Coz: Causal Profiling

Charlie Curtsinger Emery D. Berger

School of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

Abstract

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 performs a series of 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. Causal profiling can profile both throughput and latency via programmer-supplied progress points like transaction boundaries. We present Coz, a prototype causal profiler, and empirically demonstrate its efficiency (mean slowdown: 17%) and its effectiveness. As a case study, we use Coz to guide the optimization of two applications from the PARSEC suite, achieving speedups of 8% and 20%.

1. Introduction

With the end of Dennard scaling, performance is once again a first-class concern for software developers. While compiler optimizations are of some assistance, they often do not make enough of an impact on performance to meet programmers' demands [16]. The result is that programmers seeking to increase the throughput or responsiveness of their applications generally must resort to manual performance tuning.

Manually inspecting an entire program to identify optimization opportunities is impractical. Instead, developers use profilers to focus their tuning efforts on code responsible for a significant fraction of execution time. Prominent examples include `oprofile`, `perf`, and `gprof` [2, 25, 34].

Unfortunately, even when a profiler accurately reports where a program is spending the bulk of its time [41], this information can lead programmers astray. Where programs spend their time is not necessarily correlated with where programmers should focus their optimization effort. This phenomenon is especially notable in interactive applications and servers, which spend much of their time waiting for I/O, and for multithreaded code running on multicore systems.

Figure 1 illustrates the shortcomings of existing profilers with an example program. It spawns two threads that respectively invoke functions `a` and `b`. Most profilers will report that these functions each comprise roughly half of overall execution time; some profilers will additionally report that `a` is on the critical path [51].

This information is accurate but potentially misleading. On a multicore system, optimizing `a` entirely away—thus eliminating the critical path entirely—would only speed the program up by 4.5%, as `b` would become the critical path.

The heart of the problem is a mismatch between the question that current profilers answer—*where does the program spend its time?*—and the question programmers want the answer to: *where should I focus my optimization efforts?*

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 2 shows the results of running our prototype causal profiler. This profile plots the hypothetical speedup of a line of code (x-axis) versus its impact on exe-

```
1  // A multithreaded C++ program that illustrates
2  // the shortcomings of standard profilers.
3  volatile size_t x, y;
4
5  void a() { // ~6.7 seconds
6    for(x=0; x<2000000000; x++) {}
7  }
8
9  void b() { // ~6.4 seconds
10   for(y=0; y<1900000000; y++) {}
11 }
12 int main() {
13   // Spawn both threads and wait for them.
14   thread a_thread(a), b_thread(b);
15   a_thread.join(); b_thread.join();
16 }
```

```
%    cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call   name
55.20     7.20      7.20                               a()
45.19    13.09      5.89                               b()
 0.00    13.09      0.00        2     0.00     0.00
```

Figure 1. An example multithreaded program and its representative profile (from `gprof`). Standard profilers report and attribute execution time, but these do not necessarily correlate with where programmers should focus their optimization effort. On a multicore system, independently optimizing either `a` or `b` would have little to no impact on performance.

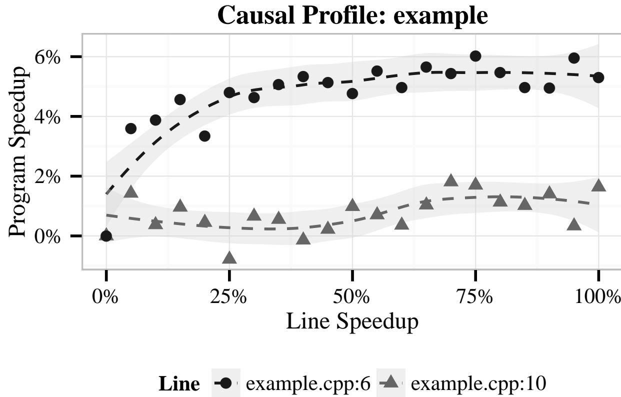

Figure 2. A causal profile for the program in Figure 1 generated by Coz. The x-axis shows the percentage of speedup in the program that would be achieved by speeding up the given line of code by the percentage given in the y-axis (the gray area corresponds to standard error). Unlike past profilers, causal profiling accurately predicts the impact of optimizations on performance.

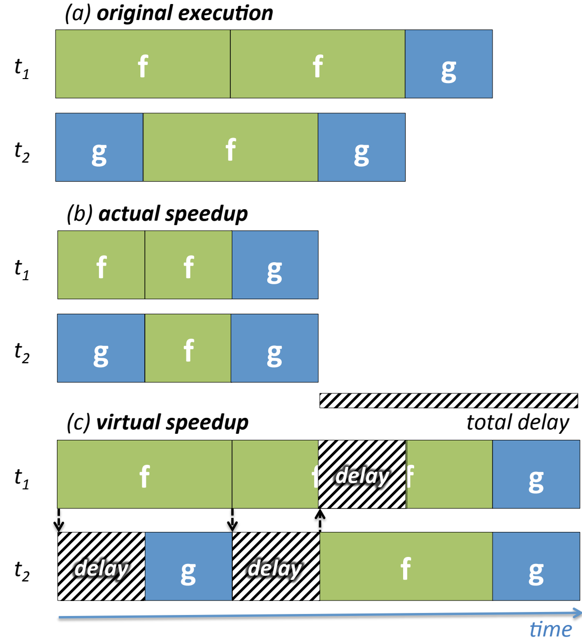

Figure 3. An illustration of virtual speedup: (a) shows the original execution of two threads, running functions `f` and `g`; (b) shows the effect of *actually* halving the runtime of `f`; (c) shows the effect of *virtual speedup*, inserting delays in other threads whenever `f` runs (denoted by the arrows). Both actual and virtual speedups cause the threads to end at the same time; by subtracting delays, the elapsed times also match.

