COZ
This *Is* the Profiler You're Looking For

CHARLIE CURTSINGER AND EMERY D. BERGER

Charlie Curtsinger is a new faculty member in the Computer Science Department at Grinnell College. His research interests include software performance, security, and reliability with an emphasis on probabilistic and statistical techniques. curtsinger@grinnell.edu

Emery Berger is a Professor in the College of Information and Computer Sciences at the University of Massachusetts Amherst, where he co-directs the PLASMA lab (Programming Languages and Systems at Massachusetts) and is a regular visiting researcher at Microsoft Research. He is the creator of a number of influential software systems including Hoard, DieHard, and DieHarder, a secure memory manager that was an inspiration for hardening changes made to the Windows 8 heap. He is currently serving as Program Chair for PLDI 2016. emery@cs.umass.edu

Causal profiling is a new approach to software profiling that tells developers which code is important for performance. Conventional profilers report where programs spend their time running, but optimizing long-running code may not improve program performance. Instead of simply observing a program, a causal profiler conducts *performance experiments* to predict the effect of speeding up many different parts of a program. During each experiment, a causal profiler uses *virtual speedup* to create the effect of optimizing part of the program, and *progress points* to measure any change in program performance as a result of the virtual speedup. A causal profile summarizes the results of many performance experiments, telling developers exactly where performance tuning would be worthwhile. Using COZ, a prototype causal profiler for Linux, we improve the performance of Memcached by 9%, SQLite by 25%, and several PARSEC applications by as much as 68%.

"Try running it with a profiler." This suggestion inevitably comes up once all reasonable ideas for improving a program's performance are exhausted. The authors of thousands of lines of code have been unable to come up with any explanation for the system's poor performance, so why do we expect a tool from 1982 to fare better [3]? Deep down, we've always known this was true. Take the historically accurate space adventure game you were playing too late last Tuesday as an example; how would a profiler know that the thread that plays lightsaber crackling sounds was less important than the thread that controls the stormtrooper you were battling? *It wouldn't.* When we look at a software profile we aren't looking for guidance, we're looking for surprises. And with parallel programs, *practically everything* is surprising. That's not to say that profilers aren't informative. Any good software profiler can tell you, with great accuracy, where a program spends its execution time. Unfortunately, this isn't the information we're looking for.

Code that runs for a long time is not necessarily a good choice for performance tuning. Developers need to know which code is important—where successful performance tuning would improve the program's end-to-end performance. Consider a function that draws a "loading ..." animation while you wait for the next level of your game to load. The animation runs just as long as the loading code, but we would never expect to speed up the program by making the animation faster.

This problem is not limited to programs that perform I/O. Figure 1 shows a simple parallel program with a similar issue. This program creates two threads, one to run the function `a()` and another to run the function `b()`. The program exits once both threads have finished. A conventional profiler like gprof, whose output for this program is shown in Figure 2, reports that the program spends roughly equal time running `a()` and `b()`. While accurate, this information is misleading; optimizing `a()` alone will speed the program up by just 4.5%, and optimizing `b()` will have no effect on performance.

example.cpp

```
void a() { // ~6.7 seconds
  for(volatile size_t x=0; x<2000000000; x++) {}
}
void b() { // ~6.4 seconds
  for(volatile size_t y=0; y<1900000000; y++) {}
}
int main() {
  // Spawn both threads and wait for them.
  thread a_thread(a), b_thread(b);
  a_thread.join(); b_thread.join();
}
```

Figure 1: A simple multithreaded program that illustrates the shortcomings of existing profilers. Optimizing `a()` will improve performance by no more than 4.5%, while optimizing `b()` would have no effect on performance.

