University of Massachusetts Amherst

ScholarWorks@UMass Amherst

Doctoral Dissertations

Dissertations and Theses

July 2016

Effective Performance Analysis and Debugging

Charles M. Curtsinger

Recommended Citation

Curtsinger, Charles M., "Effective Performance Analysis and Debugging" (2016). *Doctoral Dissertations*. 632.
https://scholarworks.umass.edu/dissertations_2/632

EFFECTIVE PERFORMANCE ANALYSIS AND DEBUGGING

A Dissertation Presented

by

CHARLES M. CURTSINGER

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2016

College of Information and Computer Sciences

EFFECTIVE PERFORMANCE ANALYSIS AND DEBUGGING

A Dissertation Presented

by

CHARLES M. CURTSINGER

Approved as to style and content by:

Emery D. Berger, Chair

Yuriy Brun, Member

Arjun Guha, Member

Nicholas G. Reich, Member

James Allan, Chair
College of Information and Computer Sciences

DEDICATION

For Nikki.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Emery Berger, for his support during my graduate career. Your advice and mentoring shaped every element of this dissertation, and your tireless advocacy for me have shaped my entire career. None of this would have been possible without you. Thank you.

Thanks as well to my fellow students Gene Novark, Tongping Liu, John Altidor, Matt Laquidara, Kaituo Li, Justin Aquadro, Nitin Gupta, Dan Barowy, Emma Tosch, and John Vilk. You took the daunting process of spending over half a decade working seemingly endless hours and made it truly fun. I hope I have helped you with your work as much as you helped me with mine.

I would also like to thank my research mentors and colleagues Steve Freund, Yannis Smaragdakis, Scott Kaplan, Arjun Guha, Yuriy Brun, Ben Zorn, Ben Livshits, Steve Fink, Rodric Rabbah, and Ioana Baldini. Your guidance and support over the years have been an enormous help, and it has been a real privilege to work with you all.

Finally, I would like to thank my parents Jim and Julie, my brother Dan, my wife Nikki, and parents in-law Kathy and Cary, for their unwavering support and understanding. Thank you for your constant encouragement, for listening to the long technical explanations you may or may not have asked for, and for tolerating all the weekends and holidays I spent working instead of spending time with you. I could never have finished a PhD without your support and encouragement.

ABSTRACT

EFFECTIVE PERFORMANCE ANALYSIS AND DEBUGGING

MAY 2016

CHARLES M. CURTSINGER

B.Sc., UNIVERSITY OF MINNESOTA

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Emery D. Berger

Performance is once again a first-class concern. Developers can no longer wait for the next generation of processors to automatically "optimize" their software. Unfortunately, existing techniques for performance analysis and debugging cannot cope with complex modern hardware, concurrent software, or latency-sensitive software services.

While processor speeds have remained constant, increasing transistor counts have allowed architects to increase processor complexity. This complexity often improves performance, but the benefits can be brittle; small changes to a program's code, inputs, or execution environment can dramatically change performance, resulting in unpredictable performance in deployed software and complicating performance evaluation and debugging. Developers seeking to improve performance must resort to manual performance tuning for large performance gains. Software profilers are meant to guide developers to important code, but conventional profilers do not produce actionable information for concurrent applications. These profilers report where a program spends its time, not where optimizations will yield performance improvements. Furthermore, latency is a critical measure of performance for software services and interactive applications, but conventional profilers measure only throughput. Many performance issues appear only when a system is under high load, but generating this load in development is often impossible. Developers need to identify and mitigate scalability issues before deploying software, but existing tools offer developers little or no assistance.

In this dissertation, I introduce an empirically-driven approach to performance analysis and debugging. I present three systems for performance analysis and debugging. Stabilizer mitigates the performance variability that is inherent in modern processors, enabling both predictable performance in deployment and statistically sound performance evaluation. Coz conducts *performance experiments* using *virtual speedups* to create the effect of an optimization in a running application. This approach accurately predicts the effect of hypothetical optimizations, guiding developers to code where optimizations will have the largest effect. Amp allows developers to evaluate system scalability using *load amplification* to create the effect of high load in a testing environment. In combination, Amp and Coz allow developers to pinpoint code where manual optimizations will improve the scalability of their software.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Developers can no longer count on increasing processor speed to automatically “optimize” their programs; performance must be a first-class concern. Unfortunately, the tools and practices supporting software performance have been neglected during decades of exponential growth in processor performance. During the last three decades, we have moved from a model of batch execution on relatively simple processors to massively-parallel software services running on processors with billions of components. These changes make it nearly impossible to reliably measure, predict, and debug performance using techniques from the 1980s and earlier. In this dissertation, I introduce three novel software performance tools that enable reliable performance measurement, effective performance debugging, and accurate performance prediction.

1.1 Performance Measurement.

Clock speed is an important factor in determining a processor’s performance, but it is by no means the only one. The speed of memory accesses has not kept pace with clock speed increases, so processors now rely on complex heuristics like caching and prefetching to hide the latency of memory accesses. When these heuristics work well, processors run orders of magnitude faster than they would without them. However, caches are have undesirable edge cases that lead to significant performance degradations. Tiny changes in the placement of a program’s code or data in memory can have catastrophic effects on performance. This makes it particularly difficult to compare the performance of two versions of a program. Programs with different algorithms almost certainly have different layouts as well; if one version runs faster than the other, is the root cause the new algorithm or the layout change? The confounding effect of program layout can lead developers to make poor decisions during performance evaluation, but can also lead to unpredictable performance of deployed software.

1.2 Performance Profiling.

Processor clock speeds have remained relatively constant for the last ten years. While larger caches and improvements in processor design still help improve the performance of new processors, the year-over-year improvements are small. As a result, developers must now resort to manual performance tuning to improve the performance of their code. Software profilers are designed to help developers decide where to focus during manual performance tuning. Unfortunately, existing profilers do not provide actionable information for concurrent programs; they only report where programs spend their time, not where performance improvements would reduce end-to-end runtime or latency in a parallel program. This leads developers to pursue manual optimizations with little or no potential payoff.

1.3 Performance Prediction.

The era of batch computation is long gone; instead, computational workloads are now dominated by service-oriented software. These services must scale to thousands or millions of concurrent requests while remaining responsive, or they can bring down an entire network of applications. This makes performance testing particularly critical. However, generating sufficient load in testing infrastructure is often impractical or impossible, and existing performance analysis and debugging techniques do not help developers identify code that limits scalability. Developers are left with only one option: deploy software with inadequate performance testing. When performance bugs inevitably appear in production, developers have no means to reproduce and investigate these bugs in the development environment.

1.4 Contributions

This dissertation presents three systems that support a new approach to performance analysis and debugging. Collectively, these systems enable reliable performance measurement, effective performance debugging, and accurate performance prediction.

1.4.1 STABILIZER: Predictable and Analyzable Performance

Developers use performance measurements to decide whether a code change has unacceptable performance overhead, or whether an optimization significantly improves performance. However, standard performance evaluation strategies are measuring more than just the effect of the code change; any change to a program's code also changes its layout, which can have a large effect on performance that obscures the real cost or benefit of code changes. Repeated executions of the program do not control for the performance effect of layout; every run executes with the same memory layout, so layout changes produce a consistent bias in experimental results. This dissertation presents STABILIZER, a system that controls for the effect of memory layout using randomization, and presents an representative performance evaluation of compiler optimizations performed with STABILIZER and sound statistical methods.

