Coz: Finding Code that Counts with Causal Profiling

Charlie Curtsinger  Emery D. Berger
School of Computer Science
University of Massachusetts Amherst
{charlie,emery}@cs.umass.edu

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 spent 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, and evaluate it on a range of highly-tuned applications: 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 [11]. Programmers seeking to increase the throughput or responsiveness of their applications thus must resort to manual performance tuning.

Since manually inspecting an entire program to find optimization opportunities is impractical, developers use profilers. Conventional profilers rank code by its contribution to total execution time. Prominent examples include oprofile, perf, and gprof [19][28][40].

Unfortunately, even when a profiler accurately reports where a program is spending 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 any faster, even though this code runs just as long as the file 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 in Figure 2a. This program spawns two threads, which invoke functions a and b respectively. Most profilers will report that these functions comprise roughly half of the total execution time. Other profilers may report that the a function is on the critical path, or that the main thread spends roughly equal time waiting for a_thread and b_thread [24]. While accurate, all of this information is potentially misleading. Optimizing a away entirely will only speed up the program by 4.5% because b becomes the new critical path.

Existing profilers do not report the potential impact of optimizations; developers are left to make these predictions given their understanding of the program. While these predictions may be easy for programs as simple as the one in Figure 1, accurately predicting the performance impact 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 a or b in isolation would have little impact on execution time.

A causal profiler conducts a series of performance experiments to empirically observe the impact of a potential optimization. Of course, it is not possible to automatically speedup any line of code by an arbitrary amount. Instead, during a performance experiment, the causal profiler uses the novel technique of virtual speedups to mimic the effect of optimizing a specific line of code by a specific amount.

Virtual speedup works by inserting pauses that slow down all code running at the same time as the line under examination. The key insight is that this slowdown has the same relative effect as running that line faster, thus “virtually” speeding it up. Figure 3 illustrates the relative equivalence between actual and virtual speedups: after accounting for delays, both have the same impact.

Each performance experiment measures the impact of some amount of virtual speedup to a single line. By sampling over the range of virtual speedup from between 0% (no change) and 100% (the line is completely eliminated), causal profiling can calculate the impact of any potential optimization on overall performance.

Causal profiling further departs from traditional profiling by making it possible to view the effect of optimizations on throughput and latency. To profile throughput, programmers place two progress points that correspond to 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.

We demonstrate causal profiling with COZ, a prototype causal profiler that works with Linux x86-64 binaries. We show that COZ imposes low execution time overhead (mean: 17%, min: 0.1%, max: 65%), making it substantially faster than gprof (up to 6× overhead).

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 optimized the performance of Memcached by 9%, SQLite by 25%, and six PARSEC applications by as much as 68%. These optimizations typically involved modifying under 10 lines of code. When possible to accurately measure the size of our optimizations on the line(s) identified by COZ, we compare the observed performance improvements to COZ’s predictions: in each case, we find that the real effect of our optimization matched COZ’s prediction.

**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 (§2).

2. It presents COZ, a causal profiler that works on unmodified Linux binaries. It describes COZ’s implementation (§3), and demonstrates its efficiency and effectiveness at identifying optimization opportunities (§4).
2. Causal Profiling Overview

Causal profiling relies on several key ideas to provide developers with actionable profiles. Virtual speedups let a causal profiler automatically create the effect of optimizing any fragment of code. Progress points let the profiler measure a program’s performance repeatedly during one run. Performance experiments apply a virtual speedup and measure the resulting effect on performance. Repeated performance experiments enable a causal profiler to identify fragments of code where optimizations will have the greatest impact. This section provides a detailed description of these key concepts, and describes the workflow of Coz, our prototype causal profiler.

Virtual speedups. A virtual speedup uses delays to create the effect of optimizing a fragment of code. Each time a selected fragment is executed, all other threads are briefly paused. The longer the pause, the larger the relative speedup. At the end of an execution, causal profiling subtracts the total pause time from runtime to determine the effective execution time. This technique is illustrated in Figure 3.

