Lecture 28 — Causal and Simulation Profiling

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

Department of Electrical and Computer Engineering
University of Waterloo

July 8, 2025

Part I

Causal Profiling

Causal Profiling

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

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

Example we'll discuss today: Coz.

Why Causal Profiling?

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!

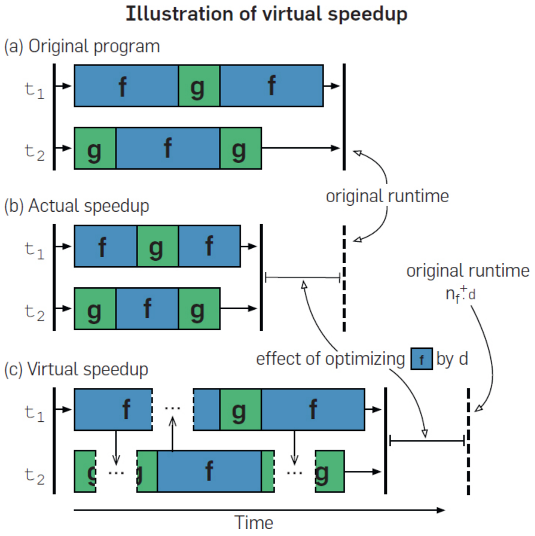
Key Idea: Make Others Worse

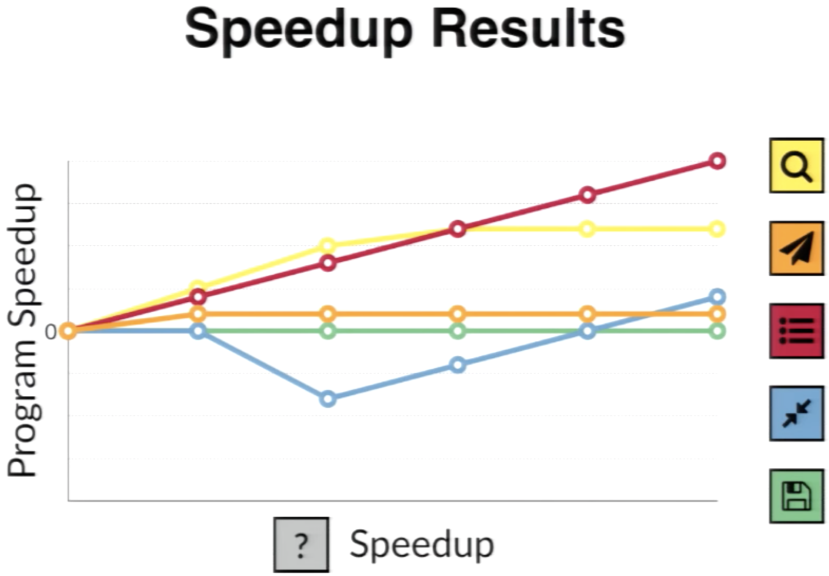
Key observation: speeding up one part is the same as slowing all others down.

How? Insert pauses.

Is that really equivalent to speeding up the target?

Slowdown is Speedup



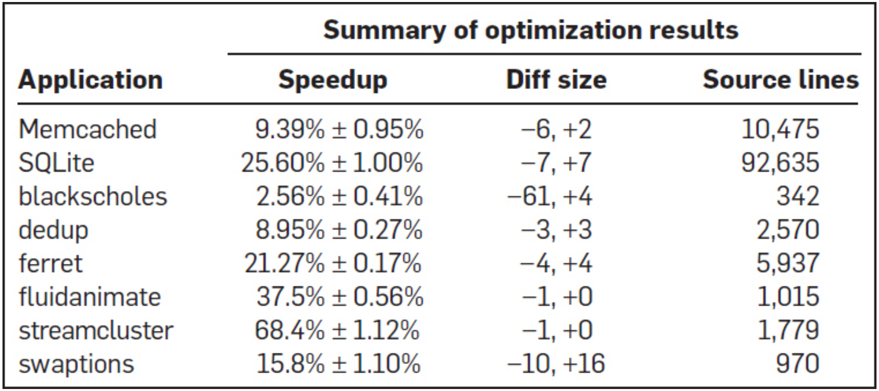

Speedup Results
Program Speedup
0
?
Speedup

That Would Be Nice

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.