The key issue with conventional profilers is that they only *observe* a program's execution. Through observation alone, they cannot tell you which code to optimize, because long-running code is not necessarily important to program performance. Speeding up a line of code might shorten an important path through the program, or it may speed up a thread that does background work, increasing contention on a critical data structure, which in turn hurts overall program performance.

```
% time   cumulative      self              self     total
           seconds    seconds    calls   Ts/call  Ts/call   name
 55.20        7.20       7.20        1                      a()
 45.19       13.09       5.89        1                      b()

% time      self     children      called      name
                                             <spontaneous>
  55.0      7.20         0.00                 a()
-------------------------------------------------------------------
                                             <spontaneous>
  45.0      5.89         0.00                 b()
```

Figure 2: A conventional profile for `example.cpp` collected with gprof

Causal Profiling

Causal profiling is a novel approach to profiling that identifies code where optimizations will have the largest impact [2]. A causal profiler is fundamentally different from a conventional profiler; rather than simply observing program execution, a causal profiler intentionally perturbs program performance to conduct *performance experiments.* During a performance experiment, a causal profiler creates the effect of speeding up some piece of a program using *virtual speedup* (more on this later). While virtually speeding up one piece of a program, a causal profiler then measures program performance to determine the effect of this speedup. Given enough performance experiments

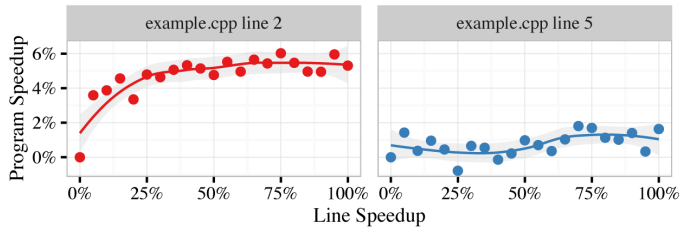

Figure 3: Causal profile for `example.cpp`

with varying locations and sizes of virtual speedup, we can construct a causal profile, which tells you both where optimizations would have an effect and how large that effect would be.

Figure 3 shows a real causal profile for the program in Figure 1 collected with COZ, a prototype causal profiler for Linux. This causal profile suggests that speeding up `a()` alone could improve program performance by up to 5.0%, very close to the actual 4.5%. Beyond this point, the thread running `b()` becomes the program's critical path. The causal profile correctly indicates that speeding up `b()` alone would have a negligible effect on performance.

Producing a causal profile requires three key pieces: we need a way to create the effect of an optimization, the profiler must apply a virtual speedup for the duration of a performance experiment, and we need a way to measure a program's performance at the end of each experiment.

Virtual Speedup

A causal profiler cannot magically speed up a part of a program and measure the effect of that speedup; if this were possible, we would just magically speed up the entire program. Instead, a causal profiler creates the effect of speeding up one part of a program by slowing everything else down. The amount that other threads are slowed down determines the size of the virtual speedup. The size of the virtual speedup can range from 0% (the code's runtime is unchanged) to 100% (the code's runtime is reduced to zero).

Figure 4 illustrates a virtual speedup in a simple parallel program. Part (a) shows the original execution of two threads running functions `f()` and `g()`, and part (b) shows the effect of *actually* speeding up `f()` by 40%; the size of this speedup was chosen arbitrarily and could be any value from 0% to 100%. Finally, part (c) shows the effect of *virtually* speeding up `f()` by 40%.

Each time `f()` runs in one thread, all other threads pause for 40% of `f`'s original execution time. While virtual speedup does not actually shorten the program's runtime, the difference between the program's original runtime and its runtime with a virtual speedup is known: it is just the number of times `f()` ran multiplied by the delay size. Given that we know both quantities, we

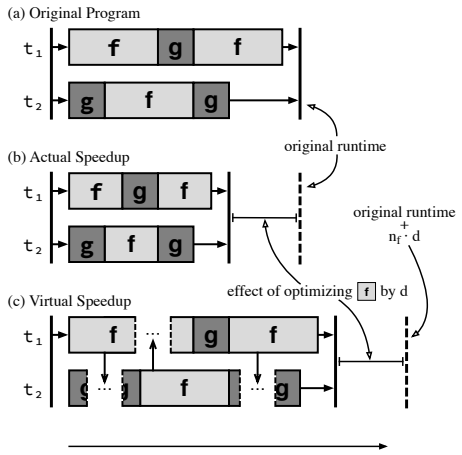

Figure 4: An illustration of a program's (a) original execution, (b) a real speedup of function `f()` by 40%, and (c) a virtual speedup of `f()` by 40%

can adjust the baseline runtime by this extra time to predict the effect of an actual speedup.