Progress points. A causal profiler uses progress points to measure program performance during execution. Developers must place progress points at a source location where some useful work has been completed. These points let a causal profiler conduct many performance experiments during a single run. Additionally, progress points enable measurement of both latency and throughput, and enable profiling of long-running applications where end-to-end execution time is meaningless.

Performance experiments. A causal profiler runs many performance experiments during a program’s execution. For each experiment, the profiler randomly selects a fragment of code to virtually speed up for the duration of the experiment. Meanwhile, the profiler measures the rate of visits to one or more progress points. Each performance experiment establishes the impact of optimizing a particular code fragment by a specific amount. Given a sufficient number of experiments, the profiler can identify which fragments will yield the largest performance gains if optimized.

A causal profiler can also identify contention, which appears as a downward sloping line on a causal profile graph. A negative slope indicates that optimizing the code fragment will hurt application performance. We find and address several instances of contention in our case studies in Section 4.

2.1 Causal Profiling Workflow

To demonstrate the effectiveness of causal profiling, we have implemented Coz, a prototype causal profiler. Coz implements all of the key components of a causal profiler: virtual speedups, progress points, and performance experiments. Coz identifies optimization opportunities at the granularity of source lines, but our technique can easily support any type of code fragment. We describe Coz’s profiling workflow in detail below.

![Illustration of Virtual Speedup](image)

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 speedup f by 40%; (c) shows the effect of virtually speeding up f by 40% by pausing the other thread each time f runs. Each inserted pause (dark gray) is equal to the size of the speedup—40% of f’s execution time (light blue). The runtime of (c) is longer than (b) by the total pause time. Adjusting the baseline runtime from (a) by the total delay time lets us measure the virtual speedup size, which matches the effect of the actual speedup.

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. Users may specify file and binary scope, which restricts Coz’s experiments to speedups in the specified files. By default, Coz will consider speedups in any source file from the main executable. Coz builds a map from instructions to source lines using the program’s debug information and the specified scope. 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. Once a line and speedup have been selected, the profiler thread saves 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 described above and the sampling approach implemented in CoZ. Every time a sample is available, a thread checks whether the sample falls in the line of code selected for virtual speedup. If so, it forces other threads to pause. This process continues until the profiler thread indicates that the experiment has completed.

**Ending an experiment.** CoZ ends the experiment after a pre-determined 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.

### 3. Implementation

The current implementation of CoZ profiles Linux x86-64 executable binaries. To map program addresses to source lines, CoZ uses DWARF debugging information. As long as debug information is available in a separate file, CoZ can profile optimized and stripped executables. Sampling is implemented using the perf_event API.

#### 3.1 Profiler Startup

The CoZ profiling code is inserted into a process using the LD_PRELOAD environment variable. This allows CoZ to intercept library calls from the program, including the libc_start_main function, which runs before main and all global constructors. Before the program’s normal execution begins, CoZ collects the names and locations of all loaded executables by reading /proc/self/maps. CoZ records the loaded address and path to each in-scope executable for later processing.

For all in-scope executables and libraries, CoZ locates DWARF debug information for the program’s main executable and libraries [15]. By default, the scope includes all source files from the main executable, but alternate source locations and libraries can be specified on the command line. If any debug information has been stripped, CoZ uses the same procedure as Gdb to search standard system paths for separate debugging information [16]. Note that debug information is available even for optimized code, and most Linux distributions offer packages that include this information for common libraries.

CoZ uses DWARF line tables to build a map from instruction pointer ranges to source lines. The DWARF format also includes both caller and callee information for inlined procedures. Special handling is required when an in-scope call site is replaced by an inlined function that is not in scope. The inlined function’s address range is assigned to the caller’s source location in the source map. This approach mirrors the process by which CoZ attributes out-of-scope samples to callsites during execution (see the discussion of sample attribution, below).

**Enabling Sampling.** Before calling the program’s main function, CoZ opens a perf_event file to start sampling in the main thread. CoZ invokes the perf_event_open system call to track high precision timer events via a memory-mapped file. CoZ samples each thread individually using the high precision timer event, and collects instruction pointers and the user-space callchain in each sample.

**Sample Attribution.** 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 process 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.