cution time (y-axis). The graph shows that optimizing `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, 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 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). More importantly, we demonstrate that causal profiling accurately predicts optimization opportunities, and that it is effective at guiding optimization efforts. We perform a case study using Coz to optimize two applications from the PARSEC benchmark suite: `dedup` and `ferret`. Using traditional profilers provides little direction, but with Coz's guidance, over the course of three hours we were able to increase the performance of these applications by 8% and 20%, respectively.

Contributions

The contributions of this paper are the following:

1. It presents **causal profiling**, which departs from past profilers by conducting on-line experiments. Using *virtual speedups* and *progress points*, causal profiling quantifies

the impact of potential optimizations on both throughput and latency (§2).

2. It presents **COZ**, a prototype implementation of causal profiling. It describes its implementation (§3), and demonstrates its efficiency and effectiveness at locating and quantifying optimization opportunities (§4).

2. Causal Profiling Overview

This section provides an overview of how causal profiling works in the context of COZ, our prototype causal profiler.

Progress Points

With COZ, developers use *progress points* to guide profiling: these let them measure the impact of any proposed optimization on either throughput or latency, rather than just end-to-end execution time. Progress points are source locations whose rate of progress developers want to optimize. A single progress point indicates throughput, while two progress points indicates the latency between events. Programmers can specify progress points at the command line (with `--progress` followed by the source file and line number) or directly in the source code via macros (§ 3.4).

The developer invokes COZ using the following command: `coz --scope program.c --- program <args>` (indicating progress points as desired). Arguments before the separator are passed to the profiler. The optional `--scope` flag and arguments tell COZ to only evaluate potential optimizations in a file or files, rather than across all libraries.

Performance Experiments and Virtual Speedup

After a brief initialization phase (§ 3.1), COZ continuously performs a number of performance experiments during program execution (§ 3.2). During each experiment, COZ randomly selects a line of code to virtually speed up. Once a line has been selected for speedup, COZ decides how much speedup to evaluate in the current performance experiment. The amount of virtual speedup applied to the selected line is chosen randomly between 0% (no change) and 100% (the line is completely eliminated).

Virtual speedup creates the *effect* of optimizing the selected line by the selected degree. The key insight is that it is possible to virtually speedup a line by pausing all other threads whenever that line is executed in one thread. Threads that execute that line then run faster relative to threads that do not. (While this approach appears simple, there are numerous intricacies involved: see §3.3.)

COZ then logs the state of all progress points and starts the performance experiment. It applies the selected virtual speedup for the duration of the experiment, after which it again logs the progress point state. The performance effect of each virtual speedup is computed later in an offline pass. Once an experiment ends, COZ selects a new line and speedup to conduct another performance experiment.

Ranking Optimization Opportunities

After all causal profiles have been collected (either from one or many runs), the developer invokes `coz-process` to do required post-processing, which generates a `.csv` file. This file consists of measured program speedups (that is, the speedup of the progress points) for varying line speedup amounts across many lines in the program. Developers can use the program `coz-plot` to plot the line speedup vs. program speedup on a *speedup curve*.

These curves show two important properties for each line of code. First, the slope of the speedup curve shows the *payback rate* for optimizing this line: the steeper the line, the greater impact a small optimization will have. Second, the maximum value on the speedup curve shows the *maximum impact*: the largest possible performance improvement from optimizing this line of code. Lines with high values for both payback rate and maximum impact offer the greatest optimization opportunity, and should be ranked accordingly.

COZ performs a linear regression to obtain a measure of a line's overall optimization potential. Lines of code are ranked by the slope of this linear regression line. Lines with both a high payback rate and maximum impact will have a steeper regression line. Developers may also wish to bias their search toward payback rate or maximum impact. Weighting points with low line speedup values (toward the left of the plot) biases the regression toward payback rate, while weighting points with large line speedups emphasizes maximum impact.

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, which supports a wide range of hardware performance counters and sample types.

3.1 COZ Initialization

A user invokes COZ using a command of the form `coz <profiler args> --- <program> <args>`. `coz` immediately runs the specified program with `exec`, passing in all arguments (including profiler options), and sets the `LD_PRELOAD` environment variable to load the main profiler runtime. This approach lets COZ interpose on library calls from the program, including the `libc_start_main` function, which runs before `main`.

Locating Binaries and Sources. COZ intercepts the call to `libc_start_main` at startup. Before running the program's `main` function, COZ walks the list of loaded executables and shared libraries using the `dl_iterate_phdr` function. Users can specify a *profiling scope* in the profiler arguments. By default, the scope includes all source files from the main

executable, but alternate source locations and libraries can be specified. COZ records the loaded address and path to each in-scope executable for later processing.

Building the Source Map. COZ uses DWARF debugging information to map program addresses to source locations [19]. Each in-scope executable is opened and checked for debugging information. If the loaded executable has been stripped of debugging information, COZ uses the same procedure as `gdb` to search standard system paths for separate debugging information [22]. Note that most Linux distributions offer packages that include debug symbols for common libraries. COZ uses the collected debug information to construct a map from address ranges to source locations.

The DWARF format includes both caller and callee information for inlined procedures. Special handling is required when an in-scope callsite 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 mirrors the process by which COZ attributes out-of-scope samples to callsites during execution (see the discussion of sample attribution, below).

Enabling Sampling. Just before calling the program's real `main` function, COZ opens a `perf_event` file to begin sampling in the main thread. The `perf_event_open` system call takes in a configuration that specifies which hardware or software event to count (such as CPU cycles, page faults, or cache misses), the number of events between samples, and options for sample collection. The `perf_event_open` system call returns a file descriptor that can be read to access event counts, or a memory-mapped file to access samples directly from a ring buffer. 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 Performance Experiments

Initiating Performance Experiments. A single profiler thread created at startup manages all performance experiments. Each performance experiment starts with the selection of a source line for virtual speedup. The profiler thread spins until the `next_line` atomic pointer is set to a valid line. Whenever this pointer is null, threads will attempt to set it to the line containing their most recent sample.

Once the profiler receives a valid line from one of the program's threads, it chooses a random speedup amount and logs the start of the experiment (including the current time, chosen line, the running count of samples in the selected line, and all progress point counter values). Speedup amounts are drawn uniformly between zero and 100%. The null virtual speedup measurement serves as a baseline for comparison with non-zero speedups—these experiments include the cost of inserting a virtual speedup without the effect of any actual speedup.

Running Performance Experiments. Once a performance experiment has started, each of the program's threads processes samples and inserts delays to perform virtual speedups. The profiler thread periodically checks if progress counters have changed, and if enough delays have been inserted. By default, an experiment cannot end until 100ms have elapsed, all progress points have been visited at least five times, and at least five delays have been inserted for virtual speedups. If the experiment has not finished after 500ms, the minimum delay condition is dropped. This cutoff ensures that an experiment will not run indefinitely if a rarely executed line is selected.

Finally, 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 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.

3.3 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, distorting the program's execution. Instead, COZ uses sampling to implement virtual speedups accurately and efficiently. Delays are in proportion to the time spent in the selected line. This allows COZ to virtually speed up the line by a specific percent, even though the number of visits to the line is unknown.

COZ periodically samples the program counter in each thread and maps each sample to a source line using DWARF debug information. When one thread receives a sample in the selected line, all other threads must pause. COZ triggers these pauses using two counters: a shared global delay count, and a local delay count that is private to each thread. When a thread's local count is less than the global count, the thread

must pause. To force other threads to pause, a thread simply increments both the global counter and its own local count. COZ checks the counters and adds any required delays immediately after processing samples.

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

$$\mathbb{E}[s] = \frac{n \cdot t}{P} \tag{1}$$

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

$$t' = \frac{n \cdot t \cdot (1 - \frac{t}{P}) + \frac{n \cdot t}{P} \cdot (t - d)}{n} = t \cdot (1 - \frac{d}{P}) \tag{2}$$

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%.

Ensuring Accurate Timing. COZ uses the `nanosleep` POSIX function to insert delays. This function only guarantees 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. COZ interposes on the `pthread_create` function to start sampling and adjust delays. 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 interrupts 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.

Potentially *unblocking* calls

`pthread_mutex_unlock`	unlock a mutex
`pthread_cond_signal`	wake one waiter on condition var.
`pthread_cond_broadcast`	wake all waiters on condition var.
`pthread_barrier_wait`	wait at a barrier
`pthread_kill`	send signal to a thread
`pthread_exit`	terminate this thread

Table 1. COZ intercepts POSIX functions that could wake a blocked thread. To ensure correctness of virtual speedups, COZ forces threads to execute any unconsumed delays before invoking any of these functions and potentially waking another thread.

Potentially *blocking* calls

`pthread_mutex_lock`	lock a mutex
`pthread_cond_wait`	await signal on condition variable
`pthread_barrier_wait`	wait at a barrier
`pthread_join`	wait for a thread to complete
`sigwait`	wait for a signal
`sigwaitinfo`	*ibid*
`sigtimedwait`	*ibid*
`sigsuspend`	*ibid*

Table 2. COZ intercepts POSIX functions that could block waiting for a thread, instrumenting them to update delay counts before and after blocking.

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: once by thread A before unlocking the mutex, and then again in thread B after acquiring the mutex.

To correct this behavior, blocked threads must inherit the delay count from the thread that unblocks them. This causal propagation ensures that any delays inserted before unblocking the thread would not be inserted again in the waking thread. For simplicity, COZ forces threads to execute all required delays before performing an operation that could wake a blocked thread. These operations include the POSIX calls given in Table 1.

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 2 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 slows 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.

3.4 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 λ as the arrival rate, we can solve for the average latency W via Little's Law, which holds for nearly any queuing system: $L = \lambda W$ [35]. 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 since the system will not be able to keep up with arrivals.

3.5 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 if the line does not run—but could bias results for programs with phases. This section derives a correction for this bias that enables COZ to accurately profile 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

$$P = \frac{T}{N} = \frac{t_A + t_B}{N}. \tag{3}$$

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:

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

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

$$\Delta p_A = \frac{\frac{t_A}{n_A} - \frac{t_A{}'}{n_A}}{\frac{t_A}{n_A}} = \frac{t_A - t_A{}'}{t_A} \tag{5}$$

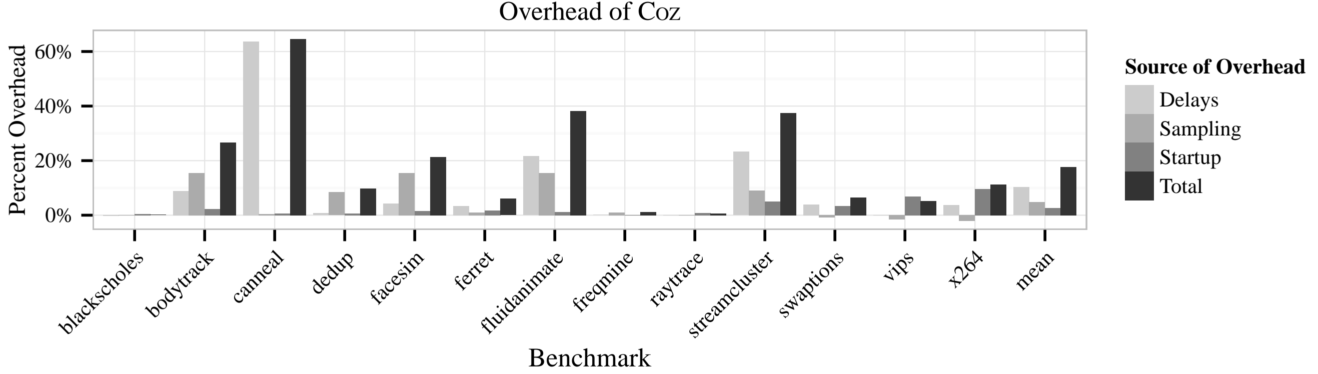

Figure 4. Percent overhead for each of COZ's possible sources of overhead. *Delays* are the overhead due to adding delays for virtual speedups, *Sampling* is the cost of collecting and processing samples, and *Startup* is the initial cost of processing debugging information.

where n_A is the number of progress point visits during phase A. Using (3), the new value for P with the virtual speedup is

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

and the percent change in P is

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

Finally, using (4) and (5),

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

COZ multiplies all measured speedups, Δp_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:

- Is COZ's overhead low enough to be practical?
- Does causal profiling enable effective performance tuning?

4.1 Methodology

We perform all experiments on a 64 core, four socket AMD Opteron machine with 60GB of memory, running Linux 3.13 with no modifications. All benchmarks are compiled using GCC version 4.8.2 at the `-O3` optimization level, with debug output enabled. We disable frame pointer elimination with the `-fno-omit-frame-pointer` so that `perf` can collect accurate call stacks with each sample.

4.2 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 4 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 source of overhead is minimal.

4.2.1 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., 6× for `ferret`, versus 6% with COZ), these results confirm that COZ has sufficiently low overhead to be used in practice.

4.3 Effectiveness

We perform two case studies to evaluate the effectiveness of using COZ to guide optimizations. We use two applications from the PARSEC benchmark suite for this task: `ferret` and `dedup`. Our initial choice of `ferret` was arbitrary; we next chose `dedup` because of its superficial similarity to `ferret` (both use pipeline-based parallelism).

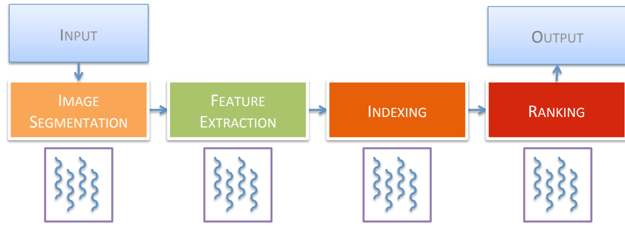

Figure 5. `ferret`'s pipeline. The middle four stages each have an associated thread pool; the input and output stages each consist of one thread. The colors represent the impact on throughput of each stage, as identified by COZ: green is low impact, orange is medium impact, and red is high impact.

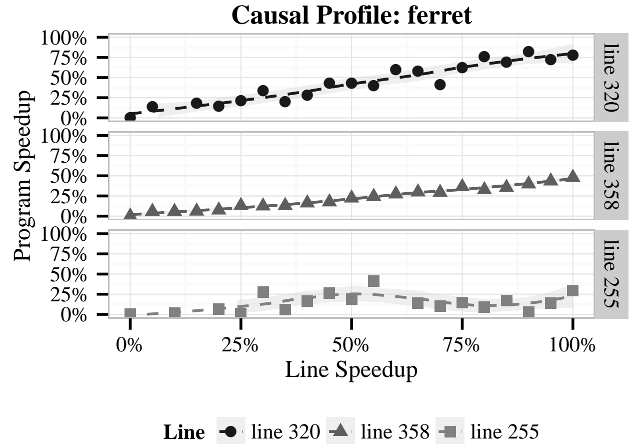

Figure 6. COZ output for the unmodified `ferret` application. The x-axis shows the amount of virtual speedup applied to each line, versus the resulting change in throughput on the y-axis. The top two lines are executed by the indexing and ranking stages; the third line is executed during image segmentation.

4.3.1 Case Study: `ferret`

> [The Ferret benchmark] is based on the Ferret toolkit which is used for content-based similarity search of feature-rich data. Ferret is parallelized using the pipeline model with six stages. The first and the last stage are for input and output. The middle four stages are for query image segmentation, feature extraction, indexing ... and ranking. Each stage has its own thread pool and the basic work unit of the pipeline is a query image. [11]

`ferret` takes two arguments: an input file and a desired number of threads, which are divided equally across the four middle stages. On a 64-core AMD Opteron system, running the unmodified `ferret` application with 16 threads per stage takes 34.5 seconds on the largest available input.

Profiling

We first inserted a call to the `CAUSAL_PROGRESS` macro in the final stage of the image search pipeline to measure throughput. We then ran COZ with the `--scope` argument to limit our attention to the `ferret-parallel.c` file, rather than across the entire Ferret toolkit and associated libraries.

Figure 6 shows the top three lines identified by COZ, using its default ranking metric. Lines 320 and 358 are calls to the `cass_table_query` function from the indexing and ranking stages. Line 255 is a call to `image_segment` in the segmentation stage of the pipeline. Figure 5 depicts `ferret`'s pipeline with the associated thread pools (colors indicate COZ's computed impact on throughput of optimizing these stages).

Iterative Optimization

The fact that `ferret` uses pipeline parallelism leads to the following optimization strategy: First, use COZ to identify optimization opportunities. For each of these, check to see what stage they belong to. Finally, optimize any stage by allocating more threads to it. We modified `ferret` to let us specify the number of threads allocated per stage (changing 4 lines of code).

Round 1. With this modification in hand, we were able to increase the throughput of the three identified lines simply by allocating additional threads to these stages. Since COZ did not find any optimization opportunities in the feature extraction stage, we took nine threads from that stage and evenly divided them across the other three stages (for each stage: 19, 7, 19, 19). With this reallocation, `ferret` ran in just 30 seconds, a 13% speedup.

Round 2. We re-ran COZ to see if there were any further optimization opportunities. Lines 320 and 358 remained as good targets for optimization, so we moved an additional two threads from the feature extraction stage (for each stage: 19, 5, 20, 20). This reassignment resulted in a runtime of 28.7 seconds, 17% faster than the original configuration.

Round 3. Finally, we performed one last round of causal profiling. This profile revealed that the segmentation stage continued to have a potential optimization impact. This fact suggested that the now-optimized indexing and ranking stages occasionally were blocked waiting for inputs to arrive from the segmentation stage. To balance the pipeline, we took all but one of the five remaining feature extraction threads and added two to indexing, and one each to segmentation and ranking (for each stage: 20, 1, 22, 21).

The resulting final runtime was 27.5 seconds, a 20% improvement over the default configuration using the same number of threads. The entire tuning process described above took approximately one hour from start to finish. It was performed by a single graduate student with no previous familiarity with the application itself.

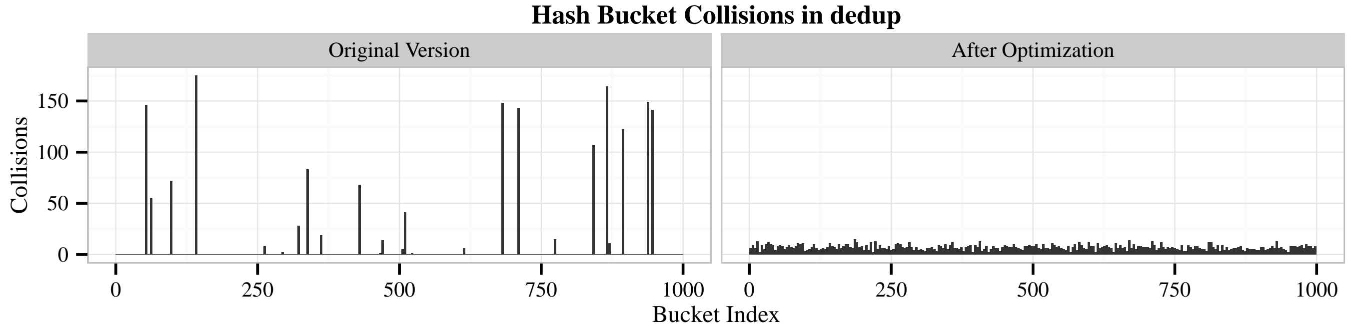

Figure 7. The number of insertions into the first 1000 hash buckets before and after using COZ to tune the dedup benchmark. COZ identified hash bucket traversal as a bottleneck. Fixing dedup's hash function and reducing the number of buckets yielded an 8% performance improvement.

Comparison with gprof

As a point of comparison, 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, tied with 56 other functions for most cumulative time. gprof also offers little guidance in iteratively optimizing a program: its output was virtually unchanged before and after optimizing ferret, despite a 20% change in performance.

4.3.2 Case Study: dedup

Our second case study is on dedup, an application that performs parallel file compression with deduplication. Like ferret, dedup uses pipeline parallelism. The pipeline is divided into three main stages: fine-grained fragmentation, hash computation, and compression. As with ferret, we placed a progress point in dedup's final output stage. Unlike ferret, dedup's work queue structure makes it nontrivial to adjust the allocation of threads to stages:

> In order to avoid lock contention, the number of queues is scaled with the number of threads, with a small group of threads sharing an input and output queue at a time. [11]

While dedup assigns threads to separate work queues to reduce contention, all threads compete for access to a shared hash table. COZ's output for dedup identified three lines of interest, all within dedup's hash computation stage. These lines all fall within a single critical section that searches a hash bucket's elements (by calling hashtable_search) in the hash computation stage.

Increasing the hash table size had no effect on performance, which suggested that the bucket contention identified by COZ was not caused by the number of buckets. We discovered that dedup's hash function is degenerate, mapping keys to only a tiny fraction of available buckets (only 2.3%). The peak number of items in a single bucket was 397.

The original hash function adds characters of the key, which leads to virtually no high order bits being set. We replaced this hash function with one that computes the bitwise XOR of 32 bit chunks of the key, which spreads the keys more evenly. We then halved the size of the hash table to reduce cache footprint. Figure 7 shows the distribution of bucket insertions before and after our fix, which increases utilization to 82% (peak items in a bucket: 11).

We ran COZ again, which produced a *negative* speedup curve for one line of the compression stage. A negative speedup curve means that optimizing the identified line would slow the program down, which is a sign of contention. We could address this by shifting threads away from this stage, but as our goal was to quickly identify performance opportunities, we decided that the restructuring required to support this change would be too intrusive.

After the changes to the hash function described above, end-to-end runtime increased by 6.5%; halving the number of buckets further improved performance, yielding an overall speedup of 8.0%. As with ferret, this result was achieved by one graduate student who was initially unfamiliar with the code; the total tuning effort took two hours.

Comparison with gprof

Again, we ran both the original and final 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. In the final profile, hashtable_search's share of execution time reduced to 1.1%.

4.3.3 Effectiveness Summary

Our case studies confirm that COZ is effective at pointing out optimization opportunities and guiding performance tuning. In both cases, the information that COZ provided was precise, pointing to specific lines, and accurate: optimizing the lines, either by adding more threads to the corresponding

stages or by resolving a problem with a hash function, significantly sped up programs (by 8% and 20%). By contrast, the information provided by a standard profiler proved to be of little assistance.

5. Related Work

Profilers have been an active area of research since the introduction of the `prof` tool [49].

5.1 General-Purpose Profilers

General-purpose profilers are typically implemented using instrumentation, sampling, or both. Instrumentation is either inserted at compile-time or through binary rewriting, and can be used to track execution counts for regions of code. Sampling profilers sample the program counter periodically to determine the fraction of time spent in each region of code. One key benefit of sampling is improved accuracy through the reduction of *probe effect*, the distortion of a program's execution due to profiler overhead. Systems based on sampling (including COZ) can arbitrarily reduce probe effect, although care must be taken to ensure that sampling is unbiased [41].

The UNIX `prof` tool and `oprofile` both use sampling exclusively [34, 49]. `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. Binary instrumentation toolkits enable detailed performance measurement of unmodified binaries. Valgrind includes two performance measurement tools, Cachegrind and Massif, that respectively measure cache and heap performance [42]. Pin is used as the basis for the Intel Parallel Amplifier, which identifies contended locks, bottlenecks, and hotspots [9, 36].

`gprof` combines instrumentation and sampling to measure execution time [25]. `gprof` produces a call graph profile, which counts invocations of functions segregated by caller. Cho et al. reduce the overhead of `gprof`'s call-graph profiling by interleaving instrumented and un-instrumented execution [14]. Path profilers add further detail, counting executions of each path through a procedure, or across procedures [5, 10]. Unlike COZ, these profilers do not identify code that will impact performance. As shown in §1 and §4.3, Optimizing functions or paths that make up a large fraction of a program's runtime is not guaranteed to improve performance.

5.2 Parallel Profilers

Parallel profilers offer specific support to aid in performance tuning parallel programs.

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 [7]. CPPROFJ extends this approach to Java programs with aspects [26]. CPPROFJ uses finer categories for time: running, blocked for a higher-priority thread, waiting on a monitor, and blocked on other events. Tallent et al. extend this approach further to support Cilk programs, with an added category for time spent managing parallelism [48]. The WAIT tool adds fine-grained categorization for enterprise Java libraries to identify bottlenecks in large-scale production Java systems [4].

Time attribution profilers use the state of other threads as a heuristic for dependence. Code that executes while other threads are blocked does not necessarily fall on the program's critical path. These profilers may thus overstate the importance of code, leading to wasted developer time. Unlike COZ, these profilers do not capture interference between threads that does not affect threads' scheduler state, such as cache coherence overhead.

Tracing Profilers. Tracing profilers intercept interactions between threads or nodes in a distributed system to construct a model of parallel performance. Monit and the Berkeley UNIX Distributed Programs Monitor collect traces of system-level events (e.g., process creation), inter-process communication, and system variables for later analysis [32, 39]. Aguilera et al. use network-level tracing to identify probable "causal paths" in distributed systems of black boxes, and identify paths that may be responsible for high latency [3]. AppInsight uses a similar technique to identify sources of latency in mobile application event handlers [44].

Unlike COZ, tracing tools must be tailored to each mechanism for thread interaction. This approach is only possible in domains where thread interaction is limited. Programs written with `pthreads` may communicate with synchronization operations, but interaction can also arise due to cache coherence protocols, race conditions, scheduling dependencies, or lock-free algorithms. In general, it is not possible for a trace to capture all instances of thread interaction without imposing prohibitively high overhead.

Critical Path Profiling. Detailed program traces can be used to construct a program activity graph, which can be used to identify the critical path. Yang and Miller developed IPS, which uses traces from message-passing programs to identify the critical path [40, 52]. They then report time spent in each procedure on the critical path. IPS-2 extends this approach with limited support for shared memory parallelism [38, 53]. Other critical path profilers rely on languages with first-class threads and synchronization to identify the critical path [28, 43, 46]. Critical path profiling relies on an accurate trace of thread interactions to construct the program activity graph. COZ measures peak and marginal impact, which correspond to measurements of the critical path, without interposing on thread interactions.

Using the program activity graph, Hollingsworth and Miller introduce two new metrics for optimization potential: *slack*, how much a procedure can be improved before

the critical path changes; and *logical zeroing*, the reduction in critical path length when a procedure is completely removed [29, 30]. These metrics are similar to peak impact measured by COZ, but can only be computed with a complete program activity graph. Collection of a program activity graph is costly, and would likely introduce significant probe effect.

Critical path profiling is not limited to software. Fields et al. apply critical path analysis to profiling microarchitectural designs, and X-Trace applies critical path analysis to networked systems [20, 21]. These domains are well suited to critical path analysis because interactions can be monitored with relatively little overhead.

Bottleneck Identification. PerfExpert, Paradyn, and work by Diamond et al. use hardware performance counters to identify hardware-level performance bottlenecks [13, 17, 37]. Bottlenecks, a profile analysis tool, uses heuristics to identify bottlenecks using call-tree profiles [6]. Given call-tree profiles for different executions, Bottlenecks can pinpoint which procedures are responsible for the difference in performance. Unlike COZ, these tools do not predict the effect of removing bottlenecks, which may not impact the critical path.

Visual Studio's contention profiler identifies locks that are responsible for significant thread blocking time [24]. BIS uses similar techniques to identify highly-contended critical sections on asymmetric multiprocessors, and automatically migrates performance-critical code to faster cores [31]. Bottle graphs present thread execution time and parallelism in a visual format that highlights program bottlenecks [18]. All three systems can only identify bottlenecks that arise from explicit thread communication, while COZ can measure parallel performance problems from any source.

Profiling for Parallelization and Scalability. Several systems have been developed to measure potential parallelism in serial programs [23, 50, 55]. Like COZ, these systems identify code that will benefit from developer time. Unlike COZ, these tools are not aimed at diagnosing performance issues in code that has already been parallelized.

Kulkarni et al. present general metrics for architecture-independent available parallelism and scalability [33]. The Cilkview scalability analyzer uses performance models for Cilk's constrained parallelism to estimate the performance effect of adding additional threads [27]. Fixing scalability issues identified by these approaches will not necessarily improve performance. COZ will detect performance problems that result from poor scaling, but only at the current level of hardware parallelism.

Other systems measure the effect of load and input size on performance [15, 47, 54]. Developers must generate a variety of input sizes for these tools, whereas COZ automatically generates diversity when conducting performance experiments.

Performance Experimentation. Snelick et al. introduce delays, which they call synthetic perturbations, to profile parallel programs [45]. This approach, similar to COZ's slowdown experiments, measures only what we call payback rate. Synthetic perturbation measures the impact of slowdowns in combination: this technique requires a complete execution of the program for each of an exponential number of configurations, making it impractical. Active Dependence Discovery (ADD) introduces performance perturbations to distributed systems and measures the impact on response time [12]. ADD requires a complete enumeration of system components, and requires developers to insert performance perturbations manually. Neither approach considers bounds on optimizations' impact due to sub-critical path length, which COZ measures during its performance experiments.

X-Ray is a tool for "differential performance evaluation." Given two inputs—one typical and a second that results in poor performance—X-Ray will find where executions on these inputs diverge and quantifies the cost of the divergence [8]. Unlike COZ, X-Ray does not evaluate individual changes in isolation, and can only report that an operation may be correlated with poor performance.

Mytkowicz et al. also use inserted delays to validate the output of profilers on single-threaded Java programs [41]. They identify hot methods using a profiler, and add delays to these methods. If the profiler output is accurate, added delays should increase the significance of the hot methods.

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 lines of code or functions with execution time or performance counters. This information can be of limited use for concurrent applications because the amount of time spent does not necessarily correlate with 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 represents 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 programmers in their optimization efforts.

COZ will be available for download at `coz-tool.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, Emma Tosch, and John Vilk for their feedback and helpful comments.

References

[1] *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, 2012.

[2] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org, July 2014.

[3] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89. ACM, 2003.

[4] Erik R. Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *OOPSLA*, pages 739–753. ACM, 2010.

[5] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, pages 85–96. ACM, 1997.

[6] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 170–194. Springer, 2004.

[7] Thomas E. Anderson and Edward D. Lazowska. Quartz: A tool for tuning parallel program performance. In *SIGMETRICS*, pages 115–125, 1990.

[8] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In Chandu Thekkath and Amin Vahdat, editors, *OSDI*, pages 307–320. USENIX Association, 2012.

[9] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with Pin. *Computer*, 43(3):34–41, March 2010.

[10] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO*, pages 46–57, 1996.

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In Andreas Moshovos, David Tarditi, and Kunle Olukotun, editors, *PACT*, pages 72–81. ACM, 2008.

[12] Aaron B. Brown, Gautam Kar, and Alexander Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Integrated Network Management*, pages 377–390. IEEE, 2001.

[13] Martin Burtscher, Byoung-Do Kim, Jeffrey R. Diamond, John D. McCalpin, Lars Koesterke, and James C. Browne. PerfExpert: An easy-to-use performance diagnosis tool for hpc applications. In *SC*, pages 1–11. IEEE, 2010.

[14] Hyoun Kyu Cho, Tipp Moseley, Richard E. Hank, Derek Bruening, and Scott A. Mahlke. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *CGO*, pages 1–10. IEEE Computer Society, 2013.

[15] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *PLDI* [1], pages 89–98.

[16] Charlie Curtsinger and Emery D. Berger. STABILIZER: Statistically sound performance evaluation. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, New York, NY, USA, 2013. ACM.

[17] Jeffrey R. Diamond, Martin Burtscher, John D. McCalpin, Byoung-Do Kim, Stephen W. Keckler, and James C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *ISPASS*, pages 32–43. IEEE Computer Society, 2011.

[18] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 355–372, New York, NY, USA, 2013. ACM.

[19] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format, Version 4*, 2010.

[20] Brian A. Fields, Rastislav Bodík, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *MICRO*, pages 228–242. ACM/IEEE, 2003.

[21] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*. USENIX, 2007.

[22] Free Software Foundation. *Debugging with GDB*, tenth edition.

[23] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI*, pages 458–469. ACM, 2011.

[24] Maxim Goldin. Thread performance: Resource contention concurrency profiling in visual studio 2010. *MSDN magazine*, page 38, 2010.

[25] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

[26] Robert J. Hall. CPPROFJ: Aspect-Capable Call Path Profiling of Multi-Threaded Java Applications. In *ASE*, pages 107–116. IEEE Computer Society, 2002.

[27] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156. ACM, 2010.

[28] Jonathan M. D. Hill, Stephen A. Jarvis, Constantinos J. Siniolakis, and Vasil P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *PDP*, pages 286–294, 1998.

[29] Jeffrey K. Hollingsworth and Barton P. Miller. Parallel program performance metrics: A comparison and validation. In *SC*, pages 4–13. IEEE Computer Society / ACM, 1992.

[30] Jeffrey K Hollingsworth and Barton P Miller. Slack: a new performance metric for parallel programs. *University of Maryland and University of Wisconsin-Madison, Tech. Rep*, 1994.

[31] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, pages 223–234. ACM, 2012.

[32] Teemu Kerola and Herbert D. Schwetman. Monit: A performance monitoring tool for parallel and pseudo-parallel programs. In *SIGMETRICS*, pages 163–174, 1987.

[33] Milind Kulkarni, Vijay S. Pai, and Derek L. Schuff. Towards architecture independent metrics for multicore performance analysis. *SIGMETRICS Performance Evaluation Review*, 38(3):10–14, 2010.

[34] John Levon and Philippe Elie. Oprofile: A system profiler for Linux, 2004.

[35] John DC Little. OR FORUM: Little's Law as Viewed on Its 50th Anniversary. *Operations Research*, 59(3):536–549, 2011.

[36] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 190–200. ACM, 2005.

[37] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[38] Barton P. Miller, Morgan Clark, Jeffrey K. Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):206–217, 1990.

[39] Barton P. Miller, Cathryn Macrander, and Stuart Sechrest. A distributed programs monitor for berkeley UNIX. In *ICDCS*, pages 43–54, 1985.

[40] Barton P. Miller and Cui-Qing Yang. IPS: An interactive and automatic performance measurement tool for parallel and distributed programs. In *ICDCS*, pages 482–489, 1987.

[41] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of Java profilers. In *PLDI*, pages 187–197. ACM, 2010.

[42] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.

[43] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Online computation of critical paths for multithreaded languages. In *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 301–313. Springer, 2000.

[44] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *OSDI*, pages 107–120. USENIX Association, 2012.

[45] Robert Snelick, Joseph JáJá, Raghu Kacker, and Gordon Lyon. Synthetic-perturbation techniques for screening shared memory programs. *Software Practice & Experience*, 24(8):679–701, 1994.

[46] Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie. Space-efficient time-series call-path profiling of parallel applications. In *SC*. ACM, 2009.

[47] Sarah AM Talbot, Andrew J Bennett, and Paul HJ Kelly. Cautious, machine-independent performance tuning for shared-memory multiprocessors. In *Euro-Par'96 Parallel Processing*, pages 106–113. Springer, 1996.

[48] Nathan R. Tallent and John M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPOPP*, pages 229–240. ACM, 2009.

[49] Ken Thompson and Dennis M Ritchie. *UNIX Programmer's Manual*. Bell Telephone Laboratories, 1975.

[50] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPOPP*, pages 185–196. ACM, 2008.

[51] Wikipedia. Intel parallel studio — wikipedia, the free encyclopedia, 2014. [Online; accessed 30-July-2014].

[52] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *ICDCS*, pages 366–373, 1988.

[53] Cui-Qing Yang and Barton P. Miller. Performance measurement for parallel and distributed programs: A structured and automatic approach. *IEEE Trans. Software Eng.*, 15(12):1615–1629, 1989.

[54] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *PLDI* [1], pages 67–76.

[55] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO*, pages 47–58. IEEE Computer Society, 2009.