Does it Work?



	Summary of optimization results		
Application	**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

Limitations, Overhead

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

More details in the author presentation:

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

Next Generation: `scoz`

The next extension `scoz` is meant to address some shortcomings:

1. Inability to look at multi-process applications

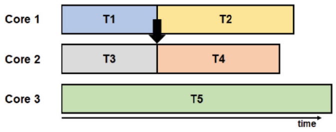
2. Cases where the OS is the bottleneck.

(Reducing systems calls is a form of “do less work”)

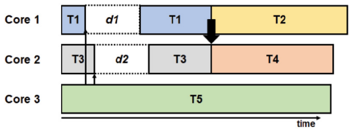
The Goal: Core-Based System

Instead of pausing threads, pause other cores.

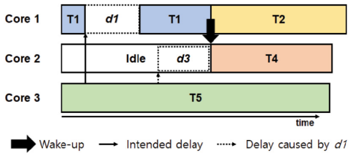
(There are details about how to charge delays to the “right” core)



(A) Original program



(B) Core-level virtual speedup



(C) Core 2 is in idle state before thread 4 wakes up

Implementation

The implementation involves one profiler thread for each core to manage its behaviour.

The thread is pinned to the core, and its purpose is primarily to call the `ndelay` interface (with preemption temporarily disabled) to stop execution.

Part II

Simulation Profiling

Simulations

Causal profiling lets us use a simulation to decide what to do.

Let’s apply this idea to some other things!

Cachegrind

This is much more performance oriented than the other two tools.

It runs simulation of how your program interacts with cache and evaluates how your program does on branch prediction.

As we discussed earlier, cache misses and branch mispredicts have a huge impact on performance.

Recall that a miss from the fastest cache results in a small penalty (10 cycles).

A miss that requires going to memory requires about 200 cycles.

A mispredicted branch costs somewhere between 10-30 cycles.

Cachegrind Reporting

Cachegrind reports data about:

- The First Level Instruction Cache (I1) [L1 Instruction Cache]
- The First Level Data Cache (D1) [L1 Data Cache]
- The Last Level Cache (LL) [L3 Cache].

Unlike normal Valgrind operation, you probably want to turn optimizations on.

Sim-u-laaaaate!

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search

--16559-- warning: L3 cache found, using its data for the LL simulation.
Found at 11 by thread 1
Found at 22 by thread 3
==16559==
==16559== I   refs:      310,670
==16559== I1  misses:      1,700
==16559== LLi misses:      1,292
==16559== I1  miss rate:    0.54%
==16559== LLi miss rate:    0.41%
==16559==
==16559== D   refs:      114,078  (77,789 rd   + 36,289 wr)
==16559== D1  misses:      4,398  ( 3,360 rd   +  1,038 wr)
==16559== LLd misses:      3,252  ( 2,337 rd   +    915 wr)
==16559== D1  miss rate:     3.8% (   4.3%     +    2.8%  )
==16559== LLd miss rate:     2.8% (   3.0%     +    2.5%  )
==16559==
==16559== LL refs:         6,098  ( 5,060 rd   +  1,038 wr)
==16559== LL misses:       4,544  ( 3,629 rd   +    915 wr)
==16559== LL miss rate:      1.0% (   0.9%     +    2.5%  )
==16559==
==16559== Branches:       66,622  (65,097 cond +  1,525 ind)
==16559== Mispredicts:     7,202  ( 6,699 cond +    503 ind)
==16559== Mispred rate:     10.8% (  10.2%     +   32.9%   )
```

Optimizations: Enabled!

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search

--16618-- warning: L3 cache found, using its data for the LL simulation.
Found at 11 by thread 1
Found at 22 by thread 3
==16618==
==16618== I   refs:      306,169
==16618== I1  misses:      1,652
==16618== LLi misses:      1,286
==16618== I1  miss rate:    0.53%
==16618== LLi miss rate:    0.42%
==16618==
==16618== D   refs:      112,015  (76,522 rd   + 35,493 wr)
==16618== D1  misses:      4,328  ( 3,353 rd   +    975 wr)
==16618== LLd misses:      3,201  ( 2,337 rd   +    864 wr)
==16618== D1  miss rate:     3.8% (   4.3%     +    2.7%  )
==16618== LLd miss rate:     2.8% (   3.0%     +    2.4%  )
==16618==
==16618== LL refs:         5,980  ( 5,005 rd   +    975 wr)
==16618== LL misses:       4,487  ( 3,623 rd   +    864 wr)
==16618== LL miss rate:      1.0% (   0.9%     +    2.4%  )
==16618==
==16618== Branches:       65,827  (64,352 cond +  1,475 ind)
==16618== Mispredicts:     7,109  ( 6,596 cond +    513 ind)
==16618== Mispred rate:     10.7% (  10.2%     +   34.7%   )
```