#### 3.2 Experiment Initialization

A single profiler thread, created during program initialization, coordinates performance experiments. Before a performance experiment can begin, a line must be selected for virtual speedup. When an experiment is not running, each program thread will set the next_line atomic variable to its most recent sample. The profiler thread spins until this variable contains a non-null value.

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 $1 - \frac{p_o}{p_s}$, 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 virtual speedup amounts.

Virtual speedups must be selected randomly to prevent bias in the results of performance experiments. A seemingly reasonably (but invalid) 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. However, this approach may both over- and under-state the impact of optimizing a particular line if its impact varies over time.

For example, 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. Conversely, a line that only affects performance during initialization would have exaggerated performance impact unless future experiments re-evaluate virtual speedup values.
for this line during normal execution. 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 virtual speedup have been selected, CoZ
saves the current values of all progress point counters and
begins the performance experiment.

3.3 Running a Performance Experiment

Once a performance experiment has started, each of the pro-
gram’s threads processes samples and inserts delays to per-
form virtual speedups. After the pre-determined experiment
time has elapsed, the profiler thread logs the end of the ex-
periment, 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 at least 10ms before starting another experiment.
This pause ensures that delays and samples processed by
threads around the end of the experiment are not accidentally
attributed to the next experiment, which would bias results.

3.4 Virtual Speedups

CoZ uses delays to create the effect of optimizing the selected
line. Every time one thread executes this line, all other threads
must pause. The length of the pause determines the amount
of virtual speedup; pausing other threads for half the selected
line’s runtime has the effect of optimizing the line by 50%.

Implementing Virtual Speedup. Tracking every visit to the
selected line would incur significant performance overhead
and distort the program’s execution. Instead, CoZ uses sam-
ping to implement virtual speedups accurately and efficiently,
delaying proportionally to the time spent in the selected line.
This lets CoZ virtually speed up the line by a specific percent,
even though the number of visits to the line is unknown.

The expected number of samples in the selected line, \( s \), is

\[
E[s] = \frac{n \cdot t}{P}
\]

where \( P \) is the period of time between samples, \( t \) is the time
required to run the selected line once, and \( n \) is the number of
times the selected line is executed.

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 \( t - d \), while executions of the selected line that are
not sampled simply take time \( t \). The average effective time to
run the selected line is

\[
t' = \frac{(n - s) \cdot t + s \cdot (t - d)}{n}
\]

Using (1), this reduces to

\[
t' = \frac{n \cdot t \cdot (1 - \frac{d}{P}) + \frac{n \cdot t}{P} \cdot (t - d)}{n} = t \cdot (1 - \frac{d}{P})
\]

The percent difference between \( t \) and \( t' \), the amount of
virtual speedup, is simply

\[
\Delta t = 1 - \frac{t'}{t} = \frac{d}{P}.
\]

This result lets CoZ virtually speed up selected lines by
a specific amount without instrumentation. Inserting a delay
that is half the sampling period will virtually speed up the
selected line by 50%.

Pausing Other Threads. When one thread receives a sam-
ples in the line selected for virtual speedup, all other threads
must pause. CoZ triggers these pauses using two counters: a
shared global counter, and per-thread local counters. These
counters are used to pause threads without using expensive
POSIX signals. The global counter stores the number of
pauses each thread should execute, while per-thread local
counters track the number of pauses each thread has executed
so far. To pause all other threads, a thread increments both
counters. Every thread checks the counters after each sample.
If a thread’s local delay count is less than the global delay
count, it must pause and increment its local counter. Each
thread checks its counter against the global count and inserts
any required delays immediately after processing samples.

Ensuring Accurate Timing. CoZ uses the nanosleep
POSIX function to insert delays. This function only guaran-
tees that the thread will pause for at least the requested time,
but the pause may be longer than requested. CoZ tracks any
excess pause time, which is subtracted from future pauses.

Thread Creation. To start sampling and adjust delays, CoZ
interposes on the pthread_create function. CoZ first
initiates perf_event sampling in the new thread. It then
copies the parent thread’s local delay count, propagating any
delays: any previously inserted delays to the parent thread
also delayed the creation of the new thread.

Thread Sampling and Delay Accounting. CoZ only inter-
rupts a thread to process samples if the thread is running. If
the thread is blocked on I/O, sample processing and delays
will be performed after the blocking call returns. For blocking
I/O, this is the desired behavior—inserting pauses during a
file read would have no effect on the time it takes to complete
the read. However, threads can also block on other threads,
which complicates delay insertion.

Consider a program with two threads: thread A is currently
holding a mutex, and thread B is waiting to acquire the mutex.
If thread B is spinning on the mutex, delaying that thread
will not necessarily have any effect on how long it waits.
Unlike with blocking I/O, this is actually the desired behavior:
thread A will have inserted these delays, which delays the
time that thread A unlocks the mutex and B can proceed. But,
if thread B is suspended while waiting for the mutex, these
delays would be inserted when the thread wakes. Any delays
required while the thread is blocked could be inserted twice:
whether the thread was suspended or simply spun on the while it was blocked. Before executing a function that may waking thread. For simplicity, C

The thread can simply skip any delays that were incurred to delay num_threads

When a thread is unblocked by one of the listed functions, CoZ guarantees that all required delays have been inserted. The thread can simply skip any delays that were incurred while it was blocked. Before executing a function that may block on thread communication, a thread saves both the local and global delay counts. When the thread wakes, it sets its local delay count to the saved delay count, plus any global delays incurred since the call. This accounting is correct whether the thread was suspended or simply spun on the synchronization primitive. Table 1 lists the functions that require this additional handling.

Optimization: Minimizing Delays
If every thread executes the selected line, forcing each thread to delay num_threads – 1 times unnecessarily slow execution. If all but one thread executes the selected line, only that thread needs to pause. The invariant that must be preserved is the following: for each thread, the number of pauses plus the number of samples in the selected line must equal the global delay count. When a sample falls in the selected line, CoZ increments only the local delay count. If the local delay count is still less than the global delay count after processing all available samples, CoZ inserts pauses. If the local delay count is larger than global delay count, the thread increases the global delay count.

3.5 Progress Points
CoZ supports three different mechanisms for progress points: source-level, breakpoint, and sampled.

Source-Level Progress Points. Source-level progress points are the only progress points that require program modification. To indicate a source-level progress point, a developer simply inserts the CAUSAL_PROGRESS macro in the program’s source code at the appropriate location.

Breakpoint Progress Points. Breakpoint progress points are specified at the command line. CoZ uses the perf_event API to set a breakpoint at the first instruction in a line specified in the profiler arguments.

Sampled Progress Points. Like breakpoint progress points, sampled progress points are specified at the command line. However, unlike source-level and breakpoint progress points, sampled progress points do not keep a count of the number of visits to the progress point. Instead, sampled progress points count the number of samples that fall within the specified line. As with virtual speedups, the percent change in visits to a sampled progress point can be computed even when the raw counts are unknown.

Measuring Latency. Source-level and breakpoint progress points can also be used 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 \( \lambda \) as the arrival rate, we can solve for the average latency \( W \) via Little’s Law, which holds for nearly any queuing system: \( W = \frac{\lambda L}{\lambda L - \lambda} \) Rewriting Little’s Law, we then compute the average latency as \( L/\lambda \).

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 since the system will not be able to keep up with arrivals.

3.6 Adjusting for Phases
CoZ randomly selects a recently-executed line of code at the start of each performance experiment. This increases the likelihood that experiments will yield useful information—a virtual speedup would have no effect on lines that never run—but could bias results for programs with phases.
If a program runs in phases, optimizing a line will not have any effect on progress rate during periods when the line is not being run. However, COZ will not run performance experiments for the line during these periods because only currently-executing lines are selected. If left uncorrected, this bias would lead COZ to overstate the effect of optimizing lines that run in phases.