1.4.2 COZ: Finding Code that Counts with Causal Profiling

Developers need to know which parts of an application are important for performance; they use this information to select code for manual performance tuning, or to determine which code should be left undisturbed when adding new features that could impose runtime overhead. To find performance-critical code, developers must consider all dependencies and sources of contention between an application's concurrently-executing tasks. Existing tools for identifying performance critical code, *software profilers*, produce misleading results because they do not consider the complex interactions between tasks in concurrent programs. Existing software profilers, which typically measure the amount of time spent executing each of a program's functions, fail to answer developers' primary question: *where will optimizations have the largest effect on program performance?* This dissertation presents *causal profiling*, a novel profiling technique that simulates the effect of optimizations to specific code fragments to empirically establish the potential benefit of a real optimization. Using COZ, a prototype causal profiler, I identify optimization opportunities in Memcached, SQLite, and the PARSEC benchmarks that lead to program speedups as large as 68%, and show that the effects of these optimizations match COZ's predictions.

1.4.3 AMP: Scalability Prediction and Guidance with Load Amplification

The growth in high performance software services poses a problem for performance evaluation and debugging. These systems are designed to handle enormous load—significantly more load than can be generated on a single test machine. Developers have no means to test these systems under realistic loads before deployment. As a result, performance issues may lie dormant in an application until it is deployed. This dissertation presents AMP, a system that simulates the effect of higher load in a running program with *load amplification*, and directly measures the system performance under the simulated load. In combination with COZ, AMP can be used to precisely locate code where optimizations will improve performance when the system is under load.

CHAPTER 2

BACKGROUND

While processor designs and software architectures have changed dramatically since the 1980s, the key ideas in software performance are at least this old. This chapter introduces the basics of software and hardware pre-1990 and the techniques for performance analysis and debugging from this area, which largely remain in use today.

2.1 Software pre–1990

Before the era of personal computers, the vast majority of programs were run a *batch jobs*, where end-to-end runtime was all that mattered. On early computers, batch jobs executed in isolation with near-complete control over a machine and all of its resources. As computers grew in computational power, multi-user systems became common; these machines maintained the illusion of exclusive control of all resources, but hardware resources were actually multiplexed over all users' jobs to execute many independent tasks in parallel. This usage model continued through the 1980s on minicomputers, which displaced large mainframes.

2.1.1 The Batch Execution Model

Batch jobs lend themselves to a simple model of performance; because each job must complete before the next can be run, runtime is additive. If a system has three jobs to execute, saving a minute on any of the three jobs would shorten the total runtime by one minute. It does not matter where this saved minute comes from, the effect is the same.

In the multi-user model, different users can execute batch jobs concurrently. While multi-user machines share the resources of a single machine between many concurrently executing tasks, these tasks were almost entirely independent. Communication between different users' computations were rare or impossible, depending on the hardware and operating system. As a result, batch jobs executed by one user resemble batch jobs executed in isolation on a less-powerful machine. The simple performance model for a single-user machine remains useful for multi-user machines as well.

2.1.2 Interactive Computation

While batch computation dominated on high performance machines, an alternative model—*interactive computation*—was growing in use, especially on personal computers. Rather than churning on data until a result is available, interactive applications perform computation in response to user inputs that occur during normal execution rather than just at the beginning. A spreadsheet application from this era would be considered an interactive application, but its computational model is remarkable similar to batch computation. The program runs in an *event loop*, which repeatedly checks for user input. Once a user has updated a cell or formula the program initiates a recalculation (effectively a batch computation). The program will not respond to any additional inputs until after the recalculation is complete and the program can resume executing the event loop. To improve the responsiveness of a spreadsheet application a developer simply needs to reduce the execution time of the recalculation, which is effectively a batch computation run in response to user input.

2.2 Hardware pre-1990

The additive property of software runtimes extended to the hardware level as well. Programs are composed of instructions, which a processor must execute in order. Executing an instruction requires several steps:

1. fetch the instruction from the computer's main memory;
2. decode the instruction from its in-memory representation;
3. collect the instruction's inputs from memory and registers, a limited set of locations used to store values that will be needed quickly;
4. perform the operation specified by the instruction; and finally,
5. place the result(s) of the instruction in registers or memory.

Early processors performed each operation in sequence for one instruction before moving on to the next one. While the execution times of some steps in instruction execution could vary, the time to run a single instruction from start to finish was often tied to a clock with a period long enough to allow *any* instruction to complete in the allotted time.

By the 1970s, many computers used *pipelined execution* to increase instruction throughput. Rather than performing all of the steps in instruction execution for a single instruction before moving on to the next instruction, a pipelined processor executes multiple instructions concurrently in stages. At any given point, a pipelined processor can be fetching one instruction, decoding another, collecting inputs for a third, and so on. The net effect of this change is improved performance, but the time required to execute a single instruction is just as predictable as in the non-pipelined model.

2.2.1 Performance Variability

Software performance was not entirely consistent; one early source of performance variability in early computers was in memory accesses. Many computers used inexpensive rotating drum memory; the storage was spread across a two dimensional area wrapped around a cylinder with a reading head spanning the length of the cylinder. An entire row of memory (the length of the cylinder) could be read at once, but reading along the circumference would require rotating the drum until the desired row was under the read head. The time required to load from drum memory depends on the distance between the current head location and the target column. A common practice among programmers seeking to optimize programs was to carefully plan memory use so the drum would already be rotated into place when a row of memory was needed. This level of control allowed careful developers to eliminate a source of unpredictability in performance through careful design and a deep understanding of the underlying hardware.

While rotating drum memory was largely displaced by core memory (which has no moving parts) by 1960, this view of hardware performance as a predictable outcome applied equally well to newer memory systems and processors designs until the 1990s. As a result, much of the work on measuring and improving performance is built on two core assumptions:

1. A program interacts with hardware in predictable, controllable ways, and
2. runtime is additive. In other words, a program's total runtime is the sum over the execution time of all its parts.

While reasonable at the time, these assumptions no longer hold; interactions between complex hardware components make performance virtually impossible to predict. Despite this fact, the practices in software performance engineering developed during this era remain in use today.

2.3 Measuring Performance

Measuring the performance of a program seems like a straightforward process; start a timer, run the program, then stop the timer. There are, of course, additional details that must be addressed:

1. How should time be measured to ensure the most accurate measurement?
2. What operations could the system perform that may distort the program's runtime, and how can these be disabled or accounted for?
3. What inputs should the program be given?

Answers to these questions are largely system- and program-dependent, but coming up with answers is not especially challenging. Some care was required to time the program accurately, but the majority of the effort in evaluating a program's performance went toward choosing representative inputs. If real inputs to a program exercise parts of the code that are not used during performance evaluation, developers may miss important opportunities to improve performance. Producing representative inputs requires collecting program inputs that will result in the same parts of the program running for approximately the same time as under real workloads.

Evaluating the performance of application-independent systems such as a new processor architecture, operating system, or runtime library adds one dimension of complexity to performance evaluation. These systems impact the performance of different programs in different ways, so which programs should be run to test the system's performance? There have been efforts to address this issue with the development of standard benchmark suites, although the selection of programs, inputs, and performance measures for benchmark suites can be controversial [69, 31]. Debates over which benchmark applications should be included, the inputs used to drive these applications, and the type of mean used to summarize results were and are still common. However, the implications for developers were minimal; given an application and representative inputs, developers could easily evaluate the performance of a new processor, operating system, or version of the application.

