

Coz: Finding Code that Counts with Causal Profiling

By Charlie Curtsinger and Emery D. Berger

Abstract

Improving performance is a central concern for software developers. To locate optimization opportunities, developers rely on software profilers. However, these profilers only report where programs spend their time: optimizing that code may have no impact on performance. Past profilers thus both waste developer time and make it difficult for them to uncover significant optimization opportunities.

This paper introduces *causal profiling*. Unlike past profiling approaches, causal profiling indicates exactly where programmers should focus their optimization efforts, and quantifies their potential impact. Causal profiling works by running *performance experiments* during program execution. Each experiment calculates the impact of any potential optimization by *virtually speeding up* code: inserting pauses that slow down all other code running concurrently. The key insight is that this slowdown has the same *relative* effect as running that line faster, thus “virtually” speeding it up.

We present Coz, a causal profiler, which we evaluate on a range of highly-tuned applications such as Memcached, SQLite, and the PARSEC benchmark suite. Coz identifies previously unknown optimization opportunities that are both significant and targeted. Guided by Coz, we improve the performance of Memcached by 9%, SQLite by 25%, and accelerate six PARSEC applications by as much as 68%; in most cases, these optimizations involve modifying under 10 lines of code.

1. INTRODUCTION

Improving performance is a central concern for software developers. While compiler optimizations are of some assistance, they often do not have enough of an impact on performance to meet programmers’ demands.² Programmers seeking to increase the throughput or responsiveness of their applications thus must resort to manual performance tuning.

Manually inspecting a program to find optimization opportunities is impractical, so developers use profilers. Conventional profilers rank code by its contribution to total execution time. Prominent examples include oprofile, perf, and gprof.^{7,9,11} Unfortunately, even when a profiler accurately reports where a program spends its time, this information can lead programmers astray. Code that runs for a long time is not necessarily a good choice for optimization. For example, optimizing code that draws a loading animation during a file download will not make the program run faster, even though this code runs just as long as the download.

This phenomenon is not limited to I/O operations. Figure 1 shows a simple program that illustrates the shortcomings of existing profilers, along with its gprof profile as shown in Figure 2a. This program spawns two threads, which invoke functions f_a and f_b , respectively. Most profilers will report that these functions comprise roughly half of the total execution time. Other profilers may report that f_a is on the critical path, or that the main thread spends roughly equal time waiting for f_a and f_b . While accurate, all of this information is potentially misleading. Optimizing f_a away entirely will only speed up the program by 4.5% because f_b becomes the new critical path.

Conventional profilers do not report the potential impact of optimizations; developers are left to make these predictions based on their understanding of the program. While these predictions may be easy for programs as simple as the one shown in Figure 1, accurately predicting the effect of a proposed optimization is nearly impossible for programmers attempting to optimize large applications.

This paper introduces *causal profiling*, an approach that accurately and precisely indicates where programmers should focus their optimization efforts, and quantifies their potential impact. Figure 2b shows the results of running Coz, our prototype causal profiler. This profile plots the hypothetical speedup of a line of code (x-axis) versus its impact on execution time (y-axis). The graph correctly shows that optimizing either f_a or f_b in isolation would have little effect.

A causal profiler conducts a series of *performance experiments* to empirically observe the effect of a potential optimization. Of course it is not possible to automatically speed up any line of code by an arbitrary amount. Instead, a causal profiler uses the novel technique of *virtual speedups* to

Figure 1. A simple multithreaded program that illustrates the shortcomings of existing profilers. Optimizing f_a will improve performance by no more than 4.5%, while optimizing f_b would have no effect on performance.

```
example.cpp
1 void a() { // ~6.7 seconds
2   for (volatile size_t x=0; x<2000000000; x++) {}
3 }
4 void b() { // ~6.4 seconds
5   for (volatile size_t y=0; y<19000000000; y++) {}
6 }
7 int main() {
8   // Spawn both threads and wait for them.
9   thread a_thread(a), b_thread(b);
10  a_thread.join(); b_thread.join();
11 }
```

The original version of this paper was published in *Proceedings the ACM 2015 Symposium on Operating Systems Principles*, 184–197.

This work was initiated and partially conducted while Charlie Curtsinger was a Ph.D. student at the University of Massachusetts Amherst.

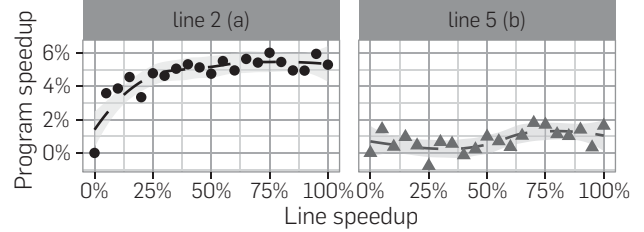
Figure 2. The gprof and causal profiles for the code in Figure 1. In the causal profile, the y-axis shows the program speedup that would be achieved by speeding up each line of code by the percentage on the x-axis. The gray area shows standard error. Gprof reports that f_a and f_b comprise similar fractions of total runtime, but optimizing f_a will improve performance by at most 4.5%, and optimizing f_b would have no effect on performance. The causal profile predicts both outcomes within 0.5%.

