Overall merit 4. Accept
Reviewer expertise 2. Some familiarity

## Paper summary

When profiling a multi-threaded program trying to reduce its execution time, what matters isn't necessarily the code line that takes the longest to execute, or that is executed the most, but the code line that *if it were to run faster* it would reduce the execution time the most. Causal profiling tries to estimate if a code line, when manually optimized, would reduce program execution time the most (actually, improve latency and throughput) by "virtually" speeding it up: it does that by slowing every other thread down while that line is executing. By measuring how "virtual speed up" of a line affects latency and throughput, the tool presented, Coz, suggests lines that would improve things the most if optimized. The authors demonstrate the effectiveness of Coz with a large number of case studies (including Memcached and SQLite), in which very targeted optimizations on code suggested by Coz yielded speedup of up to 68%, with relatively little runtime overhead compared to other profilers.

## Comments for author

The idea behind Coz is simple, which makes it all the more impressive, given how well it seems to work. Details in implementing the idea are non-trivial, and you explained them well. I enjoyed reading the paper.

The results are impressive, and it was great reading details about what exactly was identified by Coz in each case study, and how you fixed it. Without understanding each code base better, it's tough to put your optimizations in context (and, frankly, the actual optimizations don't matter all that much, as contrasted to the performance improvement they create). For example, removing all those function pointers in SQLite sounds awesome, given that the functions are so fast already, but there must have been some purpose to having function pointers there. I don't know the internals of SQLite so I can't say.

So although very enjoyable, the evaluation left me wondering if I should be convinced that Coz is a good tool for finding optimization opportunities, or that you just did a good job optimizing. You did a good job convincing me that gprof

wasn't all that useful in pointing out where to invest your optimization skills, though, so that definitely counts.

It would be valuable to point out that there are many objectives for an optimization effort, not just time to completion. For instance, optimizing a function that executes for a long time, even if it ends at a barrier that other functions stall at, improves power consumption, since a speedy core finishing up its work can do something else for some other concurrent workload, or can just go into low power. So profiling programs to see what operations take a long time isn't as useless as your paper seems to imply; it's just useful towards optimizing for a different objective goal. Putting your contribution into context doesn't diminish its value.

The difference in approach between high estimated speedup and contention is interesting. In many of the case studies, Coz identified contention rather than potential speedup, but contention is not something that you got into very deeply in the early modeling sections. As such, I felt a little that you sold me one thing (virtual speedup -> potential real speedup) but then you got the good results with another (virtual slowdown -> potential contention -> speedup by removing contention). It would be good if in the next version you did some modeling of contention and the intuition behind your approach in Section 2.

---

Figure 3(c) doesn't make sense. Why is the pause applied to the end of the first invocation of f? Was the gray square supposed to be attached to g instead?

Section 2.0 seems almost entirely redundant, given the first 2 pages. The only new content I saw was the discussion on contention in the last paragraph before 2.1. I'd say eliminate the text duplication.

The explanation about systematic bias made a lot of sense after I read section 3.2, but the reason, of course, is that there's a single profiling run during which lines and speed ups are tried in succession. If there were a profiling run per combination of line and speedup, this would not be an issue. It would help if you brought up this detail earlier (that there's a single, on-going profiled execution for all experiments) to avoid confusion.

In section 4.1, you say you run with batch size of 10. What's a batch? You never mention batches in the paper elsewhere.

The note in Figure 9 on performance improvements is insufficient to explain the figure. Somehow sampling overhead is negative (an improvement) but the rest is positive? How did you measure a negative overhead in the context of positive overhead?

## Summary

This paper presents gprof, a low-overhead profiler which can accurately attributes execution times to functions--both their own execution times and the execution times of the functions they invoke (e.g. actual_time(A) = time(A) - time(B) if A calls B). It exploits the fact that modern programs are structured and hierarchical, thereby constructing call graphs to differentiate between routine executions.

The profiling data is collected by a sampling function call inserted into the beginning of each program function. Function execution counts and call graph edge information are collected exactly by the sampling function. Function execution times can't be collected accurately on time-sharing systems, so they are inferred from the distribution of program counter values, which are collected at uniform intervals. The data is not processed or printed by the sampling function, so there's minimal interference with program execution.

After program termination, the collected data is used to build a call graph using topological sorting. We must handle recursive call cycles specially and treat them as a single node to avoid disaster. Since the dynamic call graph can change between runs, incorporating the static call graph might be desirable. Function execution times can then be propagated from leaves to roots (again, cycles are tricky).

## Context

gprof is still widely used, but it certainly has many drawbacks. On Linux we now have perf available, which can perform more comprehensive sampling than gprof. And then there's COZ from the other paper using causal profiling, which can be much more helpful than execution time only profiling.

## Discussion Points

1. "The purpose of the gprof profiling tool is to help the user evaluate alternative implementations of abstractions." gprof doesn't promise to speed up your *current* implementation.
2. "The monitoring routine also records the arc in the call graph that activated the profiled routine. The count is associated with the arc in the call graph rather than with the routine." If there are many edges, gprof can be space-inefficient. This probably doesn't make a difference on modern machines, but it should matter 30 years ago.
3. "We are fortunate to be running in a virtual memory environment ..." What's the implications of having a virtual address space? Smaller chance of hash collisions?
4. Is the program counter value collected by a separate thread? It seems strenuous to handle this in those sampling functions, as one of them must be waken up at uniform intervals.

5. "The drawback to inline expansion is that the data abstractions in the program may become less parameterized, hence less clearly defined. The profiling will also become less useful ..." I think this is a significant drawback. Modern compilers can perform aggressive optimizations, yet gprof doesn't seem to handle optimized builds very well. On the other hand, profiling and optimizing unoptimized builds may or may not have an impact on the performance of optimized builds.
6. gprof supposedly succeeded prof. I wasn't able to find much information on prof. What was prof like?