2.4 Improving Performance

There are both automatic and manual techniques for improving software performance. Automatic performance improvements largely come from *compiler optimizations*, which transform the structure of a program while maintaining semantic equivalence to the original program. Developers can manually improve a program's performance by changing algorithms and data structures, the input and output formats, and other high-level design decisions that require application-specific knowledge. Manual performance improvements require significant developer effort, so they often rely on *software profilers* to identify code that is important enough to justify manual effort. This section introduces the basic methods of automatic performance improvements with compiler optimizations, the mechanisms used by software profilers, and how software profiles are interpreted.

2.4.1 Compilation

Compilers are responsible for transforming code in a high level language to instructions the processor can execute. This process contains three major steps:

1. **Translation to intermediate representation (IR).** In this phase of compilation, often called the "front-end," high level syntax is parsed and transformed to an intermediate representation. This intermediate representation is designed to be amenable to analysis, transformation, and generation of the final machine code. While important, this step typically does not impact the performance of the final executable file.

2. **Analysis and transformation.** Once a program has been translated to IR, the compiler analyzes the IR and applies transformations to improve the performance of the final executable. These transformations fall under the general category of "optimizations," although the result is not necessarily optimal. This phase is sometimes referred to as the "middle-end" of the compiler.

3. **Code generation.** Once the compiler has completed transformations, the "back-end" is responsible for translating IR to machine-executable code. Two major components of this phase include register allocation, which assigns program variables in the IR to registers, a limited set of "variables" available on a processor; and instruction selection, the process of determining exactly which hardware instruction(s) should be used to to carry out an operation or sequence

of operations in the program's intermediate representation. Strategies for register allocation, instruction selection, and other aspects of code generation play an important role in program performance.

2.4.1.1 Compiler Optimizations

Transformations to a program's intermediate representation and strategies for code generation can have a significant impact on the program's performance. However, these strategies are ultimately just heuristics tailored to specific properties of a target processor architecture. While the details of many optimizations are highly technical or processor-dependent, many optimizations are common to most processors. Optimizations are generally organized into levels, where selecting a level enables an entire group of optimizations. Compilers generally use the command line flag `-O` to specify the optimization level. The `-O0` flag enables only the most basic strategies for code generation, `-O1` enables only a few simple transformations, `-O2` (usually the default) applies a standard suite optimizations, and `-O3` applies "advanced" optimizations that are often computationally expensive or deemed experimental. Some compilers include additional levels of optimization such as `-O4`, which runs the program to collect dynamic information which is then used to guide transformations, and `-Os`, which targets a smaller executable by enabling only the transformations that do not increase executable size. The `gcc` compiler generously accepts optimization levels of five and above, but the effect is the same as passing in the `-O4` option. This subsection briefly introduces a few of these optimizations to give a sense of what a compiler can and cannot do to improve performance automatically.

2.4.1.1.1 Constant Folding & Propagation. Compilers often use an intermediate representation in what is called single static assignment (SSA) form. In this form, variables are assigned at their declarations, and cannot be re-assigned. This simplifies the task of determining the set of possible values for a variable at any given point. The value may be a constant, the result of some other instruction, or a load from memory. Constant folding locates operations that are always given constant values as inputs and replaces the operation with its result. This eliminates an instruction from the IR—and most likely a machine instruction as well—and creates a new constant value that could potentially be used in another round of constant folding. Constant propagation complements this transformation by replacing variables that are known to contain constant values with the constant itself, enabling further folding and propagation.

2.4.1.1.2 Common Subexpression Elimination. The aptly named "common subexpression elimination" optimization identifies and eliminates repeated calculations of the same value. By computing an intermediate result and storing it in an IR variable, multiple operations that use this value can share the result rather than recomputing it. Unlike constant folding and propagation, this transformation is suitable for values that are unknown at compile time such as program inputs, values loaded from memory, or values that depend on computations that use unknown values.

2.4.1.1.3 Function Inlining. Function inlining takes calls from one function (the caller) to another (the callee), and replaces the function call by embedding the body of the callee function directly in the caller. This eliminates the modest overhead of making a function call, but the value in function inlining is largely in enabling other optimizations. If a function is called with constants as arguments, inlining the function allows constant folding and propagation to cross the function boundary, effectively specializing the inlined function for this callsite.

2.4.1.1.4 Loop Unrolling. Many program optimizations target loops, which necessarily make up a significant portion of a program's execution. A program with two billion instructions, a very large number even for complex software such as an operating system, would run for just one second on a 2GHz processor if it did not contain loops.

Loops are implemented using conditional branches in both IR and machine code; these languages do not have high-level control constructs like `if` statements or `while` loops. Before each iteration of a loop, the program executes a conditional branch, which jumps to a specific program point only if some value meets a condition (e.g. "jump to LABEL if x is less than 10"). The performance cost of a single conditional branch is very small, but loops that run for thousands of iterations will execute thousands of conditional branches. Loop unrolling takes the body of the loop and replaces it with a new body that executes multiple iterations in sequence, dividing the number of conditional branches by some constant. One goal of loop unrolling is to fit the loop body into a single *cache line*, the size of a block of memory kept in fast on-chip cache memory (see Chapter 3 for a detailed discussion of caches). Given this upper limit, most compilers unroll loops so each new iteration will execute between two and eight iterations of the original loop.

2.4.1.1.5 Strength Reduction. Strength reduction identifies special cases of expensive operations—often multiplication or division—and translates them into less-expensive operations. For example, a processor can multiply an integer by two by simply adding a zero to the right side of the number in binary representation; this is the same as multiplying a decimal number by ten by simply adding a zero. This operation, called a left shift, is much less expensive than full multiplication but achieves the same result.

The specifics of strength reduction are often processor-dependent. As an example, the x86 architecture includes the `lea` instruction, which multiplies one value by a constant, then adds the result to another value. This instruction was originally meant for computing the address of elements in an array, but works equally well for computation; in many versions of the x86 architecture, the time to execute an `lea` instruction is less than the time required to multiply a value by a constant and complete an add instruction [1].

2.4.2 Software Profilers

Once the benefit of compiler optimizations has been exhausted, developers are left to manually change code in the hope of improving performance. These changes often require higher-level decisions such as which algorithm to use, how to design a file for quick access, and other application-specific factors that cannot be managed automatically. These changes are often invasive, requiring significant intellectual and implementation effort; it is infeasible for developers to try changing every piece of a large system, so they rely on software profilers to identify important code. Conventional profilers are generally implemented using instrumentation, sampling, or a combination of both.

2.4.2.1 Instrumentation-based Profilers

Instrumentation-based profilers monitor a program's execution by inserting tracking code using the compiler. When the program is run, this monitoring code records the behavior of the program and the time between events. The collected information often includes the time required to execute a function and the number of times it was called. One issue with instrumentation-based profilers is *probe effect*; the time to execute tracking code distorts the program's runtime. If probe effect is particularly large, the profiler ends up measuring performance characteristics of the instrumentation rather than the original program. As a result, virtually no profilers use instrumentation exclusively, and many do not use it at all.

2.4.2.2 Sampling-based Profilers

Rather than instrumenting a program and monitoring every step of its execution, sampling-based or statistical profilers periodically pause a running program and record its state. Each time a sampling profiler pauses the program, it determines which function is currently executing. The number of samples in a particular function is approximately equal to the fraction of program runtime spent in that function. Sampling profilers can also examine the program's call stack to break down time spent in a function based on where it was called from. While sampling profilers necessarily admit some amount of error, they are generally more accurate than instrumentation-based profilers because they have very little probe effect. Unfortunately, there is no reliable way to count invocations of a function using this type of sampling, only the time spent in a particular function.