Instrumenting a program to track visits to `f()` and signaling other threads to pause them every time it runs would be prohibitively expensive. Instead, COZ uses sampling to approximate this approach. Instead of pausing other threads each time `f()` runs, COZ would delay other threads every time it sees a sample in `f()`. The size of the delay is proportional to the sampling interval rather than the execution time of a single call to `f()`.

Performance Experiments

A causal profiler can use virtual speedup to test the effect of a potential optimization, but how does it decide which code to virtually speed up, and by how much? Coverage is particularly important: given a large code base, we would like to find the one line of code with the largest possible payoff from optimizations. COZ applies virtual speedup at the granularity of source lines, which means there are potentially tens of thousands of program fragments that could be virtually sped up. Rather than choosing uniformly from all source lines, COZ selects from the distribution of where a program spends its time running. While long-running code may not be the best place for performance tuning, code that never runs is certainly a bad place to focus our limited energy. Once a COZ selects a line to virtually speed up, it selects a speedup size between 0% and 100% in increments of 5%.

During a performance experiment, COZ applies the same fixed virtual speedup to the selected line. All that is required to speed up a specific line is to map program execution samples, which are memory addresses of code, to source information. This is a relatively straightforward process using DWARF debugging information. While COZ currently uses source lines as the unit of virtual speedup, any fragment of code that can be mapped to addresses could be virtually sped up.

At the end of a performance experiment, COZ measures the program's performance with the virtual speedup in place. But how can we measure performance in the middle of an execution or for programs that run indefinitely? For the simple example in Figure 3, COZ runs a single performance experiment for the entire execution of the program. While this approach works for small programs, it does not scale well; large programs would require, at minimum, thousands of runs to get reasonable coverage with performance experiments. COZ solves this problem by allowing developers to specify progress points.

Progress Points

A progress point is some point in a program that should happen as frequently as possible, completing a user request, for example, or processing a block of data. Developers mark one or more progress points in their application by adding the `COZ_PROGRESS` macro, which keeps a count of the visits to this point in the code. COZ measures the rate of visits to a progress point as a proxy for performance. This allows COZ to conduct many performance experiments in a single run of a program or in programs where end-to-end runtime is not meaningful such as servers and interactive applications.

This basic notion of progress points allows us to measure throughput at some point in the code, but COZ can also use progress points to measure the latency between two points. Instead of specifying a single point, developers mark the beginning and end of a transaction using two macros, `COZ_BEGIN` and `COZ_END`. COZ does not track individual transactions as they flow through the system, but by measuring the rate of arrivals at the beginning point and the number of outstanding requests, COZ can use Little's Law to compute the average latency between the two points [4].

Using COZ

Running a program with COZ requires just three steps: (1) find one or more places to add progress points that allow COZ to measure the program's performance; (2) run the program with the command-line driver: `coz run --- <program> <args>`; and (3) use COZ's Web-based profile interface to plot the results and rank lines by potential impact. Our SOSP 2015 paper on causal profiling includes case studies where we use COZ to optimize eight different applications, three of which are included below [2]. These case studies include the compression program dedup, where COZ led us to a degenerate hash function; the embedded database SQLite, where COZ guided us to an inefficient coding practice that prevented function inlining; and the in-memory key-value store Memcached, where COZ identified unnecessary contention on a shared lock. Fixing these issues led to whole-program performance improvements of 9% for dedup, 25% for SQLite, and 9% for Memcached.

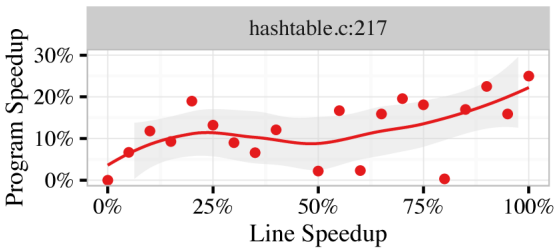

Figure 5: The causal profile for the source line `hashtable.c:217` in dedup shows the potential performance improvement of fixing a hash bucket traversal bottleneck.