Results Analysis

Interesting results: our data and instruction miss rates went down marginally but the branch mispredict rates went up!

Well sort of - there were fewer branches and thus fewer we got wrong as well as fewer we got right.

So the total cycles lost to mispredicts went down.

Is this an overall win for the code? Yes.

Do the Math

In some cases it’s not so clear cut, and we could do a small calculation.

If we just take a look at the LL misses (4 544 vs 4 487) and assume they take 200 cycles, and the branch miss penalty is 200 cycles, it went from 908 800 wasted cycles to 897 400; a decrease of 11 400 cycles.

Repeat for each of the measures and sum them up to determine if things got better overall and by how much.

Cachegrind Detailed Output

Cachegrind also produces a more detailed output file, titled cachegrind.out.<pid> (the PID in the example is 16618).

This file is not especially human-readable, but we can ask the associated tool `cg_annotate` to break it down for us.

The output is way too big for slides.

Advanced Cachegrind

Cachegrind is very... verbose... and it can be very hard to come up with useful changes based on what you see...

Assuming your eyes don't glaze over when you see the numbers.

Probably the biggest performance impact is last level cache misses (those appear as DLmr or DLmw).

You might also try to look at the Bcm and Bim (branch mispredictions) to see if you can give some better hints.

Really Advanced Cachegrind

This is a good way to validate that your changes actually worked!

Showing your manager (executive) WHY a change worked is more convincing...

Simulations as Perf Debugging

At least once a day, I walk outside and say, 'Computer, end program,' just in case.

Executing in a simulation allows for more than just counting the number of cache misses or a pipeline stalls – you can see how we got there.

Simulation Example

Docs for TMS320C55x Digital Signal Processing platform by Texas Instruments.

Get more exact values of counts, cache misses, CPU stalls...
- And you can halt on events to look into them!

Miri

Everyone was extremely cool about the duplicate Earth...

Miri & Valgrind

Miri has a lot of similarities with Valgrind!

I think its real intended use is to detect issues in `unsafe` code.

- Array bounds violations, reading uninitialized memory, memory leaks...

... Well. Except for launching GPU code.

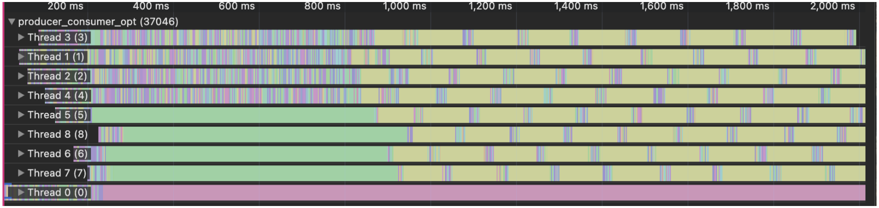
Let’s Do an Example

To run, use the miri option and it will generate a `crox` directory with some file in it; process that file with the `crox` tool to generate a json file that can be read in chrome dev tools:

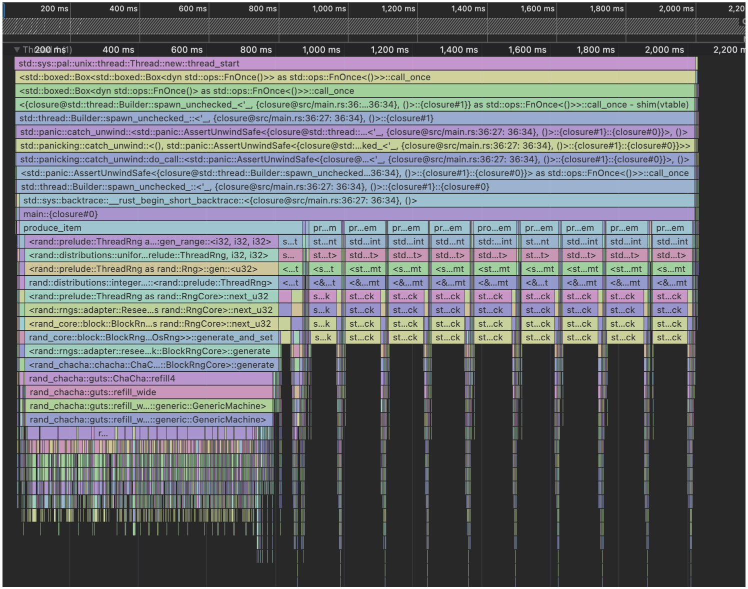
```
MIRIFLAGS="-Zmiri-disable-isolation -Zmiri-measureme=crox" cargo +nightly miri
        run
```

It produces a flamegraph-style graph!

Miri Example



Miri Example


200 ms 400 ms 600 ms 800 ms 1,000 ms 1,200 ms 1,400 ms 1,600 ms 1,800 ms 2,000 ms 2,20
std::sys::pal::unix::thread::Thread::new::thread_start
<std::boxed::Box<std::boxed::Box<dyn std::ops::FnOnce()>> as std::ops::FnOnce<()>>::call_once
<std::boxed::Box<dyn std::ops::FnOnce()> as std::ops::FnOnce<()>>::call_once
<{closure@std::thread::Builder::spawn_unchecked_<'_, {closure@src/main.rs:36:...36:34}, ()>::{closure#1}} as std::ops::FnOnce<()>>::call_once - shim(vtable)
std::thread::Builder::spawn_unchecked_::<'_, {closure@src/main.rs:36:27: 36:34}, ()>::{closure#1}
std::panic::catch_unwind::<std::panic::AssertUnwindSafe<{closure@std::thread::...<'_, {closure@src/main.rs:36:27: 36:34}, ()>::{closure#1}::{closure#0}}>, ()>
std::panicking::catch_unwind::<(), std::panic::AssertUnwindSafe<{closure@std::...ked_<'_, {closure@src/main.rs:36:27: 36:34}, ()>::{closure#1}::{closure#0}}>>
std::panicking::catch_unwind::do_call::<std::panic::AssertUnwindSafe<{closure@...<'_, {closure@src/main.rs:36:27: 36:34}, ()>::{closure#1}::{closure#0}}>, ()>
<std::panic::AssertUnwindSafe<{closure@std::thread::Builder::spawn_unchecked...36:34}, ()>::{closure#1}::{closure#0}}> as std::ops::FnOnce<()>>::call_once
std::thread::Builder::spawn_unchecked_::<'_, {closure@src/main.rs:36:27: 36:34}, ()>::{closure#1}::{closure#0}
std::sys::backtrace::__rust_begin_short_backtrace::<{closure@src/main.rs:36:27: 36:34}, ()>
main::{closure#0}
produce_item
<rand::prelude::ThreadRng a...:gen_range::<i32, i32, i32>
<rand::distributions::unifor...relude::ThreadRng, i32, i32>
<rand::prelude::ThreadRng as rand::Rng>::gen::<u32>
rand::distributions::integer...::<rand::prelude::ThreadRng>
<rand::prelude::ThreadRng as rand::RngCore>::next_u32
<rand::rngs::adapter::Resee...s rand::RngCore>::next_u32
<rand_core::block::BlockRn...s rand::RngCore>::next_u32
rand_core::block::BlockRng...OsRng>>::generate_and_set
<rand::rngs::adapter::resee...k::BlockRngCore>::generate
<rand_chacha::chacha::ChaC...::BlockRngCore>::generate
rand_chacha::guts::ChaCha::refill4
rand_chacha::guts::refill_wide
rand_chacha::guts::refill_w...::generic::GenericMachine>
rand_chacha::guts::refill_w...::generic::GenericMachine>
pr...m pr...em pro...em
st...nt std...int std:...int
st...t> std...t>
<s...t <st...mt <s...mt
<&...t <&...mt
s...k st...ck

Miri Drawbacks

Like Valgrind, it makes the program run much, much slower.

The article got a lot of comments about improving the debug version rather than the release one.

That is true, but it’s meant to be used in the change-run-evaluate loop!