2.4.2.3 Hybrid Profilers

Hybrid profilers use sampling to collect as much information as possible, while relying on instrumentation to gather information that cannot be measured using sampling. The canonical example is gprof, which uses sampling to measure time spent in each function but also inserts instrumentation to count invocations of each caller–callee pair [33]. This additional information is used to construct a *call graph*, which shows the number of times each function was called from each callsite, and how long these invocations took on average. By tracking execution time with sampling, gprof has much lower overhead than it would be if timer code was included in the instrumentation.

2.4.2.4 Interpreting a Software Profile

Profilers typically measure two quantities: the amount of time spent in a particular function, and the number of times that fragment was executed. When every piece of a program is run in sequence, this information leads developers to good targets for optimizations. If a function contributes very little to total execution time, then optimizing it will not have a large effect on total runtime. Conversely, the number of times a function is executed serves to multiply the benefits of a small performance improvement; reducing the runtime for a function that runs one million times by just 3.6 milliseconds will reduce total runtime by one hour. Unfortunately, more recent architecture, software designs, and usage models make conventional profilers much less useful.

2.4.2.5 Extensions of Software Profiling

Much of the work on software profilers has focused on collecting as much information as possible with minimal probe effect. A sampling-based profiler that pauses execution after some number of clock ticks will collect a number of samples roughly proportional to the execution time spent in a function. Likewise, pausing a program after some number of other events can roughly determine what fraction of those events occurred in a particular function. For example, sampling every n memory accesses allows a developer to see approximately what fraction of total memory accesses are issued by each function. Sampling on a variety of hardware events is available in both the perf and oprofile sampling-based profilers [46, 51]. While these approaches sometimes show surprising performance results, it generally not clear how to reduce the number of memory accesses or some other performance-degrading hardware event in a particular function.

Other profiling techniques have been developed to monitor time programs spend waiting for inputs or computation, to identify code that is amenable to parallelization, or to trace program paths; see Chapter 7 for a full discussion of other profiling techniques.

CHAPTER 3

CHALLENGES IN PERFORMANCE ANALYSIS AND DEBUGGING

Performance analysis and debugging remains a significant challenge, despite decades of work on measuring, understanding, and improving software performance. Modern processors designs often lead to good performance, but “unlucky” inputs or code changes can lead to significant performance regressions. As a result, performance is highly unpredictable and thus difficult to measure accurately. Software profilers are designed to help developers identify code where manual performance tuning would improve whole-program performance, but existing profilers are not well-suited to concurrent software or modern performance concerns. Scalability is a primary concern in software performance; developers must evaluate system in a test environment, but the load conditions in test may not expose scalability bottlenecks that will show up in deployment. Testing systems under realistic load is often impractical or impossible. Just as profilers do not lead developers to code that is important to end-to-end performance, existing approaches for evaluating scalability do not pinpoint code where optimizations will improve scalability. This chapter discusses each of these challenges in depth.

3.1 Unpredictable Performance

Software performance is inherently unpredictable. While the entire stack from hardware up to high-level programming languages is man-made, these systems are now sufficiently complex that most efforts to predict or understand performance analytically are intractable. While existing empirical approaches to performance evaluation work well for a given program, input, and execution environment, small changes to any of these variables can have large, unpredictable effects on performance. This performance unpredictability makes it difficult to deploy software that consistently meets performance requirements, and complicates the task of performance evaluation.

3.1.1 Causes of Performance Unpredictability

Memory density has grown rapidly, but the access times for memory have not kept pace with improvements in processor speed. To reduce the latency of memory accesses, processors keep a copy of recently accessed memory in smaller, faster *cache memories* on the processor. The policy of keeping recently-accessed memory in cache is based on a simple heuristic: programs tend to reuse the same memory repeatedly, so recently accessed memory is likely be needed again soon. When they work well, caches can improve memory system performance by at least an order of magnitude.

One major downside of caches is that they lead to performance unpredictability. A cache with space for one million bytes of recently-accessed memory will dramatically improve the performance of memory accesses unless the program accesses its 1,000,001th byte. At this point, there is no open spot in the cache to store the 1,000,001th byte, so another entry must be *evicted* to make space. If the next byte the program accesses happens to be the one that was just evicted, the cache is no longer effective: every access will have to go to memory rather than finding the value in the cache. Most processor caches use an LRU (least-recently used) policy for eviction, where the oldest entry is removed from the cache. This policy can lead to exactly the problem above, called “cache thrashing”, where a memory access pattern causes the cache to evict memory just before it is needed again.

This situation is made worse by the organization of real caches; engineering constraints dictate that a cache cannot simply be a bucket for the million most-recently accessed bytes. Instead, the cache is divided into *sets* and *ways*. The location of a byte in main memory determines which set it can be cached in, and each set has some number of ways that can hold copies of memory assigned to the same set. Four-way set-associative caches are a common model, where each cache set has space for four recently accessed memory locations. An access pattern that uses five locations that map to the same set before repeating a memory access will experience thrashing. As a result, the history of memory accesses made by an application determine its memory performance. A small change in this access pattern can make cache performance go from a near-perfect hit rate to thrashing, causing a significant performance degradation.

A key concept in the design of modern computers is the *stored program* model. Both the instructions and data required to execute a program reside in a computer’s memory. In the same way that the performance of data accesses depends on the program’s memory access history, accesses to instructions depend on the execution history of the program. Both data and instruction accesses are input-dependent, so any effort to predict access patterns before running a program is not possible except in very limited situations.

Modern processors use a similar mechanism to hide the latency of branch instructions, which programs use to conditionally execute statements. When the result of a branch instruction depends on a computation that has not yet completed—such as one that depends on a memory access—a *branch predictor* can guess whether the branch is likely to be taken and speculatively begin executing instructions at the branch destination. When a branch predictor is incorrect, the results of any speculatively executed instructions must be discarded and the processor resumes execution at the correct location. Branch predictors use the location of a branch instruction in memory to track whether the branch is usually taken or not. In the same way that memory is assigned to cache sets, branch instructions are assigned to branch predictor table entries using their locations in memory. When two branches map to the same branch predictor table entry they can conflict; if one branch is usually taken while the other is not, this leads to a condition known as *branch aliasing*. Branch aliasing causes more branch mispredictions, which can more than offset the benefits of accurate branch prediction elsewhere in the software. Predicting the branching behavior of a program is as challenging as predicting its access patterns, so eliminating branch aliasing not practical, even for simple programs with only a handful of branches.

3.1.2 Challenges for Performance Evaluation

On its face, performance evaluation seems straightforward: run two versions of a program several times, then use confidence intervals or a hypothesis test to decide which version is faster. Unfortunately, even small changes to a program's code or runtime environment will change the program's memory layout. Changing program layout changes which program objects conflict in the cache, with unpredictable performance effects. Performance unpredictability is not limited to random variations between runs; changing a program's code, inputs, or execution environment will change its layout in memory, leading to a consistent change in performance. Repeated runs of the program do not control for this bias because each run uses the same layout.

