation: speeding up one part is the same as slowing all others down.

How? Insert pauses.

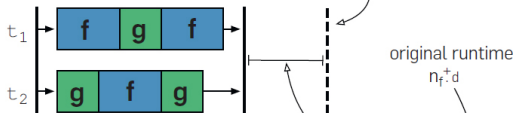Is that really equivalent to speeding up the target?

Illustration of virtual speedup

Speedup Results
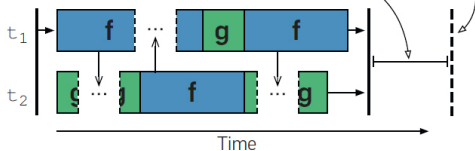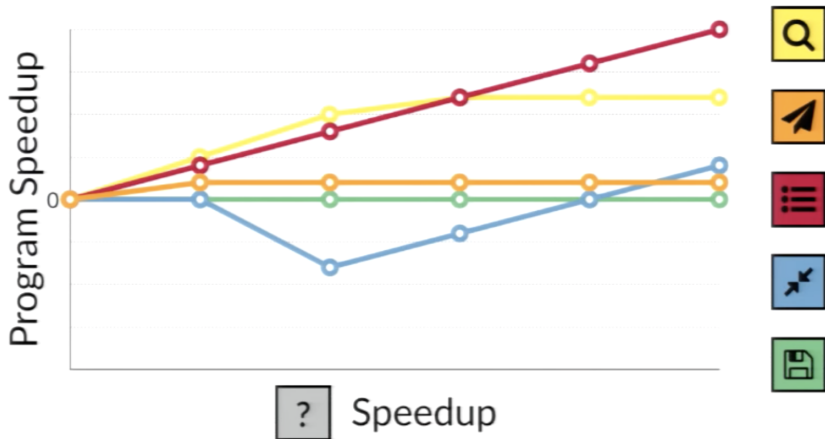
Just because hypothetically speeding up a particular part of the program would be beneficial, doesn't mean that it's possible to speed up that part.

We still need to assess what we can do and how difficult it would be.

Application	Summary of optimization results		
	Speedup	Diff size	Source lines
Memcached	9.39% ± 0.95%	−6, +2	10,475
SQLite	25.60% ± 1.00%	−7, +7	92,635
blackscholes	2.56% ± 0.41%	−61, +4	342
dedup	8.95% ± 0.27%	−3, +3	2,570
ferret	21.27% ± 0.17%	−4, +4	5,937
fluidanimate	37.5% ± 0.56%	−1, +0	1,015
streamcluster	68.4% ± 1.12%	−1, +0	1,779
swaptions	15.8% ± 1.10%	−10, +16	970

Only works on one machine where we have access to all threads.

Estimated overhead is 17.6% total:
    2.6% startup debug information collection
    4.8% sampling
    10.2% inserted pauses

More details in the author presentation:

```
https://www.youtube.com/watch?v=jE0V-p1odPg
```

As with queueing theory:

allocs $>$ frees $\implies$ 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 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.

```
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)
  }
}
```

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

The ˆ 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.

What does Massif do?

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

Next up: example from Massif docs.

```rust
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();
}
```

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==
```

What happened?

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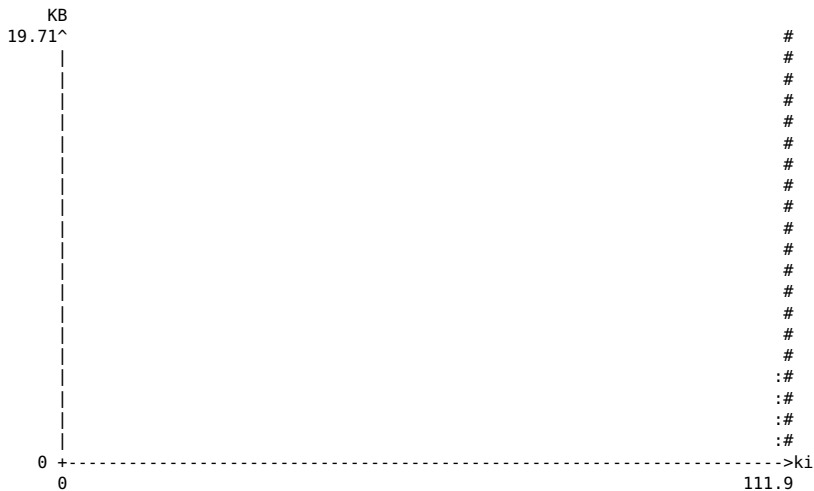   (like memcheck or helgrind or cachegrind.)

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.

```
      KB
 19.71^                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                      #
      |                                                     :#
      |                                                     :#
      |                                                     :#
      |                                                     :#
    0 +------------------------------------------------>ki
      0                                              111.9
```

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.

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.

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?

- 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`).

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.