Conventional profile for `example.cpp`

% cumulative time	self seconds	self seconds	calls	self Ts/call	total Ts/call	name
55.20	7.20	7.20	1			a()
45.19	13.09	5.89	1			b()
% time	self	children	called	name		
55.0	7.20	0.00		a()		
45.0	5.89	0.00		b()		

(a) A gprof profile for `example.cpp`

Causal profile for `example.cpp`



(b) Causal profile for `example.cpp`

mimic the effect of optimizing a specific fragment of code by a fixed amount. Fragments could be functions, basic blocks, source lines. A fragment is virtually sped up by inserting pauses to slow all other threads each time the fragment runs. The key insight is that this slowdown has the same relative effect as running that fragment faster, thus “virtually” speeding it up. Figure 3 shows the equivalence of virtual and actual speedups.

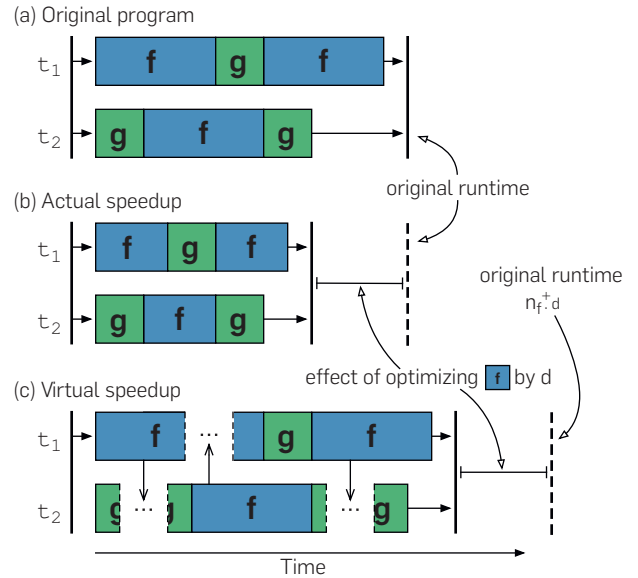
Each performance experiment measures the effect of virtually speeding up a fragment of code by a specific amount. Speedups are measured in percent change; a speedup of 0% means the fragment’s runtime is unchanged, while 75% means the fragment takes a quarter of its original runtime. By conducting many performance experiments over a range of virtual speedups, a causal profiler can predict the effect of any potential optimization on a program’s performance.

Causal profiling further departs from conventional profiling by making it possible to view the effect of optimizations on both *throughput* and *latency*. To profile throughput, developers specify a *progress point*, indicating a line in the code that corresponds to the end of a unit of work. For example, a progress point could be the point at which a transaction concludes, when a web page finishes rendering, or when a query completes. A causal profiler then measures the rate of visits to each progress point to determine any potential optimization’s effect on throughput. To profile latency, programmers instead place progress points at the start and end of an event of interest, such as when a transaction begins and completes. A causal profiler then reports the effect of potential optimizations on the average latency between those two progress points.

To demonstrate the effectiveness of causal profiling, we have developed Coz, a causal profiler for Linux. We show that causal profiling accurately predicts optimization opportunities, and that it is effective at guiding optimization efforts. We apply Coz to Memcached, SQLite, and the extensively studied PARSEC benchmark suite. Guided by Coz’s output, we improve the performance of Memcached by 9%, SQLite by 25%, and six PARSEC applications by as much as 68%. Our changes typically require modifying under 10 lines of code. We also show that Coz imposes low execution overhead. When it is possible, we compare the observed performance improvements to Coz’s predictions. In each case, we find that the real effect of our

Figure 3. An illustration of virtual speedup: (a) shows the original execution of two threads running functions f and g ; (b) shows the effect of a *actually* speeding up f by 40%; (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 (shown as ellipsis). The difference between the runtime in (c) and the original runtime plus $n_f \cdot d$ —the number of times f ran times the delay size—is the same as the effect of actually optimizing f .

Illustration of virtual speedup



optimization matches Coz’s prediction.

1.1. Contributions

This paper makes the following contributions:

1. It presents *causal profiling*, which identifies code where optimizations will have the largest impact. Using *virtual speedups* and *progress points*, causal profiling directly measures the effect of potential optimizations on both throughput and latency (Section 2).
2. It presents *Coz*, a causal profiler that works on unmodified Linux binaries. It describes Coz’s implementation (Section 3), and demonstrates its efficiency and effec-

tiveness at identifying optimization opportunities (Section 4).

2. CAUSAL PROFILING OVERVIEW

This section describes the major steps in collecting, processing, and interpreting a causal profile with Coz, our prototype causal profiler.

Profiler startup. A user invokes Coz using a command of the form `coz run --- <program> <args>`. At the beginning of the program's execution, Coz collects debug information for the executable and all loaded libraries to build a source map. By default, Coz will consider speedups in any source file from the main executable. This means Coz will only test potential optimizations in the main program's source files. Users should use binary and source file scope options to specify exactly which code they are willing or able to change to improve their program's performance. Once the source map is constructed, Coz creates a profiler thread and resumes normal execution.

Experiment initialization. Coz's profiler thread begins an experiment by selecting a line to virtually speed up, and a randomly-chosen percent speedup. Both parameters must be selected randomly; any systematic method of exploring lines or speedups could lead to systematic bias in profile results. One might assume that Coz could exclude lines or virtual speedup amounts that have not shown a performance effect early in previous experiments, but prioritizing experiments based on past results would prevent Coz from identifying an important line if its performance only matters after some warmup period. Once a line and speedup have been selected, the profiler thread records the number of visits to each progress point and begins the experiment.

Applying a virtual speedup. Every time the profiled program creates a thread, Coz begins sampling the instruction pointer from this thread. Coz processes samples within each thread to implement a sampling version of virtual speedups. In Section 3.4, we show the equivalence between the virtual speedup mechanism as shown in Figure 3, and the sampling approach used by Coz. Each thread periodically processes its own samples; threads check whether any samples fall within the line of code selected for virtual speedup. If so, all other threads must pause. This process continues until the profiler thread indicates that the experiment has completed.