The impact of these layout changes is unpredictable and substantial. Mytkowicz, Diwan, et al. evaluate the effect of two program changes impact program layout: changing the size of shell environment variables, and changing program link order after compilation [62]. Both changes have no effect on programs' semantics, but result in substantial performance swings; they find that changing the size of environment variables can degrade performance by as much as 300%, and changing link order can hurt performance by as much as 57%.

Conventional performance evaluation strategies do not fix program layout, nor do they explore the space of possible program layouts [30]. This leads to a problem with experimental design. A conventional performance evaluation has two independent variables: the code changes, and the unintended layout changes. When execution time changes, the evaluator cannot determine which variable is responsible: is it because of changes to the code or the collateral impact on layout? Without this information, developers may make poor performance engineering decisions.

3.1.3 Challenges for Software Deployment

Modern software systems are often built using a service-oriented approach. Rather than running from start to finish, a software service runs continuously, constantly accepting and responding to requests. These services often must provide performance guarantees to be useful; they must be able to handle a large number of concurrent requests while still returning results within a "reasonable" amount of time. As an example, Netflix's video streaming system is built from over 100 such services, all of which must meet performance guarantees to provide a good user experience [84].

The requirements for a specific service are application-dependent, but performance unpredictability poses a problem for any performance-sensitive service. An application may be meeting performance requirements in both test and deployment, but any change to the operating system, libraries, hardware, or the code for the services themselves can lead to unpredictable changes in performance. Developers are currently unable to predict when a performance disruption may occur, or to insulate their software from wild performance swings.

One extreme case of software performance requirements is in hard real-time systems. Airplanes, cars, satellites, and many other devices are now controlled by software. This software must continuously monitor the device and its environment and, critically, respond to changes in a timely manner to avoid significant financial losses, injury, or death.

The real-time systems community has recognized the inherent unpredictability in software performance on modern hardware. As a result, real-time systems often run on older processor architectures, or on hardware with caches, branch predictors, and other complex mechanisms disabled. Rather than accept the risk of brittle performance, developers of real-time systems opt for significantly worse performance and scale back on software capabilities to accommodate this loss in performance. This allows developers to safely evaluate whether their software meets performance requirements without fear of brittle performance; with caches disabled, these systems nearly always run at their worst-case performance. However, growing demands on real-time systems—most notably in automotive control software—provide significant pressure for real-time systems developers to adopt newer processors with its performance-improving complexity intact [19].

3.2 Limitations of Software Profilers

Manually inspecting a program to find optimization opportunities is impractical, so developers use profilers to decide which code to optimize. These tools play an important role in performance analysis and debugging, but they are not well-suited to modern hardware and workloads.

3.2.1 Concurrency

Conventional profilers rank code by its contribution to total execution time. Prominent examples include oprofile, perf, and gprof [51, 46, 33]. Unfortunately, even when a profiler accurately reports where a program spends its time, this information can lead programmers astray. The problem is a mismatch between the question that current profilers answer—*where does the program spend its time?*—and the question programmers want answered: *where should I focus my optimization efforts?* Code that runs for a long time is not necessarily a good choice for optimization. For example, optimizing code that draws a loading animation during a file download will not make the program run faster, even though this code runs just as long as the download.

This phenomenon is not limited to I/O operations. Figure 3.1 shows a simple program that illustrates the shortcomings of existing profilers. This program spawns two threads, which invoke functions f_a and f_b respectively. The gprof output for this program is shown in Figure 3.2. Other profilers may report that f_a is on the critical path, or that the main thread spends roughly equal time waiting for f_a and f_b [41]. While accurate, all of this information is potentially misleading. Optimizing f_a away entirely will only speed up the program by 4.5% because f_b becomes the new critical path.

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

Figure 3.1: A simple multithreaded program that illustrates the shortcomings of existing profilers. Optimizing f_a will improve performance by no more than 4.5%, while optimizing f_b would have no effect on performance.

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

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

Figure 3.2: A gprof profile for the program in Figure 3.1. This profile shows that f_a and f_b comprise similar fractions of total runtime. While true, this is misleading: optimizing f_a will improve performance by at most 4.5%, and optimizing f_b would have no effect on performance.

Conventional profilers do not report the potential impact of optimizations; developers are left to make these predictions based on their understanding of the program. While these predictions may be easy for programs as simple as the one in Figure 3.1, accurately predicting the effect of a proposed optimization is nearly impossible for programmers attempting to optimize large applications.

3.2.2 Throughput and Latency

Even if the output of conventional profilers was a useful guide for reducing end-to-end runtimes, profilers would be of limited use for modern software. End-to-end runtime is a proxy for an application's throughput; the faster it can complete its execution, the greater its throughput. However, the growth in service-oriented software means that latency is an equally important measure of performance in many systems. A service that produces a million responses per second but takes ten minutes between request and response is not useful.

Unfortunately, conventional profilers are not easily adapted to deal with latency. Because they simply report where a program spends its time, they do not distinguish between time spend processing a transaction and other work done by the system. The picture of a program's execution provided by a conventional profiler includes portions of many transactions, but is often dominated by waiting: even a heavily loaded service often spends much of its time waiting for requests to arrive. To make effective use of a conventional profile, developers of latency-sensitive services must have a near-perfect mental model of their system's behavior to determine which code is important and which code executes off the critical path for a transaction. While this may be feasible for small programs, developers' mental models of their systems are often flawed, and these flaws in program understanding can obscure performance bugs [67].

3.3 Building Scalable Software

Service-oriented software poses a special challenge for performance evaluation. These programs are designed to handle heavy load—much more than a single machine can generate—but the performance of such a system is highly dependent on the amount and type of load on the system. Like a multi-user system, a software service will process requests from different clients concurrently. However, unlike the multi-user system, these requests are often not independent. An example of one service is the Memcached caching server.

Memcached is a widely-used key-value store typically used as a caching layer in front of a database. Clients of a memcached server can issue `PUT` requests to insert a new value into the cache, associated with a specified key. A `GET` request searches for a given key and returns the associated value, if it exists. Because requests from different clients can potentially read and write the same data, access to the set of stored values must be controlled via synchronization. If multiple clients are issuing `GET` requests for a given key, a read-write lock would allow these concurrent requests to safely execute in parallel. A `PUT` request, however, cannot proceed until all readers have released the lock. If load on the server is sufficiently high, the `PUT` request may never successfully acquire an exclusive lock on the entry to update the value, a situation known as "writer starvation."

Resolving this issue could be as straightforward as switching to a different synchronization primitive, such as one that gives writers priority ahead of any later read requests. However, developers are unlikely to fix this issue if they are never able to generate sufficient load on a memcached in testing to expose the issue. Successfully triggering this issue requires that the test environment is able to generate sufficient load to induce contention, and that the load is representative of real executions. Unfortunately, for a scalable service, a single machine *should not* be able to generate enough load to saturate the server; if this were possible, the service would not be scalable.

Even if generating enough load was possible, developers would need enormous traces of all requests to replay in order to generate representative load. These traces are required because issuing random `PUT` and `GET` requests would not trigger the scalability issue; it will only occur when many clients issue concurrent `GET` requests for the same key.

One potential workaround for this issue is to embed performance measurement code into deployed services, but this measurement code may have overhead that is unacceptable for deployment. Even if the overhead of performance monitoring is low enough to run in deployment, the results of this performance monitoring will have the same drawbacks as output from conventional profilers; the data tell developers how much time was spent in each part of the program, not which parts of the program to optimize to improve performance in deployment.

