

Lecture 28 — Causal Profiling

Jeff Zarnett

2025-06-22

Causal Profiling

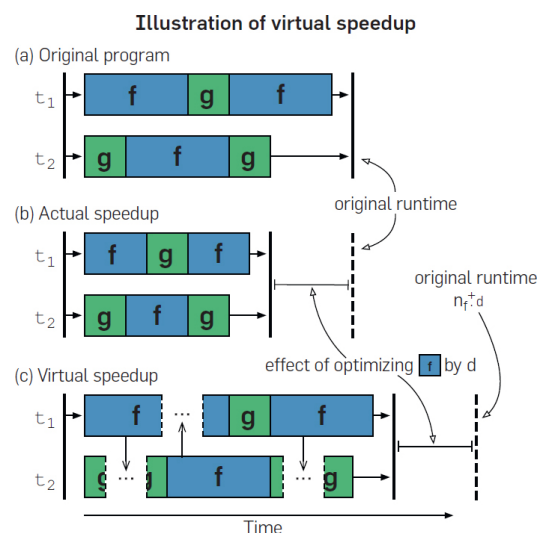
At this point we've got some experience in identifying areas of the program that we think are slow or are limiting the maximum performance of our program. If we are presented with more than one thing, how do we know which of those would yield the most benefit? Is it possible that optimizing something would actually have no effect or even a negative effect? The scientific approach would be to do an experiment and find out. That is, change the code, see the impact, re-evaluate. What causal profiling offers us is a way to run those experiments without changing any code. That could be a significant savings of time and effort.

One such causal profiler is called Coz (pronounced like “cause”) [CB15]. It does a what-if analysis that says: what would be the impact of speeding up this particular part of the code?

A very simple approach would just look at the time of the program and just calculate what happens if the runtime of function `work()` is reduced by, say, 10%. But that isn't a realistic approach, because speeding up that function might have no overall impact or change the execution time by increasing lock contention or some other mechanism. No, we actually need a simulation.

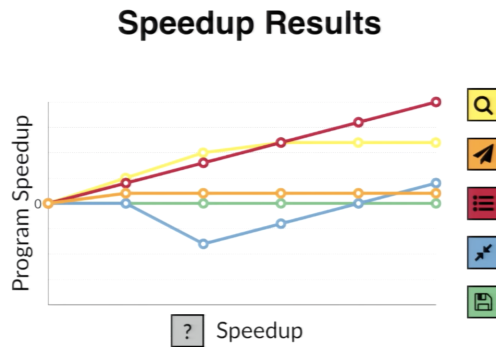
The key observation of the Coz profiler authors is the idea that speeding up some area of code is fundamentally the same as slowing down every other part of the code. It's all relative! This is what we would call a virtual speedup. How is the other code slowed down? Adding some pauses to (stopping the execution of) the other threads. That certainly makes them run slower. Maybe you're not convinced that slowing down everything else is equivalent?

Shown below is the argument from [CB15] in visual form. The original runtime is shown as (a). Hypothetically, say we make function `f` faster by some factor $d = 40\%$ that gives us (b). But, if instead of actually making function `f` faster, let's pause other threads for $0.4 \times t(f)$ where $t(f)$ is the original execution time of `f`. Then we get an equivalent effect to that of optimizing `f` by d .



The tool provides some graphs of potential program speedup. From the recorded presentation, it might look something like the diagram below, where the boxes on the right correspond to different theoretical actions in the

application being examined:



This pretend application shows all the possible outcomes, from continuous linear speedup, to speedup capped at some point, to no effect, and finally to where optimizing something makes the overall program runtime worse.

It's easy to imagine scenarios corresponding to each of those. If we're computing the n-body problem, anything that improves the calculation of forces will certainly make things better. And it's easy to imagine that sometimes optimizing a part of the program does not improve anything because that code is not on the critical path. We can also easily imagine that improving something works up to a point where it ceases to be the limiting factor. But making things worse? We already covered that idea: speeding up a thread may increase lock contention or add something to the critical path. At some point, things may recover and be a net benefit.

It is important to remember that 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. And almost certainly not possible to do so to an arbitrary degree. We still have to make an assessment of what optimizations we can make and how difficult it would be to actually realize those improvements.

Once we've made a change to the program, then it's time to run an experiment again with the new baseline to see what else we can do.

The paper has a table summarizing the optimizations they applied to a few different programs, which seems to support the idea that the tool can be used effectively to get a meaningful speedup with relatively few lines of code [CB15]:

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

There are potentially some limitations to this, of course. Putting pauses in the execution of the code can work when the execution of the program is all on the same machine and we have control over all the threads; it would need some meaningful extension to work for a distributed system where coordinating things across many servers would be needed.

Using some benchmarking workload, the authors estimate a 17.6% overhead for this tool, which is broken down into 2.6% for startup debug information collection, sampling at 4.8%, and 10.2% is the delay that's caused by slowing down other threads to create a virtual speedup [CB15].

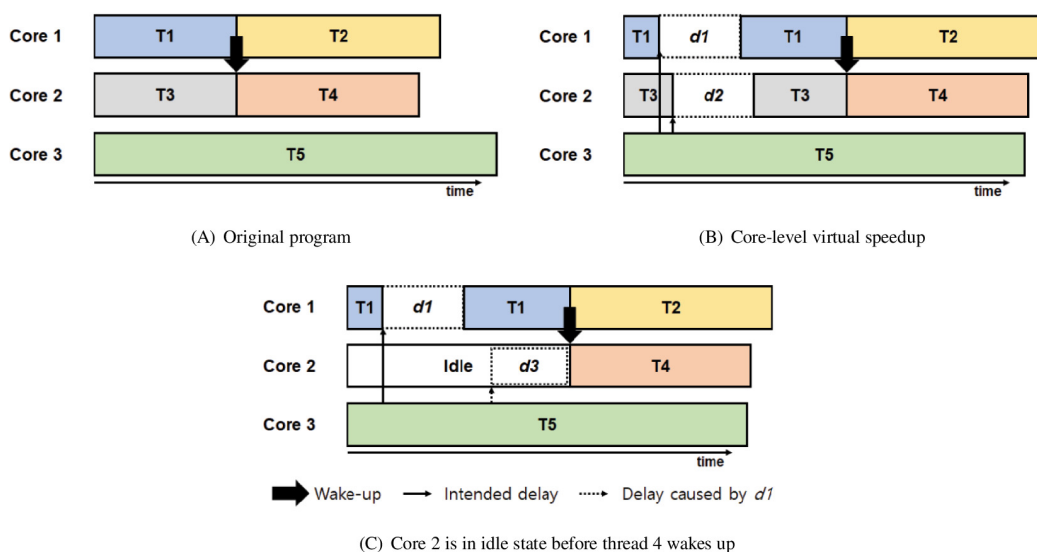
If you'd like to see some more details of the tool, you can see the author presentation at the ACM symposium here: <https://www.youtube.com/watch?v=jE0V-p1odPg>

The Next Generation: SCOZ

A few years later, another group of researchers worked to extend the Coz profiler to account for some limitations: inability to look at multi-process applications (which we just talked about), but also cases where the OS itself becomes a bottleneck [AKNJ21]. You may recall that looking into the kernel is not always permitted for a given profiling tool – depending on permissions. It is therefore obvious to imagine that if one cannot look, then one cannot touch.

If, indeed, the kernel is a bottleneck, it may or may not be possible to do very much about it. Avoiding system calls is one of the “do less work” strategies that you have likely already considered; oftentimes doing the system call is because we must. In an open-source OS or a kernel module we control, then we would have the ability to do something about it, if coz could give some recommendations.

The approach of [AKNJ21] is to move from a thread-based system to a core-based one. So where, previously, to simulate speedup of a particular section of code the tool would pause all other threads, now it pauses execution on all other cores. There are some details to consider about what to do when there are idle cores to make sure that the delay is charged to the “right” core when one is woken up by another. The diagram the authors provides helps to explain it well:



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 of code on that specific core [AKNJ21].

This extension does allow scoz to do more than coz, but it does require running within the kernel, which imposes certain limitations to where and how it can be applied. That's an interesting tradeoff!

References

- [AKNJ21] Minwoo Ahn, Donghyun Kim, Taekeun Nam, and Jinkyu Jeong. Scoz: A system-wide causal profiler for multicore systems. *Software: Practice and Experience*, 51(5):1043–1058, 2021. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2930>, arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2930>, doi:10.1002/spe.2930.
- [CB15] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815409.