Ending an experiment. Coz ends the experiment after a predetermined time has elapsed. If there were too few visits to progress points during the experiment—five is the default minimum—Coz doubles the experiment time for the rest of the execution. Once the experiment has completed, the profiler thread logs the results of the experiment, including the effective duration of the experiment (runtime minus the total inserted delay), the selected line and speedup, and the number of visits to all progress points. Before beginning the next experiment, Coz will pause for a brief cooloff period to allow any remaining samples to be processed before the next experiment begins.

Producing a causal profile. After an application has been profiled with Coz, the results of all the performance experiments can be combined to produce a causal profile. Each experiment has two independent variables: the line chosen for virtual

speedup and the amount of virtual speedup. Coz records the dependent variable, the rate of visits to each progress point, in two numbers: the total number of visits to each progress point and the effective duration of the experiment (the real runtime minus the total length of all pauses). Experiments with the same independent variables can be combined by adding the progress point visits and experiment durations.

Once experiments have been combined, Coz groups experiments by the line that was virtually sped up. Any lines that do not have a measurement of 0% virtual speedup are discarded; without this baseline measurement, we cannot compute a percent speedup relative to the original program. Measuring this baseline separately for each line guarantees that any line-dependent overhead from virtual speedups, such as the additional cross-thread communication required to insert delays when a frequently-executed line runs, will not skew profile results. By default, Coz also discards any lines with fewer than five different virtual speedup amounts (a plot that only shows the effect of a 75% virtual speedup is not particularly useful). Finally, we compute the percent program speedup for each grouped experiment as the percent change in rate of visits to each progress point over the baseline (virtual speedup of 0%). Coz then plots the resulting table of line and program speedups for each line, producing the profile graphs shown in this paper.

Interpreting a causal profile. Once causal profile graphs have been generated, it is up to the user to interpret them and make an educated choice about which lines may be possible to optimize. To help the user identify important lines, Coz sorts the graphs by the slope of their linear regression. Steep upward slopes indicate a line where optimizations will generally have a positive impact, while a flat line indicates that optimizing this line will not improve program performance. Coz also finds lines with a steep *downward* slope, meaning any optimization to this line will actually hurt performance. This downward sloping profile is a strong indicator of contention; the line that was virtually sped up interferes with the program's critical path, and optimizing this line increases the amount of interference. This phenomenon is surprisingly common, and can often result in significant optimization opportunities. In three of the eight applications we sped up using Coz—fluidanimate, streamcluster, and Memcached—the causal profile indicated contention issues. Fixing these issues resulted in speedups of 37.5%, 68.4%, and 9.4% respectively.

3. IMPLEMENTATION

This section describes Coz's basic functionality and implementation. We briefly discuss the core mechanisms required to support profiling unmodified Linux x86-64 executables, along with implementation details for each of the key components of a causal profiler: performance experiments, progress points, and virtual speedups.

3.1. Core mechanisms

Coz uses sampling to implement both virtual speedups and progress points. When a user starts a program with the `coz` command, Coz injects a profiling runtime library into the program's address space using `LD_PRELOAD`. This runtime library creates a dedicated profiler thread to run

performance experiments, but also intercepts each thread startup and shutdown to start and stop sampling in the thread using the `perf_event` API. Coz collects both the current program counter and user-space call stack from each thread every 1ms. To reduce overhead, Coz processes samples in batches of ten by default (every 10ms). Batching samples increases the time between when a thread runs a virtually sped-up function and delaying other threads, potentially introducing some inaccuracy in virtual speedup; however, processing samples less frequently reduces the time spent running Coz instead of the program being profiled. Processing samples more frequently is unlikely to improve accuracy, as the additional overhead would distort program execution.

Attributing samples to source locations. Coz uses DWARF debug information to map sampled program counter values to source locations. The profiled program does not need to contain DWARF line information; Coz will use the same search procedure as GNU Debugger (GDB) to locate external debug information if necessary.⁵ Note that debug information is available even for optimized code, and most Linux distributions offer packages that include debug information.

3.2. Performance experiments

Coz uses a dedicated profiler thread to coordinate performance experiments. This thread is responsible for selecting a line to virtually speed up, selecting the size of the virtual speedup, measuring the effect of the virtual speedup on progress points, and writing profiler output.

Starting a performance experiment. A single profiler thread, created during program initialization, coordinates performance experiments. First, the profiler selects a source line to virtually speed up. To do this, all program threads sample their instruction pointers and map these addresses to source lines. The first thread to sample a source line that falls within the specified profiling scope sets this as the line selected for virtual speedup.

Once the profiler receives a valid line from one of the program's threads, it chooses a random virtual speedup between 0% and 100%, in multiples of 5%. For any given virtual speedup, the effect on program performance is $1 - \frac{p_s}{p_0}$, where p_0 is the period between progress point visits with no virtual speedup, and p_s is the same period measured with some virtual speedup s . Because p_0 is required to compute program speedup for every p_s , a virtual speedup of 0 is selected with 50% probability. The remaining 50% is distributed evenly over the other choices.

Lines for virtual speedup must be selected randomly to prevent bias in the results of performance experiments. A naive approach would be to begin conducting performance experiments with small virtual speedups, gradually increasing the speedup until it no longer has an effect on program performance. Using this policy, a line that has no performance impact during a program's initialization would not be measured later in execution when optimizing it could have significant performance benefit. Any systematic approach to exploring the space of virtual speedup values could potentially lead to systematic bias in the profile output.