Case Study: dedup

The dedup application, part of the PARSEC suite, performs parallel file compression via deduplication [1]. We added a progress point to dedup's code just after a single block of data is compressed (`encoder.c:189`).

COZ identifies the source line `hashtable.c:217` as an opportunity for optimization; Figure 5 shows the causal profile results for this line. This plot shows that improving the performance of the code that runs this line will result in a nearly one-to-one performance improvement in program performance up to 20%, with modest additional gains for performance improvements over 20%. This code is the top of the `while` loop in `hashtable search` that traverses the linked list of entries that have been assigned to the same hash bucket. This suggests that dedup's shared hash table has a significant number of collisions. Hash collisions could be caused by two things: a hash table that is too small or a hash function that does not evenly distribute elements throughout the hash table. Increasing the hash table size had no effect on performance, so the only remaining culprit is the hash function. It turns out dedup's hash function was mapping keys to just 2.3% of the available hash table buckets; over 97% of hash buckets were never used during the entire execution, and the 2.3% of buckets that were used at all contained an average of 76.7 entries.

The original hash function adds characters of the hash table key, which leads to virtually no high-order bits being set. The resulting hash output is then passed to a bit-shifting procedure intended to compensate for poor hash functions. Removing the bit-shifting step increased hash table utilization to 54.4%, and changing the hash function to use bitwise XOR on 32-bit chunks of the key increased hash bucket utilization to 82.0%. This three-line change resulted in an 8.95% ± 0.27% performance improvement. Figure 6 shows the rate of bucket collisions of the original hash function, the same hash function without the bit shifting "improvement," and our final hash function. The entire optimization required changing just three lines of code. This entire process, from profiling to a completed patch, took just two hours.

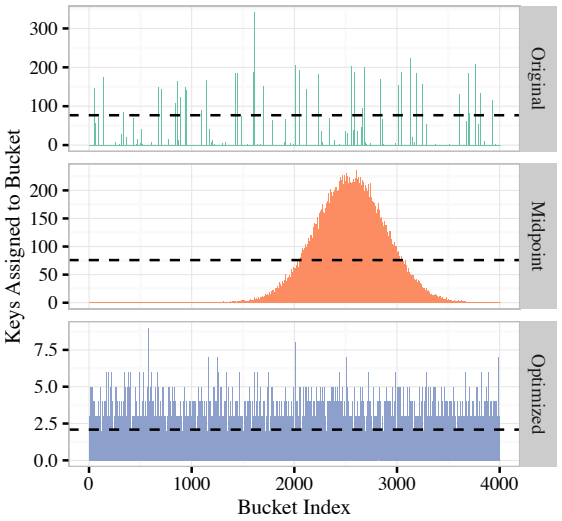

Figure 6: Hash collision rate before, during, and after performance tuning for a subset of dedup's hash buckets. Dashed black lines show the average number of items per utilized bucket. Note the different y-axes. Fixing dedup's hash function improved performance by 9%.

Case Study: SQLite

The SQLite database, which can be included as a single large C file, is used for many applications—including Firefox, Chrome, Safari, Opera, Skype, iTunes—and is a standard component of Android, iOS, Blackberry 10 OS, and Windows Phone 8. We evaluated SQLite's performance using a simple write-intensive parallel workload, where each thread rapidly inserts rows to its own private table. While this benchmark is synthetic, it exposes any scalability bottlenecks in the database engine itself because all threads should theoretically operate independently. This benchmark executes a progress point each time an insert to the database is completed.

COZ identified three important optimization opportunities, shown in Figure 7. Interestingly, the profile suggests that a small improvement to these lines' performance would speed up the program, but large performance improvements could actually be detrimental; this is evidence of contention elsewhere in the program. While resolving contention did not play a role in optimizing SQLite, contention is a factor in the next case study, which examines Memcached.