To eliminate this bias, we break the program’s execution into two logical phases: phase A, during which the selected line runs, and phase B, when it does not. These phases need not be contiguous. The total runtime $T = t_A + t_B$ is the sum of the durations of the two phases. The average progress rate during the entire execution is:

$$ P = \frac{T}{N} = \frac{t_A + t_B}{N}. \quad (3) $$

COZ collects samples during the entire execution, recording the number of samples in each line. We define $s$ to be the number of samples in the selected line, of which $s_{obs}$ occur during a performance experiment with duration $t_{obs}$. The expected number of samples during the experiment is:

$$ \mathbb{E}[s_{obs}] = s \cdot \frac{t_{obs}}{t_A}, \quad \text{therefore} \quad t_A \approx s \cdot \frac{t_{obs}}{s_{obs}}. \quad (4) $$

COZ measures the effect of a virtual speedup during phase A,

$$ \Delta p_A = \frac{p_A - p_A'}{p_A} $$

where $p_A'$ and $p_A$ are the average progress periods with and without a virtual speedup; this can be rewritten as:

$$ \Delta p_A = \frac{\frac{t_A}{n_A} - \frac{t_A'}{n_A}}{\frac{t_A}{n_A}} = \frac{t_A - t_A'}{t_A} \quad \text{where } n_A \text{ is the number of progress point visits during phase } A. $$

Using (4), the new value for $P$ with the virtual speedup is

$$ P' = \frac{t_A' + t_B}{N} $$

and the percent change in $P$ is

$$ \Delta P = \frac{P - P'}{P} = \frac{t_A + t_B}{N} \cdot \frac{s}{T} - \frac{t_A' + t_B}{N} \cdot \frac{s}{T} = \frac{t_A - t_A'}{T}. $$

Finally, using (4) and (5),

$$ \Delta P = \Delta p_A \cdot \frac{t_A}{T} \approx \Delta p_A \cdot \frac{t_{obs}}{s_{obs}} \cdot \frac{s}{T}. \quad (6) $$

COZ multiplies all measured speedups, $\Delta p_A$, by the correction factor $\frac{t_{obs}}{s_{obs}} \cdot \frac{s}{T}$ in its final report.

4. Evaluation

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

### Table 3: Summary of Optimization Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedup</th>
<th>Diff Size</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>2.56% ± 0.41%</td>
<td>-61, +4</td>
<td>342</td>
</tr>
<tr>
<td>dedup</td>
<td>8.95% ± 0.27%</td>
<td>-3, +3</td>
<td>2570</td>
</tr>
<tr>
<td>ferret</td>
<td>21.27% ± 0.17%</td>
<td>-4, +4</td>
<td>5937</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>37.5% ± 0.56%</td>
<td>-1, +0</td>
<td>1015</td>
</tr>
<tr>
<td>streamcluster</td>
<td>68.4% ± 1.12%</td>
<td>-1, +0</td>
<td>1779</td>
</tr>
<tr>
<td>swaptions</td>
<td>15.8% ± 1.10%</td>
<td>-10, +16</td>
<td>970</td>
</tr>
<tr>
<td>Memcached</td>
<td>9.39% ± 0.95%</td>
<td>-6, +2</td>
<td>10475</td>
</tr>
<tr>
<td>SQLite</td>
<td>25.60% ± 1.00%</td>
<td>-7, +7</td>
<td>92635</td>
</tr>
</tbody>
</table>

4.1 Experimental Setup

We perform all experiments on a 64 core, four socket AMD Opteron machine with 60GB of memory, running Linux 3.14 with no modifications. All applications are compiled using GCC version 4.9.1 at the -03 optimization level and debug information generated with -g. We disable frame pointer elimination with the -fno-omit-frame-pointer flag so the Linux can collect accurate call stacks with each sample. COZ is run with the default sampling period of 1ms, and a sample batch size of ten. Each performance experiment runs for a minimum of 100ms with a cooloff period of 10ms after each experiment. 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 3 summarizes the results of our optimization efforts. We describe our experience using COZ with each application below.

4.2.1 Case Study: blackscholes

The blackscholes benchmark, provided by Intel, solves the Black–Scholes differential equation to price a portfolio of stock options. We placed a progress point after each thread completes one round of the iterative approximation to the differential equation (blackscholes.c:259). COZ identifies many lines in the CNDf and BlkSchlsEqEurNoDiv functions that would have a small impact if optimized. This same code was identified as a bottleneck by ParaShares [27]; this is the only optimization we describe here that was previously reported. This block of code performs the main numerical work of the program, and uses many temporary
variables to break apart the complex computation. Manually eliminating common subexpressions and combining 61 piecewise calculations into 4 larger expressions resulted in a 2.56% ± 0.41% program speedup.

4.2.2 Case Study: dedup

The dedup application performs 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. This suggests that dedup’s shared hash table has a significant number of collisions. Increasing the hash table size had no effect on performance. This 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. This increased hash table utilization to 82.0% and resulted in an 8.95% ± 0.27% performance improvement. Figure 4 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. As with ferret, this result was achieved by one graduate student who was initially unfamiliar with the code; 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. As with ferret, 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%.

4.2.3 Case Study: ferret

The ferret benchmark performs a content-based image similarity search. Ferret consists of a pipeline with six stages: the first and the last stages are for input and output. The middle four stages perform image segmentation, feature extraction, indexing, and ranking. Ferret takes two arguments: an input file and a desired number of threads, which are divided equally across the four middle stages. We first inserted a progress point in the final stage of the image search pipeline to measure throughput (ferret-parallel.c:398). We then ran COZ with the --source-scope argument to limit our attention to the ferret-parallel.c file, rather than across the entire ferret toolkit.

Figure 5 shows the top three lines identified by COZ, using its default ranking metric. Lines 320 and 358 are calls to cass_table_query from the indexing and ranking stages. Line 255 is a call to image_segment in the segmentation stage. Figure 5 depicts ferret’s pipeline with the associated thread pools (colors indicate COZ’s computed impact on throughput of optimizing these stages).

Because each important line falls in a different pipeline stage, and because COZ did not find any important lines in the queues shared by adjacent stages, we can easily “optimize” a specific line by shifting threads to that stage. We modified

![Figure 4: In the dedup benchmark, COZ identified hash bucket traversal as a bottleneck. Collisions per-bucket for the first 4000 buckets before, midway through, and after optimization of the dedup benchmark (note different y-axes). The dashed horizontal line shows average collisions per-utilized bucket for each version. Replacing dedup’s hash function improved performance by 8%.](image-url)

![Figure 5: Ferret’s pipeline. The middle four stages each have an associated thread pool; the input and output stages each consist of one thread. The colors represent the impact on throughput of each stage, as identified by COZ: green is low impact, orange is medium impact, and red is high impact.](image-url)
ferret to let us specify the number of threads assigned to each stage separately, a four-line change.

Coz did not find any important lines in the feature extraction stage, so we shifted threads from this stage to the three other main stages. After three rounds of profiling and adjusting thread assignments, we arrived at a final thread allocation of 20, 1, 22, and 21 to segmentation, feature extraction, indexing, and ranking respectively. The reallocation of threads led to a $21.27\% \pm 0.17\%$ speedup over the original configuration, using the same number of threads.

**Comparison with gprof.** We also ran ferret with gprof in both the initial and final configurations. Optimization opportunities are not immediately obvious from that profile. For example, in the flat profile, the function `cass_table_query` appears near the bottom of the ranking, and is tied with 56 other functions for most cumulative time.

Gprof also offers little guidance for optimizing ferret. In fact, its output was virtually unchanged before and after our optimization, despite a large performance change.

### 4.2.4 Case Study: fluidanimate

The fluidanimate benchmark, also provided by Intel, is a physical simulation of an incompressible fluid for animation. The application spawns worker threads that execute in eight concurrent phases, separated 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.

Coz identifies a single modest potential speedup in the thread creation code, but there was no obvious way to speed up this code. However, Coz also identified two significant points of contention, indicated by a downward sloping causal profile. Figure [7] shows Coz’s output for these two lines. This result tells us that optimizing the indicated line of code would actually slow down the program, rather than speed it up. Both lines Coz identifies are in a custom barrier implementation, immediately before entering a loop that repeatedly calls `pthread_mutex_trylock`. Removing this spinning from the barrier would reduce the contention, but it was simpler to replace the custom barrier with the default `pthread_barrier` implementation. This one line change led to a $37.5\% \pm 0.56\%$ speedup.

#### 4.2.5 Case Study: streamcluster

The streamcluster benchmark performs online clustering of streaming data. As with fluidanimate, worker threads execute in concurrent phases separated by a custom barrier, where we placed a progress point. Coz identified a call to a random number generator as a potential line for optimization. Replacing this call with a lightweight random number generator had a modest effect on performance ($\sim 2\%$ speedup). As with fluidanimate, Coz highlighted the custom barrier implementation as a major source of contention. Replacing this barrier with the default `pthread_barrier` led to a $68.4\% \pm 1.12\%$ speedup.

#### 4.2.6 Case Study: swaptions

The swaptions benchmark is a Monte Carlo pricing algorithm for swaptions, a type of financial derivative. Like blackscholes and fluidanimate, this program was developed by Intel. We placed a progress point after each iteration of the worker threads’ main loop (`HJM_Securities.cpp:99`).

Coz identified three significant optimization opportunities, all inside nested loops over a large multidimensional array. One of these loops just zeroed out consecutive values, so we replaced all but the outermost loop with a call to `memset`. A second loop filled part of the same large array with values from a distribution function, with no obvious opportunities for optimization. The third nested loop iterated over the same array again, but traversed the dimensions in an irregular order. We reordered the loops to traverse dimensions from left to right whenever possible in order to improve the locality of the loop body. This change, along with the call to `memset`, sped execution by $15.8\% \pm 1.10\%$. 
4.2.7 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. Because reference counts are updated atomically, we can safely remove the lock from this function, which resulted in a 9.39% ± 0.95% speedup.

4.2.8 Case Study: SQLite

The SQLite database library is widely used by many applications to store relational data. The embedded 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 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.
throughput of line 320 by 27%. Coz predicted that this improvement would result in a 21.4% program speedup, which is nearly the same as the 21.2% we observe.

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 just 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 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%.

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 9 breaks down the total overhead of running Coz on each of the PARSEC benchmarks by category. The average overall overhead is 17%.

The primary contributor to Coz’s overhead is the introduction of delays for virtual speedup. This source of overhead can be reduced by performing fewer performance experiments during a program’s run, in exchange for increasing the execution time required to collect useful causal profiles.

The second greatest contributor to Coz’s overhead is sampling overhead: the cost of collecting samples, processing those samples, and producing profile output. The primary cost is due to initiating sampling with the perf API for every new thread. In addition, sampling is disabled during introduced delays, which requires two system calls (one before the delay, and one after).

Finally, startup overhead is due to Coz’s initial processing of debugging information for the profiled application. Because the benchmarks are sufficiently long running (mean: 103s) to amortize startup time, this effect is minimal.

Efficiency Summary. Coz’s profiling overhead is on average 17% (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., 6x 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 identifies and quantifies optimization opportunities, while most past work on profilers has focused on collecting detailed (though not necessarily actionable) information with low overhead.

5.1 General-Purpose Profilers

General-purpose profilers are typically implemented using instrumentation, sampling, or both. Systems based on sampling (including causal profiling) can arbitrarily reduce probe effect, although sampling must be unbiased.

The UNIX prof tool and oprofile both use sampling exclusively. Oprofile can sample using a variety of hardware performance counters, which can be used to identify cache-hostile code, poorly predicted branches, and other hardware bottlenecks. Gprof combines instrumentation and sampling to measure execution time. Gprof produces a call graph profile, which counts invocations of functions segregated by caller. Cho, Moseley, et al. reduce the overhead of Gprof’s call-graph profiling by interleaving instrumented and un-instrumented execution. Path profilers add further detail, counting executions of each path through a procedure, or across procedures.

5.2 Parallel Profilers

Past work on parallel profiling has focused on identifying the critical path or bottlenecks, although optimizing the critical path or removing the bottleneck may not significantly improve program performance.

Critical Path Profiling. IPS uses traces from message-passing programs to identify the critical path, and reports the amount of time each procedure contributes to the critical path. IPS-2 extends this approach with limited support for shared memory parallelism. Other critical path profilers rely on languages with first-class threads and synchronization to identify the critical path. Hollingsworth and Miller introduce two new metrics to approximate optimization potential: slack, how much a proce-
dure can be improved before the critical path changes; and *logical zeroing*, the reduction in critical path length when a procedure is completely removed [23]. These metrics are similar to the optimization potential measured by a causal profiler, but can only be computed with a complete program activity graph. Collection of a program activity graph is costly, and could introduce significant probe effect.

**Bottleneck Identification.** Several approaches have used hardware performance counters to identify hardware-level performance bottlenecks [9, 15, 33]. Techniques based on binary instrumentation can identify cache and heap performance issues, contended locks, and other program hotspots [5, 32, 37]. ParaShares and Harmony identify basic blocks that run during periods with little or no parallelism [26, 27]. Code identified by these tools is a good candidate for parallelization or classic serial optimizations. Bottlenecks, a profile analysis tool, uses heuristics to identify bottlenecks using call-tree profiles [3]. Given call-tree profiles for different executions, Bottlenecks can pinpoint which procedures are responsible for the difference in performance. The FreeLunch profiler and Visual Studio’s contention profiler identify locks that are responsible for significant thread blocking time [12, 13]. BIS uses similar techniques to identify highly contended critical sections on asymmetric multiprocessors, and automatically migrates performance-critical code to faster cores [25]. Bottleneck graphs present thread execution time and parallelism in a visual format that highlights program bottlenecks [14]. Unlike causal profiling, these tools do not predict the performance impact of removing bottlenecks. All of these systems can only identify bottlenecks that arise from explicit thread communication, while causal profiling can measure parallel performance problems from any source, including cache coherence protocols, scheduling dependencies, and I/O.

**Profiling for Parallelization and Scalability.** Several systems have been developed to measure potential parallelism in serial programs [17, 44, 36]. Like causal profiling, these systems identify code that will benefit from developer time. Unlike causal profiling, these tools are not aimed at diagnosing performance issues in code that has already been parallelized.

Kulkarni, Pai, and Schuff present general metrics for available parallelism and scalability [29]. The Cilkview scalability analyzer uses performance models for Cilk’s constrained parallelism to estimate the performance effect of adding additional hardware threads [21]. Causal profiling can detect performance problems that result from poor scaling on the current hardware platform.

**Time Attribution Profilers.** Time attribution profilers assign “blame” to concurrently executing code based on what other threads are doing. Quartz introduces the notion of “normalized processor time,” which assigns high cost to code that runs while a large fraction of other threads are blocked [4]. CPPOFJ extends this approach to Java programs with aspects [20]. CPPOFJ uses finer categories for time: running, blocked for a higher-priority thread, waiting on a monitor, and blocked on other events. Tallent and Mellor-Crummey extend this approach further to support Cilk programs, with an added category for time spent managing parallelism [42]. The WAIT tool adds fine-grained categorization to identify bottlenecks in large-scale production Java systems [1]. Unlike causal profiling, these profilers can only capture interference between threads that directly affects their scheduler state.

5.3 Performance Guidance and Experimentation

Several systems have employed delays to extract information about program execution times. Mytkowicz et al. use inserted delays to validate the output of profilers on single-threaded Java programs [36]. Snelick, Jája et al. use delays to profile parallel programs [39]. This approach measures the impact of slowdowns in combination, which is impractical because it requires a complete execution of the program for each of an exponential number of configurations. Active Dependence Discovery (ADD) introduces performance perturbations to distributed systems and measures their impact on response time [8]. ADD requires a complete enumeration of system components, and requires developers to insert performance perturbations manually. Song and Lu use machine learning to identify performance anti-patterns in source code [40]. None of these approaches quantify the effect of potential optimizations, which causal profiling measures directly.

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.

Acknowledgments

Omitted for double-blind reviewing. This material is based upon work supported by the National Science Foundation under Grants No. CCF-1012195 and CCF-1439008. Charlie Curtisinger was supported by a Google PhD Research Fellowship. The authors thank Dan Barowy, Emma Tosch, and John Vilk for their feedback and helpful comments.
References