Once a line and speedup amount have been selected, Coz saves the current values of all progress point counters and begins the performance experiment.

Running a performance experiment. Once a performance experiment has started, each of the program's threads processes samples and inserts delays to perform virtual speedups. After the predetermined experiment time has elapsed, the profiler thread logs the end of the experiment, including the current time, the number and size of delays inserted for virtual speedup, the running count of samples in the selected line, and the values for all progress point counters. After a performance experiment has finished, Coz waits until all samples collected during the current experiment have been processed. By default, Coz processes samples in groups of ten, so this pause time is just ten times the sampling rate of 1ms. Lengthening this cooloff period will reduce Coz's overhead by inserting fewer delays at the cost of increased profiling time to conduct the same number of performance experiments.

3.3. Progress points

Before a program can be profiled with Coz, developers must add at least one progress point to the program. To indicate a progress point, a developer simply inserts the `COZ_PROGRESS` macro in the program's source code at the appropriate location. Coz also has experimental support for progress points that use sampling and breakpoints, which we describe in our Symposium on Operating Systems Principles (SOSP) paper.³

Measuring latency. Coz can also use progress points to measure the impact of an optimization on latency rather than throughput. To measure latency, a developer must specify two progress points: one at the start of some operation, and the other at the end. The rate of visits to the starting progress point measures the arrival rate, and the difference between the counts at the start and end points tells us how many requests are currently in progress. By denoting L as the number of requests in progress and λ as the arrival rate, we can solve for the average latency W via Little's Law, which holds for nearly any queuing system: $L = \lambda W$. Little¹² Rewriting Little's Law, we then compute the average latency as L/λ .

Little's Law holds under a wide variety of circumstances, and is independent of the distributions of the arrival rate and service time. The key requirement is that Little's Law only holds when the system is *stable*: the arrival rate cannot exceed the service rate. Note that all usable systems are stable: if a system is unstable, its latency will grow without bound.

3.4. Virtual speedup

A critical component of any causal profiler is the ability to virtually speed up any fragment of code. This section describes the derivation of virtual speedup using notation summarized in Table 1. A naive implementation of virtual speedups as shown in Figure 3; each time the function f runs, all other threads are paused briefly. If f has an average runtime of \bar{t}_f each time it is called and threads are paused for time d each time f runs, then f has an *effective* average runtime of $\bar{e}_f = \bar{t}_f - d$.

Table 1. Notation used in Section 3.4.

Table of notation	
\bar{t}_f	Average runtime of code fragment f
d	Length of delay inserted for virtual speedup
n_f	Total number of executions of code fragment f
\bar{e}_f	Effective average runtime of f with virtual speedup
P	Sampling period
s_f	Number of samples in code fragment f

If the *real* runtime of f was $\bar{t}_f - d$, but we forced every thread in the program to pause for time d after f ran (including the thread that just executed f) we would measure the same total runtime as with a virtual speedup. The only difference between virtual speedup and a real speedup with these additional pauses is that we use the time d to allow one thread to finish executing f . The pauses inserted for virtual speedup increase the total runtime by $n_f \cdot d$, where n_f is the total number of times f is called by any thread. Subtracting $n_f \cdot d$ from the total runtime with virtual speedup gives us the execution time we would measure if f had runtime $\bar{t}_f - d$.

Implementing virtual speedup with sampling. The previous discussion of virtual speedups assumes an implementation where each execution of small code fragment—a single line of code—causes all other threads instantaneously pause for a fraction of the time require to run the line. Unfortunately, this approach would incur prohibitively high overhead that would distort program execution. Instead, COZ periodically samples the program counter and counts samples that fall in the line selected for virtual speedup. Other threads are delayed proportionally to the number of samples. The number of samples in f with a sampling period of P is approximately,

$$s_f \approx \frac{n_f \cdot \bar{t}_f}{P}. \quad (1)$$

In our original model of virtual speedups, delaying other threads by time d each time the selected line is executed has the effect of shortening this line's runtime by d . With sampling, only some executions of the selected line will result in delays. The effective runtime of the selected line *when sampled* is $\bar{t}_f - d$, while executions of the selected line that are not sampled simply take time \bar{t}_f . The *effective* average time to run the selected line is,

$$\bar{e}_f = \frac{(n_f - s_f) \cdot \bar{t}_f + s_f \cdot (\bar{t}_f - d)}{n_f}. \quad (2)$$

Using (1), this reduces to,

$$\bar{e}_f = \frac{n_f \cdot \bar{t}_f \cdot \left(1 - \frac{\bar{t}_f}{P}\right) + \frac{n_f \cdot \bar{t}_f}{P} \cdot (\bar{t}_f - d)}{n_f} = \bar{t}_f \cdot \left(1 - \frac{d}{P}\right). \quad (3)$$

The relative difference between t and \bar{e}_f , the amount of virtual speedup, is simply,

$$\Delta \bar{t}_f = 1 - \frac{\bar{t}_f}{\bar{e}_f} = \frac{d}{P}. \quad (4)$$

This result lets COZ virtually speed up selected lines without instrumentation. Inserting a delay that is one quarter of the sampling period will virtually speed up the selected line by 25%.