At startup, SQLite populates a large number of structs with function pointers to implementation-specific functions, but most of these functions are only ever given a default value determined by compile-time options. The three functions COZ identified unlock a standard pthread mutex, retrieve the next item from a shared page cache, and get the size of an allocated object. These simple functions do very little work, so the overhead of the indirect function call is relatively high, particularly because

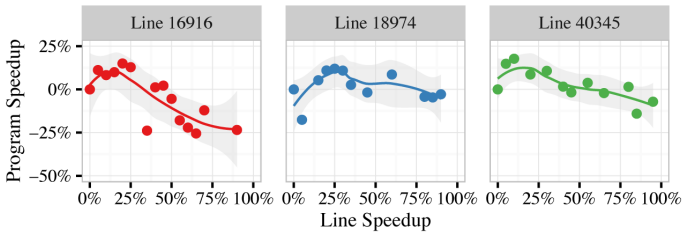

Figure 7: COZ's output for SQLite before optimizations

these functions are all likely candidates for inlining. Replacing these indirect calls with direct calls—which only required changes to seven lines of SQLite code—resulted in a 25.60% ± 1.00% speedup.

Case Study: Memcached

Memcached is a widely used in-memory key-value store, typically used as a cache in front of a database server. To evaluate Memcached's performance, we ran a version of the Redis performance benchmark ported to Memcached (available at https://github.com/antirez/mc-benchmark). This program spawns 50 parallel clients that collectively issue 100,000 `SET` and `GET` requests for a variety of keys. We added a progress point at the end of the `process_command` function in Memcached, which will execute after each client request is completed.

The vast majority of the source lines COZ profiles have virtually no potential for performance impact; this is hardly surprising given the level of performance tuning attention Memcached has received [5]. Excluding lines with little or no potential performance impact—which have a flat causal profile—most of the lines COZ identifies are cases of contention with a characteristic downward-sloping causal profile plot. This downward slope shows that optimizing this particular line of code would *hurt* rather than help program performance. If speeding up a line of code would hurt program performance, then some action that follows this line must contend with the program's critical path. One such line is at the start of the `item_remove` function, which locks an item in the cache, decrements its reference count, and frees the item if its reference count is zero.

To reduce lock-initialization overhead, Memcached uses a static array of locks to protect items, where each item selects a lock using a hash of its key. Consequently, locking any one item can potentially contend with independent accesses to other items whose keys happen to hash to the same lock index. However, Memcached uses atomic increment and decrement operations for reference counts; locking at this point is completely unnecessary. Resolving this issue along with two similar fixes required changing just six lines of code and resulted in a 9.39% ± 0.95% performance improvement.

Conclusion

Causal profiling is a radical departure from previous approaches to software profiling. Conventional profilers simply observe a program's execution, leaving developers to apply some intuition or a performance model to decide which parts of the program are important for performance. With a causal profiler, the program is the performance model. Instead of simply observing a program while attempting to minimize changes to that program's performance, a causal profiler *intentionally* alters program performance to conduct performance experiments. By carefully coordinating delays across a program's execution, a causal profiler can create the effect of optimizing a specific code fragment. By directly measuring the effect of a performance change, a causal profiler can tell developers exactly where optimizations will make a difference.

COZ is available at http://coz-profiler.org.

References

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architecture and Computation Techniques (PACT 2008)*, pp. 72–81: http://parsec.cs.princeton.edu/doc/parsec-report.pdf.

[2] Charlie Curtsinger and Emery D. Berger, "COZ: Finding Code that Counts with Causal Profiling," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 2015)*, pp. 184–197: http://dx.doi.org/10.1145/2815400.2815409.

[3] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, "gprof: a Call Graph Execution Profiler," in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126: http://dx.doi.org/10.1145/989393.989401.

[4] John D. C. Little, "A Proof for the Queueing Formula: L = λW," *Operations Research*, vol. 9, no. 3 (1961), pp. 383–387.

[5] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani, "Scaling Memcache at Facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pp. 385–398: https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf.