Causal Profiling: Finding Optimizations that Count

Charlie Curtsinger Emery D. Berger

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

Abstract

While traditional profilers reveal where a program spends its time, they do not indicate where developers should spend their time. This paper introduces *causal profiling*, a novel profiling approach that not only indicates exactly where programmers should focus their optimization efforts, but also quantifies their potential impact. Causal profiling works by performing a series of performance experiments at runtime. These experiments quantify the impact of any potential optimization via *virtual speedups*, which emulate the effect of real speedups by slowing down concurrently-executing tasks. Causal profiling further departs from traditional profilers by identifying optimization opportunities that increase throughput or reduce latency. We present Coz, a prototype causal profiler, and empirically demonstrate its efficiency and effectiveness at guiding optimization efforts.

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 [Curtsinger and Berger 2013]. 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` [Levon and Elie 2004; kernel.org 2014; Graham et al. 1982].

Unfortunately, even when a profiler accurately reports where a program is spending the bulk of its time [Mytkowicz et al. 2010], this information can lead programmers astray. Where programs spend their time is not necessarily where programmers should focus their optimization efforts: *correlation does not imply causation*. 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 the `a` function is on the critical path [Intel 2014].

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 execution time (y-axis). The graph shows that optimizing `a` or `b` in isolation would have little impact on execution time.

```
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.

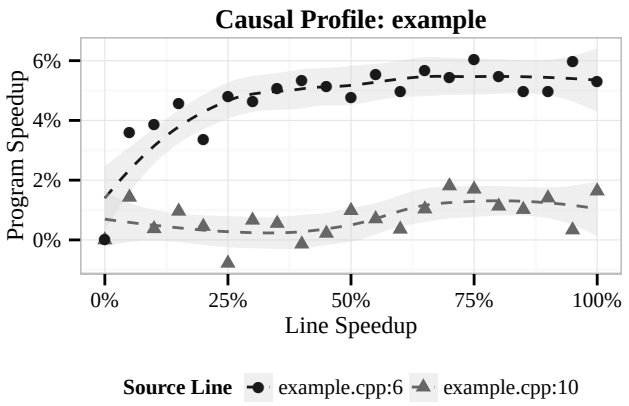

Figure 2: A causal profile for the program in Figure 1 generated by Coz. The y-axis shows the program speedup that would be achieved by speeding up the given line of code by the percentage given in the x-axis (the gray area shows standard error). Unlike past profilers, causal profiling accurately predicts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

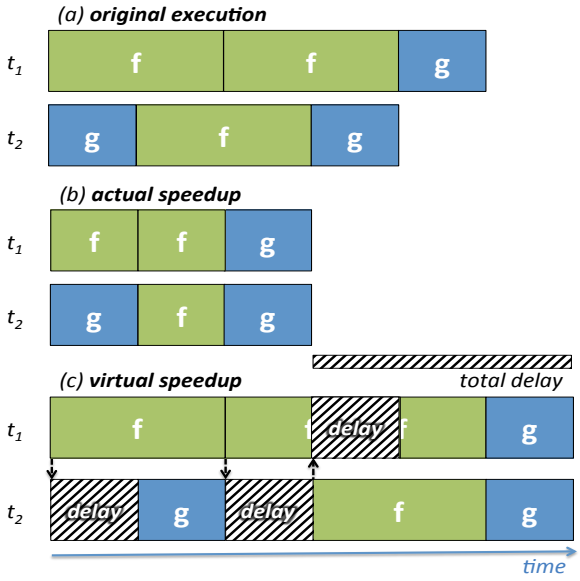

Figure 3: An illustration of virtual speedup: (a) shows the original execution of two threads, running functions `f` and `g`; (b) shows the effect of *actually* halving the runtime of `f`; (c) shows the effect of *virtual speedup*, inserting delays in other threads whenever `f` runs (denoted by the arrows).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

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).

We show that causal profiling accurately predicts optimization opportunities, and that it is effective at guiding optimization efforts. We apply Coz to the the extensively studied PARSEC benchmark suite, and identify the best optimization opportunities in each benchmark. With Coz's guidance, we were in short order able to increase the performance of two benchmarks by 8% and 20%. These performance gains closely match Coz's predictions of 9% and 21%.

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 (§3).
2. It presents **Coz**, a prototype implementation of causal profiling. It describes its implementation (§4), and demonstrates its efficiency and effectiveness at locating and quantifying optimization opportunities (§5).

2. 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.

2.1 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 care must be taken to ensure that sampling is unbiased [Mytkowicz et al. 2010].

The UNIX `prof` tool and `oprofile` both use sampling exclusively [Thompson and Ritchie 1975; Levon and Elie 2004]. `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 [Graham et al. 1982]. `gprof` produces a call graph profile, which counts invocations of functions segregated by caller. Cho et al. [2013] 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 [Ball and Larus 1996; Ammons et al. 1997].

Binary instrumentation enables detailed performance measurement of unmodified binaries, sometimes at high cost. Valgrind includes two performance measurement tools that measure cache and heap performance [Nethercote and Seward 2007]. Pin is the basis for the Intel Parallel Amplifier, which identifies contended locks, bottlenecks, and hotspots [Bach et al. 2010; Luk et al. 2005].

2.2 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. Miller and Yang [1987] describe IPS, which uses traces from message-passing programs to identify the critical path. They then report time spent in each procedure on the critical path. IPS-2 extends this approach with limited support for shared memory parallelism [Yang and Miller 1989; Miller et al. 1990]. Other critical path profilers rely on languages with first-class threads and synchronization to identify the critical path [Hill et al. 1998; Oyama et al. 2000; Szebenyi et al. 2009]. Using the program activity graph, Hollingsworth and Miller [1994] 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. These metrics are similar to the peak impact measured by causal profiling, 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. PerfExpert, Paradyn, and Diamond et al. [2011] use hardware performance counters to identify hardware-level performance bottlenecks [Miller et al. 1995; Burtscher et al. 2010]. Bottlenecks, a profile analysis tool, uses heuristics to identify bottlenecks using call-tree profiles [Ammons et al. 2004]. Given call-tree profiles for different executions, Bottlenecks can pinpoint which procedures are responsible for the difference in performance. Unlike causal profiling, these tools do not predict the performance impact of removing bottlenecks.

Visual Studio's contention profiler identifies locks that are responsible for significant thread blocking time [Goldin 2010]. BIS uses similar techniques to identify highly contended critical sections on asymmetric multiprocessors, and automatically migrates performance-critical code to faster cores [Joao et al. 2012]. Bottle graphs present thread execution time and parallelism in a visual format that highlights program bottlenecks [Du Bois et al. 2013]. All three 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 [von Praun et al. 2008; Zhang et al. 2009; Garcia et al. 2011]. 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 et al. [2010] present general metrics for available parallelism and scalability. The Cilkview scalability analyzer uses performance models for Cilk's constrained parallelism to estimate the performance effect of adding additional hardware threads [He et al. 2010]. 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 [Anderson and Lazowska 1990]. CPPROFJ extends this approach to Java programs with aspects [Hall 2002]. CPPROFJ 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 [2009] extend this approach further to support Cilk programs, with an added category for time spent managing parallelism. The WAIT tool adds fine-grained categorization for enterprise Java libraries to identify bottlenecks in large-scale production Java sys-

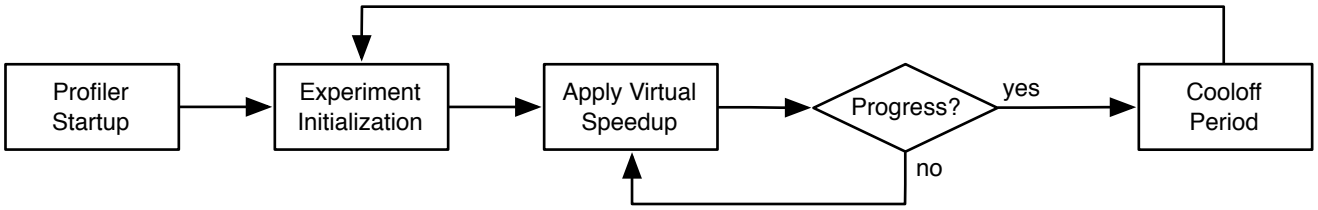

Figure 4: Coz's profiling workflow. At startup, Coz builds a map of source information for the main executable and libraries. To begin an experiment, Coz selects a random source line and speedup amount. Using virtual speedups, Coz creates the effect of optimizing the selected line. After all progress points have been visited and a minimum experiment duration has elapsed, Coz stops the experiment and records the results. After a cooloff period, Coz selects a new random line and speedup, and begins the next experiment.

tems [Altman et al. 2010]. Unlike causal profiling, these profilers can only capture interference between threads that directly affects their scheduler state.

2.3 Performance Experimentation

Several systems have employed delays to extract information about program execution times. Mytkowicz et al. [2010] use inserted delays to validate the output of profilers on single-threaded Java programs. Snelick et al. [1994] use delays to profile parallel programs. 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 [Brown et al. 2001]. ADD requires a complete enumeration of system components, and requires developers to insert performance perturbations manually. Neither approach can quantify the effect of optimizations, which causal profiling measures via its virtual speedup experiments.

3. Causal Profiling Overview

Causal profiling relies on several key ideas to provide developers with actionable profiles. *Virtual speedups* allow a causal profiler to automatically create the effect of optimizing any fragment of code. *Progress points* allow the profiler to 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, a causal profiler 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 place progress points at a source location where some useful work has been completed. These points allow a causal profiler to 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.

3.1 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. Figure 4 shows Coz's profiling workflow, which we describe in detail below.

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 4.4, 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 will only end a performance experiment once sufficient time has elapsed, and all progress points have been visited. The profiler thread periodically checks the count of visits to progress points. 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 wait for a cooloff period to allow any remaining samples to be processed.

4. 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.

4.1 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 [DWARF Debugging Information Format Committee 2010]. 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 [Free Software Foundation Free Software Foundation]. 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 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 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. 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 a configuration that specifies which hardware or software event to count (e.g., 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.

4.2 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 zero and 100%, in increments of 5%. For any given virtual speedup, the effect on program performance is $1 - \frac{p_s}{p_0}$, where p_0 is the period between progress point visits with zero 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 between 5% and 100%.

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.

4.3 Running a Performance Experiment

Once a performance experiment has started, each of the program's threads processes samples and inserts delays to perform virtual speedups. Once a 100ms have elapsed, the profiler thread periodically checks if progress points have been reached at least five times. After the experiment ends, a cooloff period of 10ms.

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.

4.4 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, 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.

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.

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.

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.

4.5 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

Progress Points and Optimization Opportunities for PARSEC Benchmarks

Benchmark	*Progress Point*	*Top Optimization*	*Benchmark*	*Progress Point*	*Top Optimization*
b.scholes	blackscholes.c:259	blackscholes.m4.cpp:183	f.animate	all barrier_wait calls	pthreads.cpp:849
bodytrack	TicketDispenser.h:106	ParticleFilter.h:262	freqmine	fp_tree.cpp:383	fp_tree.cpp:301
canneal	annealer_thread.cpp:87	netlist_elem.cpp:82	s.cluster	all barrier_wait calls	streamcluster.cpp:1763
dedup	encoder.c:189	hashtable.c:217	swaptions	HJM_Securities.cpp:99	HJM_SimPath....cpp:154
facesim	taskQDistCommon.c:109	MATRIX_3X3.h:136	vips	threadgroup.c:360	im_Lab2LabQ.c:98
ferret	ferret-parallel.c:398	ferret-parallel.c:320	x264	encoder.c:1165	common.c:687

Table 3: The locations of inserted progress points for each of the PARSEC benchmarks, and the top optimization opportunities that Coz identifies (the full line for `swaptions` is `HJM_SimPath_Forward_Blocking.cpp:154`). To demonstrate Coz's effectiveness, we used Coz's output to manually optimize two benchmarks: `ferret` and `dedup`. We exclude the `raytrace` program due to time constraints.

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: $L = \lambda W$ [Little 2011]. 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.

4.6 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:

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

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:

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

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),

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

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

5. 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?

5.1 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

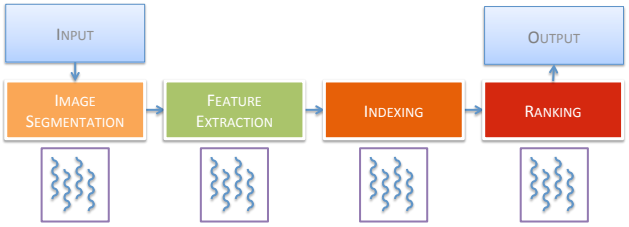

Figure 5: `ferret`'s pipeline. The middle four stages each have an associated thread pool; the input and output stages each consist of one thread. The colors represent the impact on throughput of each stage, as identified by Coz: green is low impact, orange is medium impact, and red is high impact.

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. Coz is run with the default sampling period of 1ms, and a sample batch size of ten. Experiments run for a minimum of 100ms with a cooloff period of 10ms after each experiment. An experiment does not end until every progress point has been reached at least five times. Due to space limitations, we only profile throughput (and not latency) in this evaluation.

5.2 Effectiveness

We run the PARSEC benchmark suite with Coz, with progress points and highlighted optimization opportunities shown in Table 3. For two of these applications, we evaluate the effectiveness of using Coz to guide optimizations (`ferret` and `dedup`).

5.2.1 Case Study: `ferret`

The `ferret` benchmark performs a content-based similarity search of images. `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. Each stage has its own thread pool. `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 `--source-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).

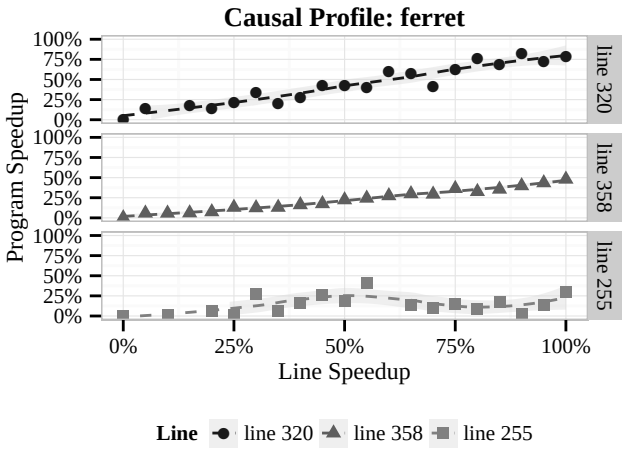

Figure 6: Coz output for the unmodified `ferret` application. The x-axis shows the amount of virtual speedup applied to each line, versus the resulting change in throughput on the y-axis. The top two lines are executed by the indexing and ranking stages; the third line is executed during image segmentation.

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

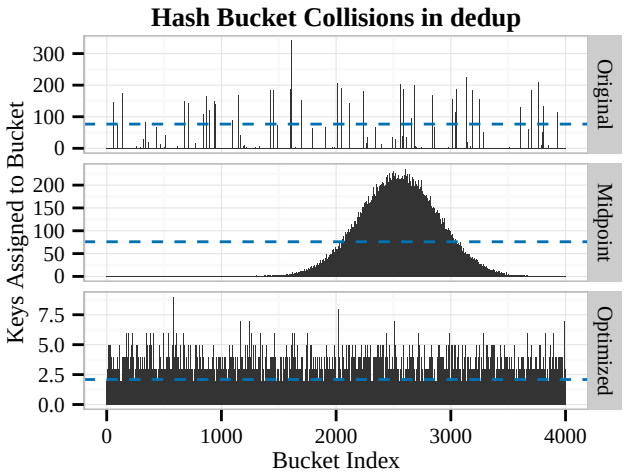

Figure 7: In the `dedup` benchmark, Coz identified hash bucket traversal as a bottleneck. Collisions per-bucket for the first 4000 buckets before, midway through, and after optimization of the `dedup` benchmark (note different y-axes). The dashed horizontal line shows average collisions per-utilized bucket for each version. Replacing `dedup`'s hash function yielded an 8% runtime performance improvement.

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.

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.

5.2.2 Case Study: `dedup`

Our second case study is on `dedup`, an application that performs parallel file compression with 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.

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 bin. This suggests that `dedup`'s shared hash table has a significant number of collisions.

Increasing the hash table size had no effect on performance, meaning a degenerate hash function must be responsible for the high degree of hashtable collisions. `dedup`'s hash function maps keys to just 2.3% of the available buckets. The peak number of items in a single bucket was 397.

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% runtime performance improvement. Figure 7 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 twelve lines of code.

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 decreased 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%.

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.

5.3 Accuracy

We use the results of the optimizations performed in our two case studies to evaluate the accuracy of Coz's predictions.

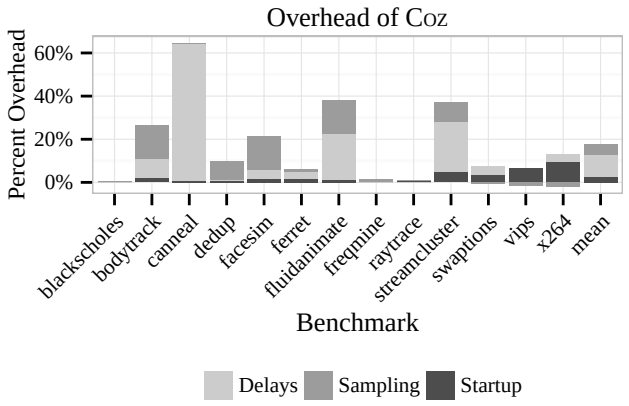

Figure 8: Percent overhead for each of Coz's possible sources of overhead. *Delays* are the overhead due to adding delays for virtual speedups, *Sampling* is the cost of collecting and processing samples, and *Startup* is the initial cost of processing debugging information. Note that sampling results in slight performance *improvements* for `swaptions`, `vips`, and `x264`.

The optimization strategy we employed for `ferret` lets us evaluate the accuracy of Coz's predictions. By increasing the number of threads for the indexing stage from 16 to 22, we have effectively sped up line 320 by 27%. Coz predicted that this improvement would result in a 21.4% program speedup, which is nearly the same as the 20% we observe.

In the `dedup` benchmark, 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 this line by 96%. For this speedup, Coz predicted a performance improvement of 9%; we observe an improvement of 8%.

Both of these results demonstrate that Coz's predictions are highly accurate.

5.4 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 8 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.

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.

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.

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

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

Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *PLDI*. ACM, 85–96.

Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. 2004. Finding and Removing Performance Bottlenecks in Large Systems. In *ECOOP (Lecture Notes in Computer Science)*, Vol. 3086. Springer, 170–194.

Thomas E. Anderson and Edward D. Lazowska. 1990. Quartz: A Tool for Tuning Parallel Program Performance. In *SIGMETRICS*. 115–125.

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. 2010. Analyzing Parallel Programs with Pin. *Computer* 43, 3 (March 2010), 34–41. DOI:http://dx.doi.org/10.1109/MC.2010.60

Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *MICRO*. 46–57.

Aaron B. Brown, Gautam Kar, and Alexander Keller. 2001. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. In *Integrated Network Management*. IEEE, 377–390.

Martin Burtscher, Byoung-Do Kim, Jeffrey R. Diamond, John D. McCalpin, Lars Koesterke, and James C. Browne. 2010. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In *SC*. IEEE, 1–11.

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

Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 12.

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

Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 2013. 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)*. ACM, New York, NY, USA, 355–372. DOI: http://dx.doi.org/10.1145/2509136.2509529

DWARF Debugging Information Format Committee 2010. *DWARF Debugging Information Format, Version 4*. DWARF Debugging Information Format Committee. http://http://dwarfstd.org/

Free Software Foundation. *Debugging with GDB* (tenth ed.). Free Software Foundation. https://www.sourceware.org/gdb/onlinedocs/gdb/

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

Maxim Goldin. 2010. Thread Performance: Resource Contention Concurrency Profiling in Visual Studio 2010. *MSDN magazine* (2010), 38. http://msdn.microsoft.com/en-us/magazine/ff714587.aspx

Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 1982. gprof: a Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*. ACM, 120–126.

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

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

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

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

Intel 2014. *Intel VTune Amplifier 2015*. Intel. https://software.intel.com/en-us/node/529213

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

kernel.org 2014. *perf: Linux profiling with performance counters*. kernel.org. https://perf.wiki.kernel.org

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

John Levon and Philippe Elie. 2004. Oprofile: A system profiler for Linux. (2004).

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

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

Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. 1995. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* 28, 11 (1995), 37–46.

Barton P. Miller, Morgan Clark, Jeffrey K. Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. 1990. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Trans. Parallel Distrib. Syst.* 1, 2 (1990), 206–217.

Barton P. Miller and Cui-Qing Yang. 1987. IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs. In *ICDCS*. 482–489.

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

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

Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. 2000. Online Computation of Critical Paths for Multithreaded Languages. In *IPDPS Workshops (Lecture Notes in Computer Science)*, Vol. 1800. Springer, 301–313.

Robert Snelick, Joseph JáJá, Raghu Kacker, and Gordon Lyon. 1994. Synthetic-perturbation Techniques for Screening Shared Memory Programs. *Software Practice & Experience* 24, 8 (1994), 679–701.

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

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

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

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

Cui-Qing Yang and Barton P. Miller. 1989. Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach. *IEEE Trans. Software Eng.* 15, 12 (1989), 1615–1629.

Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. 2009. Alchemist: A Transparent Dependence Distance Profiling Infrastructure. In *CGO*. IEEE Computer Society, 47–58.