Pausing threads. When one thread receives a sample in the line selected for virtual speedup, all other threads must pause. Rather than using POSIX signals, which would have prohibitively high overhead, COZ controls inter-thread pausing using counters. The first counter, shared by all threads, records the number of times each thread should have paused so far. Each thread has a local counter of the number of times that thread has already paused. Whenever a thread's local count of pauses is less than the number of required pauses in the global counter, a thread must pause (and increment its local counter). To signal all other threads to pause, a thread simply increments both the global counter and its own local counter. Every thread checks if pauses are required after processing its own samples.

Thread creation. To start sampling and adjust delays, COZ intercepts calls to the `pthread_create` function. When a new thread is created, COZ first begins sampling in the new thread. It then inherits the parent thread's local delay count; any previously inserted delays to the parent thread also delayed the creation of the new thread.

Handling suspended threads. COZ only collects samples and inserts delays in a thread while that thread is actually executing. As a result, a backlog of required delays will accumulate in a thread while it is suspended. When a thread is suspended on a blocking I/O operation, this is the desired behavior; pausing the thread while it is already suspended on I/O would not delay the thread's progress. COZ simply adds these delays after the thread unblocks.

However, a thread can also be suspended while waiting for another thread using pthreads synchronization operations. As with blocking I/O, required delays will accumulate while the thread is suspended but COZ may not need to insert all of these delays when the thread resumes. When a thread resumes after waiting on a lock, another thread must have released the lock. If the unlocking thread has executed all the required delays, then the blocked thread has effectively already been delayed. The suspended thread should be credited for any delays inserted in the thread responsible for waking it up. Otherwise, the thread should insert all the necessary delays that accumulated during the time the thread was suspended. To implement this policy, COZ forces threads to execute all required delays before blocking or waking other threads.

Attributing samples to source lines. Samples are attributed to source lines using the source map constructed at startup. When a sample does not fall in any in-scope source line, the profiler walks the sampled callchain to find the first in-scope address. This policy has the effect of attributing all out-of-scope execution to the last in-scope callsite responsible. For example, a program may call `printf`, which calls `vfprintf`, which in turn calls `strlen`. Any samples collected during this chain of calls will be attributed to the source line that issues the original `printf` call.

Optimizations & phase correction. Coz attempts to minimize the number of delays inserted when all threads must be paused. Minimizing delays reduces the performance overhead of profiling without changing the accuracy of virtual speedups. The profile analysis also includes a correction for programs with phases; if left uncorrected, phases could overstate the potential effect of an optimization. For details, see our SOSP paper.³

4. EVALUATION

Our evaluation answers the following questions: (1) Does causal profiling enable effective performance tuning? (2) Are Coz's performance predictions accurate?, and (3) Is Coz's overhead low enough to be practical?.

4.1. Experimental setup

We perform all experiments on a 64 core, four socket Advanced Micro Devices (AMD) Opteron machine with 60GB of memory, running Linux 3.14 with no modifications. All applications are compiled using GNU Compiler Collection (GCC) version 4.9.1 at the `-O3` optimization level and debug information generated with `-g`. We disable frame pointer elimination with the `-fno-omit-frame-pointer` flag so Linux can collect accurate call stacks with each sample. Coz is run with the default sampling period of 1ms, with sample processing set to occur after every 10 samples. Each performance experiment runs with a cooling-off period of 10ms after each experiment to allow any remaining samples to be processed before the next experiment begins. Due to space limitations, we only profile throughput (and not latency) in this evaluation.

4.2. Effectiveness

We demonstrate causal profiling's effectiveness through case studies. Using Coz, we collect causal profiles for Memcached, SQLite, and the PARSEC benchmark suite. Using these causal profiles, we were able to make small changes to two of the real applications and six PARSEC benchmarks, resulting in performance improvements as large as 68%. Table 2 summarizes the results of our optimization efforts. We describe some of our experience using Coz here.

Case study: SQLite. The SQLite database library is widely used by many applications to store relational data. This embedded database, which can be included as a single large C file, is used by many applications such as Firefox, Chrome, Safari, Opera, Skype, iTunes, and is a standard component of Android and iOS. We evaluated SQLite performance using a 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. We placed a progress point in the benchmark itself (which is linked with the database), which executes after each insertion.

Coz identified three important optimization opportunities, as shown in Figure 4a. At startup, SQLite populates a large number of structs with function pointers to

Table 2. All benchmarks were run 10 times before and after optimization. Standard error for speedup was computed using Efron's bootstrap method. All speedups are statistically significant at the 99.9% confidence level ($\alpha = 0.001$) using the one-tailed Mann-Whitney U test. Diff size reports the number of lines removed and added.

Summary of optimization results			
Application	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

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. Replacing these indirect calls with direct calls resulted in a $25.60\% \pm 1.00\%$ speedup.

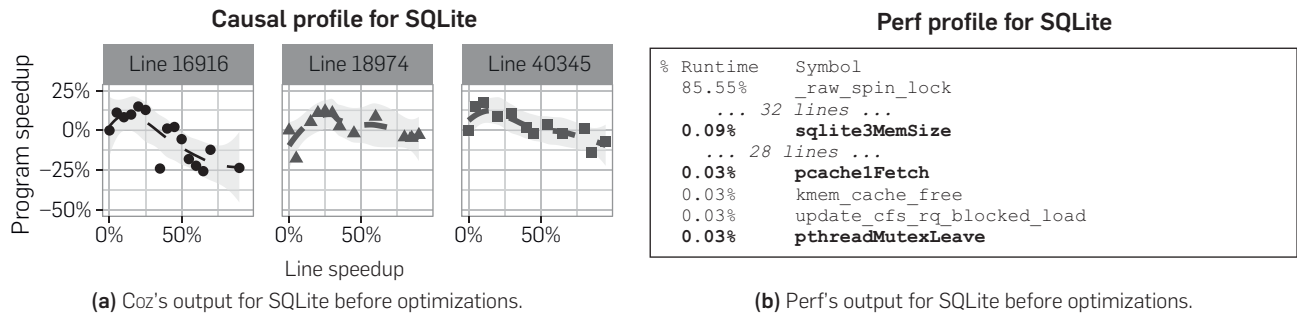
Comparison with conventional profilers. Unfortunately, running SQLite with gprof segfaults immediately. The application does run with the Linux perf tool, which reports that the three functions Coz identified account for a total of just 0.15% of total runtime (Figure 4b). Using perf, a developer would be misled into thinking that optimizing these functions would be a waste of time. Coz accurately shows that the opposite is true: optimizing these functions has a dramatic impact on performance.

Case study: Memcached. Memcached is a widely-used in-memory caching system. To evaluate cache performance, we ran a benchmark ported from the Redis performance benchmark. This program spawns 50 parallel clients that collectively issue 100,000 SET and GET requests for randomly chosen keys. We placed a progress point at the end of the `process_command` function, which handles each client request.

Most of the lines Coz identifies are cases of contention, with a characteristic downward-sloping causal profile plot. One such line is at the start of `item_remove`, which locks an item in the cache and then decrements its reference count, freeing it if the count goes to zero. To reduce lock initialization overhead, Memcached uses a static array of locks to protect items, where each item selects its 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. Reference counts are updated atomically inside this critical section, and the thread that decrements the reference count to zero has exclusive access to this item. As a result, we can safely remove the lock from this function, which yielded a $9.39\% \pm 0.95\%$ speedup.

Case study: Dedup. The dedup application performs

Figure 4. Coz and perf output for SQLite before optimizations. The three lines in the causal profile correspond to the function prologues for `sqlite3MemSize`, `pthreadMutexLeave`, and `pcache1Fetch`. A small optimization to each of these lines will improve program performance, but beyond about a 25% speedup, Coz predicts that the optimization would actually lead to a slowdown. Changing indirect calls into direct calls for these functions improved overall performance by $25.6\% \pm 1.0\%$. While the perf profile includes the functions we changed, they account for just 0.15% of the sampled runtime.



parallel file compression via deduplication. This process is divided into three main stages: fine-grained fragmentation, hash computation, and compression. We placed a progress point immediately after dedup completes compression of a single block of data (`encoder.c:189`).

Coz identifies the source line `hashtable.c:217` as the best opportunity for optimization. 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. These results suggest that dedup's shared hash table has a significant number of collisions. Increasing the hash table size had no effect on performance. This discovery led us to examine dedup's hash function, which could also be responsible for the large number of hash table collisions. We discovered that dedup's hash function maps keys to just 2.3% of the available buckets; over 97% of buckets were never used during the entire execution.

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. We removed the bit shifting step, which increased hash table utilization to 54.4%. We then changed the hash function to bitwise XOR 32 bit chunks of the key. Our new hash function increased hash table utilization to 82.0% and resulted in an $8.95\% \pm 0.27\%$ performance improvement. The entire optimization required changing just three lines of code, and the entire profiling and tuning effort took just two hours.

Comparison with gprof. We ran both the original and optimized versions of dedup with gprof. The optimization opportunities identified by Coz were not obvious in gprof's output. Overall, `hashtable_search` had the largest share of highest execution time at 14.38%, but calls to `hashtable_search` from the hash computation stage accounted for just 0.48% of execution time; Gprof's call graph actually obscured the importance of this code. After optimization, `hashtable_search`'s share of execution time reduced to 1.1%.

Case study: Fluidanimate and streamcluster. The fluidanimate and streamcluster benchmarks use a similar ap-

proach to parallelism. The application spawns worker threads that execute in a series of concurrent phases, with each phase separated from the next by a barrier. We placed a progress point immediately after the barrier, so it executes each time all threads complete a phase of the computation.

In both benchmarks, Coz also identified the same points of contention: PARSEC's barrier implementation. Figure 5 shows the evidence for this contention in fluidanimate. This profile tells us that optimizing the indicated line of code would actually *slow down* the program, rather than speed it up. Both lines run immediately before entering a loop that repeatedly calls `pthread_mutex_trylock`. Removing this spinning from the barrier would reduce contention, but it was easier to replace the custom barrier with the default `pthread_barrier` implementation. This one line change led to a $37.5\% \pm 0.56\%$ speedup in fluidanimate, and a $68.4\% \pm 1.12\%$ speedup in streamcluster.

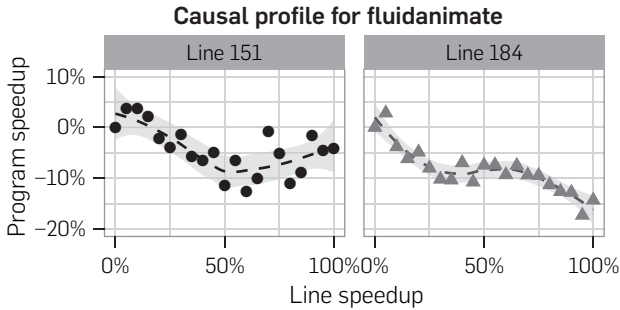
Effectiveness Summary. Our case studies confirm that Coz is effective at identifying optimization opportunities and guiding performance tuning. In every case, the information Coz provided led us directly to the optimization we implemented. In most cases, Coz identified around 20 lines of interest, with as many as 50 for larger programs (Memcached and x264). Coz identified optimization opportunities in all of the PARSEC benchmarks, but some required more invasive changes that are out of scope for this paper.

4.3. Accuracy

For most of the optimizations described above, it is not possible to quantify the effect our optimization had on the specific lines that Coz identified. However, for two of our case studies—ferret and dedup—we can directly compute the effect our optimization had on the line Coz identified and compare the resulting speedup to Coz's predictions. Our results show that Coz's predictions are highly accurate.

For dedup, Coz identified the top of the while loop that traverses a hash bucket's linked list. By replacing the degenerate hash function, we reduced the average number of elements in each hash bucket from 76.7 to 2.09. This change reduces the number of iterations from 77.7 to 3.09 (accounting for the final trip through the loop). This reduction

Figure 5. Coz output for fluidanimate, prior to optimization. Coz finds evidence of contention in two lines in `parsec_barrier.cpp`, the custom barrier implementation used by both fluidanimate and stream-cluster. This causal profile reports that optimizing either line will slow down the application, not speed it up. These lines precede calls to `pthread_mutex_trylock` on a contended mutex. Optimizing this code would increase contention on the mutex and interfere with the application's progress. Replacing this inefficient barrier implementation sped up fluidanimate and streamcluster by 37.5% and 68.4% respectively.



corresponds to a speedup of the line Coz identified by 96%. For this speedup, Coz predicted a performance improvement of 9%, very close to our observed speedup of 8.95%. Results for ferret are similar; Coz predicted a speedup of 21.4%, and we observe an actual speedup of 21.2%.

4.4. Efficiency

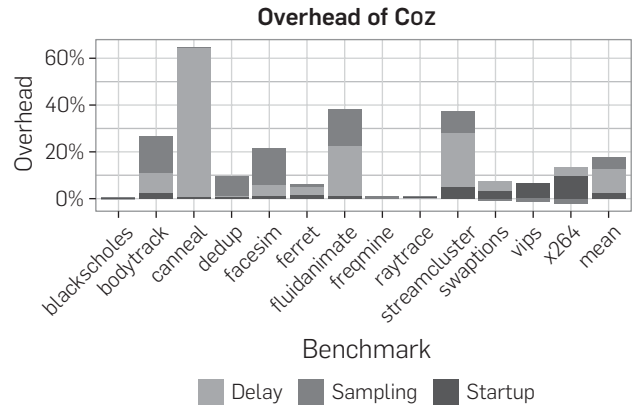
We measure Coz's profiling overhead on the PARSEC benchmarks running with the native inputs. The sole exception is streamcluster, where we use the test inputs because execution time was excessive with the native inputs.

Figure 6 breaks down the total overhead of running Coz on each of the PARSEC benchmarks by category. The average overhead with Coz is 17.6%. Coz collects debug information at startup, which contributes 2.6% to the average overhead. Sampling during program execution and attributing these samples to lines using debug information is responsible for 4.8% of the average overhead. The remaining overhead (10.2%) comes from the delays Coz inserts to perform virtual speedups.

These results were collected by running each benchmark in four configurations. First, each program was run without Coz to measure a baseline execution time. In the second configuration, each program was run with Coz, but execution terminated immediately after startup work was completed. Third, programs were run with Coz configured to sample the program's execution but not to insert delays (effectively testing only virtual speedups of size zero). Finally, each program was run with Coz fully enabled. The difference in execution time between each successive configuration give us the startup, sampling, and delay overheads, respectively.

Reducing overhead. Most programs have sufficiently long running times (mean: 103s) to amortize the cost of processing debug information, but especially large executables can be expensive to process at startup (e.g., x264 and vips). Coz could be modified to collect and process debug information lazily to reduce startup overhead. Sampling overhead comes

Figure 6. Percent overhead for each of Coz's possible sources of overhead. Delay is the overhead from adding delays for virtual speedups, Sampling is the cost of collecting and processing samples, and Startup is the initial cost of processing debugging information. Note that sampling results in slight performance improvements for swaptions, vips, and x264.



mainly from starting and stopping sampling with the `perf_event` API at thread creation and exit. This cost could be amortized by sampling globally instead of per-thread, which would require root permissions on most machines. If the `perf_event` API supported sampling all threads in a process, this overhead could be eliminated. Delay overhead, the largest component of Coz's total overhead, could be reduced by allowing programs to execute normally for some time between each experiment. Increasing the time between experiments would significantly reduce overhead, but a longer profiling run would be required to collect a usable profile.

Efficiency summary. Coz's profiling overhead is on average 17.6% (minimum: 0.1%, maximum: 65%). For all but three of the benchmarks, its overhead was under 30%. Given that the widely used gprof profiler can impose much higher overhead (e.g., 6 times for ferret, versus 6% with Coz), these results confirm that Coz has sufficiently low overhead to be used in practice.

5. RELATED WORK

Causal profiling differs from past profiling techniques, which have focused primarily on collecting as much detailed information as possible about a program without disturbing its execution. Profilers have used a wide variety of techniques to gather different types of information in different settings, which we summarize here.

5.1. General-purpose profilers

General-purpose profilers are designed to monitor where a program spends its execution time. Profilers such as gprof and oprofile are typical of this category.^{7,11} While oprofile uses sampling exclusively, gprof mixes sampling and instrumentation to measure both execution time and collect call graphs, which show how often each function was called, and where it was called from. Later extensions to this work have reduced the overhead of call graph profiling and added additional detail with path profiling, but

the functionality of general-purpose profilers remains largely unchanged. This kind of profiling information is useful for identifying where a program spends its time, but not necessarily where developers should work to improve performance.

5.2. Parallel profilers

Several techniques have been used to identify performance and scalability bottlenecks in parallel programs. Systems such as IPS trace the execution of a running program to identify its *critical path*, the longest sequence of dependencies in the complete program dependence graph.¹³ While this approach can work well for message-passing systems, it would require instrumenting all memory accesses in a modern shared-memory parallel program; this would impose substantial overhead, likely distorting the results far too much to be representative of an un-profiled execution.

Other parallel profilers, such as FreeLunch and the WAIT tool, identify code that runs while some of a program's threads sit idle.^{1,4} These systems assign some level of blame for blocking to all of a program's code. The idea is that code running while other threads are blocked must be responsible for the reduced parallelism. This heuristic works well for some parallel performance issues, but not all performance bottlenecks change a thread's scheduler state.

5.3. Profiling for scalability

Several systems have been developed to measure potential parallelism in serial programs.^{6,16,17} Other systems instead examine parallel programs to predict how well the program will scale to larger numbers of hardware threads.¹⁰ These approaches are distinct and complimentary to causal profiling. These tools help developers parallelize and scale applications, while Coz helps developers improve an existing parallel program at the current level of parallelism.

5.4. Performance experimentation


Coz is a significant departure from past profiling techniques in that it intentionally perturbs a program's execution to model the effect of an optimization. While this technique is unique for software profilers, the idea of a performance experiment has appeared in other systems. Mytkowicz et al.¹⁴ use delays to validate the output of profilers on single-threaded Java programs.¹⁴ Snelick et al.¹⁵ use delays to profile parallel programs.¹⁵ This approach measures the effect of slowdowns in combination, which requires a complete execution of the program for each of an exponential number of configurations. While these techniques involve performance experiments, Coz is the first system to use performance perturbations to create the effect of an optimization.

6. CONCLUSION

Profilers are the primary tool in the programmer's toolbox for identifying performance tuning opportunities. Previous profilers only observe actual executions and correlate code with execution time or performance counters. This information can be of limited use because the amount of time spent does not necessarily correspond to where programmers

should focus their optimization efforts. Past profilers are also limited to reporting end-to-end execution time, an unimportant quantity for servers and interactive applications whose key metrics of interest are throughput and latency. Causal profiling is a new, experiment-based approach that establishes causal relationships between hypothetical optimizations and their effects. By virtually speeding up lines of code, causal profiling identifies and quantifies the impact on either throughput or latency of any degree of optimization to any line of code. Our prototype causal profiler, Coz, is efficient, accurate, and effective at guiding optimization efforts. Coz is now a standard package on current Debian and Ubuntu platforms and can be installed via the command `sudo apt-get install coz-profiler` or it can be installed from source on any Linux distribution; all source is at <http://coz-profiler.org>.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CCF-1012195 and CCF-1439008. Charlie Curtsinger was supported by a Google PhD Research Fellowship. The authors thank Dan Barowy, Steve Freund, Emma Tosch, John Vilk, and Tim Harris for their feedback and helpful comments. 

References

1. Altman, E.R., Arnold, M., Fink, S., Mitchell, N. Performance analysis of idle programs. *OOPSLA. ACM* (2010), 739–753.
2. Curtsinger, C., Berger, E.D. Stabilizer: Statistically sound performance evaluation. In *ASPLOS* (New York, NY, USA, 2013), ACM.
3. Curtsinger, C., Berger, E.D. Coz: Finding code that counts with causal profiling. In *SOSP*, (ACM, New York, NY, 2015), 184–197.
4. David, F., Thomas, G., Lawall, J., Muller, G. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*, (ACM, New York, NY, 2014), 291–307.
5. Free Software Foundation. *Debugging with GDB*, 10th edn., The Free Software Foundation, Boston, MA.
6. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI*, (ACM, New York, NY, 2011), 458–469.
7. Graham, S.L., Kessler, P.B., McKusick, M.K. Gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, (ACM, New York, NY, 1982), 120–126.
8. Intel. *Intel VTune Amplifier*, 2015.
9. kernel.org. *perf: Linux profiling with performance counters*, 2014.
10. Kulkarni, M., Pai, V.S., Schuff, D.L. Towards architecture independent metrics for multicore performance analysis. *SIGMETRICS Performance Evaluation Review* 38, 3 (2010), 10–14.
11. Levon, J., Elie, P. Oprofile: A system profiler for Linux, 2004. <http://oprofile.sourceforge.net/>.
12. Little, J.D. OR FORUM: Little's Law as Viewed on Its 50th Anniversary. *Operations Research* 59, 3 (2011), 536–549.
13. Miller, B.P., Yang, C.-Q. IPS: An interactive and automatic performance measurement tool for parallel and distributed programs. In *ICDCS*, 1987, 482–489.
14. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F. Evaluating the accuracy of Java profilers. In *PLDI* (2010) ACM, 187–197.
15. Snelick, R., JáJá, J., Kacker, R., Lyon, G. Synthetic-perturbation techniques for screening shared memory programs. *Software Practice & Experience* 24, 8 (1994), 679–701.
16. von Praun, C., Bordawekar, R., Cascaval, C. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP* (2008), ACM, 185–196.
17. Zhang, X., Navabi, A., Jagannathan, S. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO* (2009), IEEE Computer Society, 47–58.

Charlie Curtsinger (curtsinger@grinnell.edu), Department of Computer Science, Grinnell College, USA.

Emery D. Berger (emery@cs.umass.edu), College of Information and Computer Sciences, University of Massachusetts Amherst, USA.