CHAPTER 4

STABILIZER: PREDICTABLE AND ANALYZABLE PERFORMANCE

The task of performance evaluation forms a key part of both systems research and the software development process. Researchers working on systems ranging from compiler optimizations and runtime systems to code transformation frameworks and bug detectors must measure their effect, evaluating how much they improve performance or how much overhead they impose [16, 15]. Software developers need to ensure that new or modified code either in fact yields the desired performance improvement, or at least does not cause a performance regression (that is, making the system run slower). For large systems in both the open-source community (e.g., Firefox and Chromium) and in industry, automatic performance regression tests are now a standard part of the build or release process [78, 75].

In both settings, performance evaluation typically proceeds by testing the performance of the actual application in a set of scenarios, or a range of benchmarks, both before and after applying changes or in the absence and presence of a new optimization, runtime system, etc.

In addition to measuring *effect size* (here, the magnitude of change in performance), a statistically sound evaluation must test whether it is possible with a high degree of confidence to reject the *null hypothesis*: that the performance of the new version is indistinguishable from the old. To show that a performance optimization is statistically significant, we need to reject the null hypothesis with high confidence (and show that the direction of improvement is positive). Conversely, we aim to show that it is not possible to reject the null hypothesis when we are testing for a performance regression.

Unfortunately, even when using current best practices (large numbers of runs and a quiescent system), the conventional approach is unsound. The problem is due to the interaction between software and modern architectural features, especially caches and branch predictors. These features are sensitive to the addresses of the objects they manage. Because of the significant performance penalties imposed by cache misses or branch mispredictions (e.g., due to aliasing), their reliance on addresses makes software exquisitely sensitive to memory layout. Small changes to code, such as adding or removing a stack variable, or changing the order of heap allocations, can have a ripple effect that alters the placement of every other function, stack frame, and heap object.

The impact of these layout changes is unpredictable and substantial: Mytkowicz et al. show that just changing the size of environment variables can trigger performance degradation as high as 300% [62]; we find that simply changing the link order of object files can cause performance to decrease by as much as 57%.

Failure to control for layout is a form of *measurement bias*: a systematic error due to uncontrolled factors. All executions constitute just one sample from the vast space of possible memory layouts. This limited sampling makes statistical tests inapplicable, since they depend on multiple samples over a space, often with a known distribution. As a result, it is currently not possible to test whether a code modification is the direct cause of any observed performance change, or if it is due to incidental effects like a different code, stack, or heap layout.

4.0.1 Contributions

This chapter presents STABILIZER, a system that enables statistically sound performance analysis of software on modern architectures. To our knowledge, STABILIZER is the first system of its kind.

STABILIZER forces executions to sample over the space of all memory configurations by efficiently and repeatedly randomizing the placement of code, stack, and heap objects at runtime. We show analytically and empirically that STABILIZER's use of randomization makes program execution independent of the execution environment, and thus eliminates this source of measurement bias. Re-randomization goes one step further: it causes the performance impact of layout effects to follow a Gaussian (normal) distribution, by virtue of the Central Limit Theorem. In many cases, layout effects dwarf all other sources of execution time variance [62]. As a result, STABILIZER often leads to execution times that are normally distributed.

By generating execution times with Gaussian distributions, STABILIZER enables statistically sound performance analysis via parametric statistical tests like ANOVA [27]. STABILIZER thus provides a push-button solution that allows developers and researchers to answer the question: does a given change to a program affect its performance, or is this effect indistinguishable from noise?

We demonstrate STABILIZER's efficiency ($< 7\%$ median overhead) and its effectiveness by evaluating the impact of LLVM's optimizations on the SPEC CPU2006 benchmark suite. Across the

SPEC CPU2006 benchmark suite, we find that the `-O3` compiler switch (which includes argument promotion, dead global elimination, global common subexpression elimination, and scalar replacement of aggregates) does not yield statistically significant improvements over `-O2`. In other words, the effect of `-O3` versus `-O2` is indistinguishable from random noise.

We note in passing that STABILIZER's low overhead means that it could be used at deployment time to reduce the risk of performance outliers, although we do not explore that use case here. Intuitively, STABILIZER makes it unlikely that object and code layouts will be especially "lucky" or "unlucky." By periodically re-randomizing, STABILIZER limits the contribution of each layout to total execution time.

4.1 Overview

This section provides an overview of STABILIZER's operation, and how it provides properties that enable statistically rigorous performance evaluation.

4.1.1 Comprehensive Layout Randomization

STABILIZER dynamically randomizes program layout to ensure it is independent of changes to code, compilation, or execution environment. STABILIZER performs extensive randomization: it dynamically randomizes the placement of a program's functions, stack frames, and heap objects. Code is randomized at a per-function granularity, and each function executes on a randomly placed stack frame. STABILIZER also periodically *re-randomizes* the placement of functions and stack frames during execution.

4.1.2 Normally Distributed Execution Time

When a program is run with STABILIZER, the effect of memory layout on performance follows a normal distribution because of layout re-randomization. Layout effects make a substantial contribution to a program's execution. In the absence of other large sources of measurement bias, STABILIZER causes programs to run with normally distribution execution times.

At a high level, STABILIZER's re-randomization strategy induces normally distributed executions as follows: Each random layout contributes a small fraction of total execution time. Total execution time, the sum of runtimes with each random layout, is proportional to the mean of sampled layouts.

The *Central Limit Theorem* states that "the mean of a sufficiently large number of independent random variables . . . will be approximately normally distributed" [27]. With a sufficient number of randomizations (30 is typical), and no other significant sources of measurement bias, execution time will follow a Gaussian distribution. Section 4.3 provides a more detailed analysis of STABILIZER's effect on execution time distributions.

4.1.3 Sound Performance Analysis

Normally distributed execution times allow researchers to evaluate performance using *parametric* hypothesis tests, which provide greater *statistical power* by leveraging the properties of a known distribution (typically the normal distribution). Statistical power is the probability of correctly rejecting a false null hypothesis. Parametric tests typically have greater power than non-parametric tests, which make no assumptions about distribution. For our purposes, the null hypothesis is that a change had no impact. Failure to reject the null hypothesis suggests that more samples (benchmarks or runs) may be required to reach confidence, or that the change had no impact. Powerful parametric tests can correctly reject a false null hypothesis—that is, confirm that a change did have an impact—with fewer samples than non-parametric tests.

4.1.4 Evaluating Code Modifications

To test the effectiveness of any change (known in statistical parlance as a *treatment*), a researcher or developer runs a program with STABILIZER, both with and without the change. Each run is a sample from the treatment's *population*: the theoretical distribution from which samples are drawn. Given that execution times are drawn from a normally distributed population, we can apply the Student's t-test [27] to calculate the significance of the treatment.

The null hypothesis for the t-test is that the difference in means of the source distributions is zero. The t-test's result (its *p-value*) tells us the probability of observing the measured difference between sample means, assuming both sets of samples come from the same source distribution. If the p-value is below a threshold α (typically 5%), the null hypothesis is rejected; that is, the two source distributions have different means. The parameter α is the probability of committing a type-I error: erroneously rejecting a true null hypothesis.

It is important to note that the t-test can detect *arbitrarily small* differences in the means of two populations (given a sufficient number of samples) regardless of the value of α. The difference in means does not need to be 5% to reach significance with $\alpha = 0.05$. Similarly, if STABILIZER adds 4.8% overhead to a program, this does not prevent the t-test from detecting differences in means that are smaller than 4.8%.

4.1.5 Evaluating Compiler and Runtime Optimizations

To evaluate a compiler or runtime system change, we instead use a more general technique: analysis of variance (ANOVA). ANOVA takes as input a set of results for each combination of benchmark and treatment, and partitions the total variance into components: the effect of random variations between runs, differences between benchmarks, and the collective impact of each treatment across all benchmarks [27]. ANOVA is a generalized form of the t-test that is less likely to commit type I errors (rejecting a true null hypothesis) than running many independent t-tests. Section 4.5 presents the use of STABILIZER and ANOVA to evaluate the effectiveness of compiler optimizations in LLVM.

4.1.6 Evaluating Optimizations Targeting Layout

All of STABILIZER's randomizations (code, stack, and heap) can be enabled independently. This independence makes it possible to evaluate optimizations that target memory layout. For example, to test an optimization for stack layouts, STABILIZER can be run with only code and heap randomization enabled. These randomizations ensure that incidental changes, such as code to pad the stack or to allocate large objects on the heap, will not affect the layout of code or heap memory. The developer can then be confident that any observed change in performance is the result of the stack optimization and not its secondary effects on layout.

4.2 Implementation

STABILIZER uses a compiler transformation and runtime library to randomize program layout. STABILIZER performs its transformations in an optimization pass run by the LLVM compiler [50]. STABILIZER's compiler transformation inserts the necessary operations to move the stack, redirects heap operations to the randomized heap, and modifies functions to be independently relocatable.

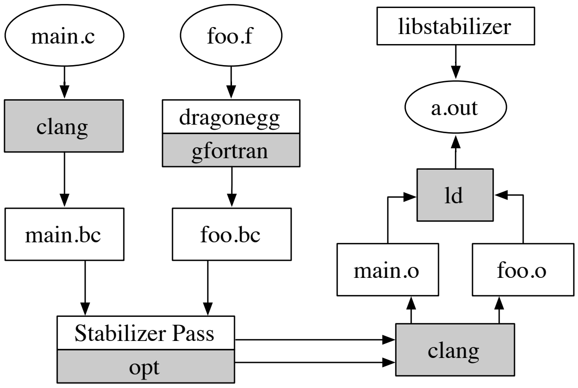

Figure 4.1: The procedure for building a program with STABILIZER. This process is automated by the `szc` compiler driver.

STABILIZER's runtime library exposes an API for the randomized heap, relocates functions on-demand, generates random padding for the stack, and re-randomizes both code and stack at regular intervals.

4.2.1 Building Programs with Stabilizer

When building a program with STABILIZER, each source file is first compiled to LLVM bytecode. STABILIZER builds Fortran programs with `gfortran` and the `dragonegg` GCC plugin, which generates LLVM bytecode from the GCC front-end [77]. C and C++ programs can be built either with `gcc` and `dragonegg`, or LLVM's `clang` front-end [76].

The compilation and transformation process is shown in Figure 4.1. This procedure is completely automated by STABILIZER's compiler driver (`szc`), which is compatible with the common `clang` and `gcc` command-line options. Programs can easily be built and evaluated with STABILIZER by substituting `szc` for the default compiler/linker and enabling randomizations with additional flags.

4.2.2 Heap Randomization

STABILIZER uses a power of two, size-segregated allocator as the base for its heap [86]. Optionally, STABILIZER can be configured to use TLSF (two-level segregated fits) as its base allocator [55]. STABILIZER was originally implemented with the DieHard allocator [11, 66]. DieHard is a bitmap-based randomized allocator with power-of-two size classes. Unlike conventional allocators, DieHard does not use recently-freed memory for subsequent allocations. This lack of reuse and the added TLB pressure from the large virtual address space can lead to very high overhead.

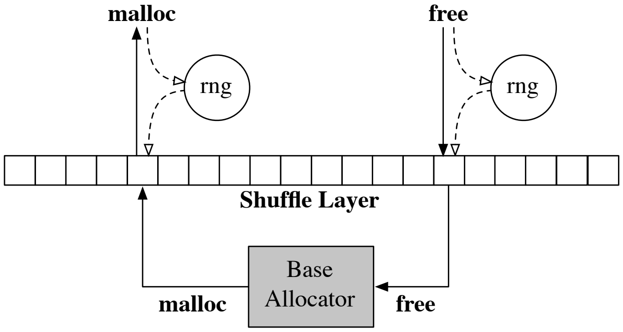

Figure 4.2: An illustration of STABILIZER's heap randomization. STABILIZER efficiently randomizes the heap by wrapping a deterministic base allocator in a shuffling layer. At startup, the layer is filled with objects from the base heap. The `malloc` function generates a random index, removes the indexed object from the shuffling layer, and replaces it with a new one from the base heap. Similarly, the `free` function generates a random index, frees the indexed object to the base heap, and places the newly freed object in its place.

While STABILIZER's base allocators are more efficient than DieHard, they are not fully randomized. STABILIZER randomizes the heap by wrapping its base allocator in a shuffling layer built with HeapLayers [12]. The shuffling layer consists of a size N array of pointers for each size class. The array for each size class is initialized with a fill: N calls to `Base::malloc` are issued to fill the array, then the array is shuffled using the Fisher-Yates shuffle [26]. Every call to `Shuffle::malloc` allocates a new object `p` from `Base::malloc`, generates a random index `i` in the range $[0, N)$, swaps `p` with `array[i]`, and returns the swapped pointer. `Shuffle::free` works in much the same way: a random index `i` is generated, the freed pointer is swapped with `array[i]`, and the swapped pointer is passed to `Base::free`. The process for `malloc` and `free` is equivalent to one iteration of the inside-out Fisher-Yates shuffle. Figure 4.2 illustrates this procedure. STABILIZER uses the Marsaglia pseudo-random number generator from DieHard [54, 11].

The shuffled heap parameter N must be large enough to create sufficient randomization, but values that are too large will increase overhead with no added benefit. It is only necessary to randomize the index bits of heap object addresses. Randomness in lower-order bits will lead to misaligned allocations, and randomized higher order bits impose additional pressure on the TLB. NIST provides a standard statistical test suite for evaluation pseudorandom number generators [10].

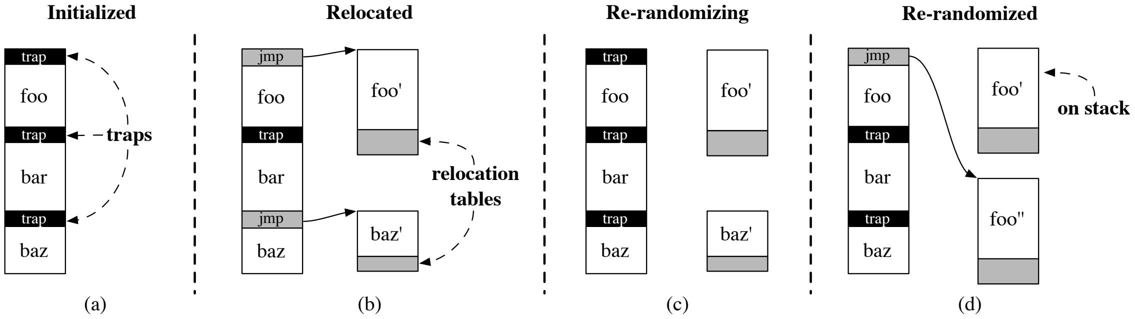

Figure 4.3: An illustration of STABILIZER's code randomization. *(a)* During initialization, STABILIZER places a trap instruction at the beginning of each function. When a trapped function is called, it is relocated on demand. *(b)* Each randomized function has an adjacent relocation table, populated with pointers to all referenced globals and functions. *(c)* A timer triggers periodic re-randomizations (every 500ms by default). In the timer signal handler, STABILIZER places traps at the beginning of every randomized function. *(d)* Once a trapped function is called, STABILIZER walks the stack, marks all functions with return addresses on the stack, and frees the rest; `baz`′ is freed in this example because there are no live references to this function location. Any remaining functions (`foo`′) will be freed after a future re-randomization once they are no longer on the stack. Future calls to `foo` will be directed to a new, randomly located version (`foo`″).

We test the randomness of values returned by libc's `lrand48` function, addresses returned by the DieHard allocator, and the shuffled heap for a range of values of N. Only the index bits (bits 6-17 on the Core2 architecture) were used. Bits used by branch predictors differ significantly across architectures, but are typically low-order bits generally in the same range as cache index bits.

The `lrand48` function passes six tests for randomness (Frequency, BlockFrequency, CumulativeSums, Runs, LongestRun, and FFT) with $> 95\%$ confidence, failing only the Rank test. DieHard passes these same six tests. STABILIZER's randomized heap passes the same tests with the shuffling parameter $N = 256$. STABILIZER uses this heap configuration to randomly allocate memory for both heap objects and functions.

4.2.3 Code Randomization

STABILIZER randomizes code at function granularity. Every transformed function has a *relocation table* (see Figure 4.3(b)), which is placed immediately following the code for the function. Functions are placed randomly in memory using a separate randomized heap that allocates executable memory.

Relocation tables are not present in a binary built with STABILIZER. Instead, they are created at runtime immediately following each randomly located function. The sizes of functions are not available in the program's symbol table, so the address of the next function is used to determine the function's endpoint. A function refers to its adjacent relocation table with a PC-relative offset. This approach means that two randomly located copies of the same function do not share a relocation table.

Some constant floating point operands are converted to global variable references during code generation. STABILIZER converts all non-zero floating point constants to global variables in the IR so accesses can be made indirect through the relocation table.

Operations that convert between floating-point and integers do not contain constant operands, but still generate implicit global references during code generation. STABILIZER cannot rewrite these references. Instead, STABILIZER adds functions to each module to perform int-to-float and float-to-int conversions, and replaces the LLVM `fptosi`, `fptoui`, `sitofp`, and `uitofp` instructions with calls to these conversion functions. The conversion functions are the only code that STABILIZER cannot safely relocate.

Finally, STABILIZER renames the `main` function. The STABILIZER runtime library defines its own main function, which initializes runtime support for code randomization before executing any randomized code.

4.2.3.1 Initialization.

At compile time, STABILIZER replaces the module's libc constructors with its own constructor function. At startup, this constructor registers the module's functions and any constructors from the original program. Execution of the program's constructors is delayed until after initialization.

The `main` function, defined in STABILIZER's runtime, overwrites the beginning of every relocatable function with a software breakpoint (the `int 3` x86 instruction, or `0xCC` in hex); see Figure 4.3(a). A pointer to the function's runtime object is placed immediately after the trap to allow for immediate relocation (not shown).

4.2.3.2 Relocation.

When a trapped function is executed, the STABILIZER runtime receives a `SIGTRAP` signal and relocates the function (Figure 4.3(b)). Functions are relocated in three stages: first, STABILIZER requests a sufficiently large block of memory from the code heap and copies the function body to this location. Next, the function's relocation table is constructed next to the new function location. STABILIZER overwrites the beginning of the function's original base address with a static jump to the relocated function (replacing the trap instruction). Finally, STABILIZER adds the function to the set of "live" functions.

4.2.3.3 Re-randomization.

STABILIZER re-randomizes functions at regular time intervals (500ms by default). When the re-randomization timer expires, the STABILIZER runtime places a trap instruction at the beginning of every live function and resumes execution (Figure 4.3(c)). Re-randomization occurs when the next trap is executed. This delay ensures that re-randomization will not be performed during the execution of non-reentrant code.

STABILIZER uses a simple garbage collector to reclaim memory used by randomized functions. First, STABILIZER adds the memory used by each live functions to a set called the "pile." STABILIZER then walks the stack. Every object on the pile pointed to by a return address on the stack is marked. All unmarked objects on the pile are freed to the code heap.

4.2.4 Stack Randomization.

STABILIZER randomizes the stack by adding a random amount of space (up to 4096 bytes) between each stack frame. STABILIZER's compiler pass creates a 256 byte stack pad table and a one-byte stack pad index for each function. On entry, the function loads the index-th byte, increments the index, and multiplies the byte by 16 (the required stack alignment on `x86_64`). STABILIZER moves the stack down by this amount prior to each function call, and restores the stack after the call returns.

The STABILIZER runtime fills every function's stack pad table with random bytes during each re-randomization. The stack pad index may overflow, wrapping back around to the first entry. This wraparound means functions may reuse a random stack pad several times between re-randomizations,

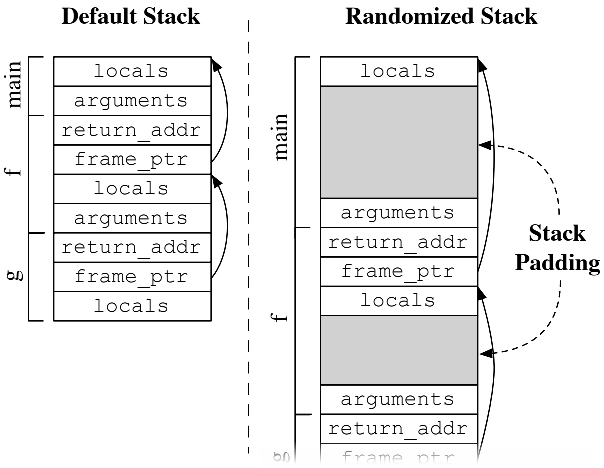

Figure 4.4: An illustration of STABILIZER's stack randomization. STABILIZER randomizes the stack by adding up to a page of random pad at each function call. Functions are instrumented to load a pad size from the stack pad table. STABILIZER periodically refills this table with new random values to re-randomize the stack.

but the precise location of the stack is determined by the stack pad size used for each function on the call stack. This combination ensures that stack placement is sufficiently randomized.

4.2.5 Architecture-Specific Implementation Details

STABILIZER runs on the x86, x86_64 and PowerPC architectures. Most implementation details are identical, but STABILIZER requires some platform-specific support.

4.2.5.1 STABILIZER on 64-bit x86

Supporting the x86_64 architecture introduces two complications for STABILIZER. The first is for the jump instructions: jumps, whether absolute or relative, can only be encoded with a 32-bit address (or offset). STABILIZER uses `mmap` with the `MAP_32BIT` flag to request memory for relocating functions, but on some systems (Mac OS X), this flag is unavailable.

To handle cases where functions must be relocated more than a 32-bit offset away from the original copy, STABILIZER simulates a 64-bit jump by pushing the target address onto the stack and issuing a return instruction. This form of jump is much slower than a 32-bit relative jump, so high-address memory is only used after low-address memory is exhausted.

4.2.5.2 STABILIZER on PowerPC and 32-bit x86

PowerPC and x86 both use PC-relative addressing for control flow, but data is accessed using absolute addresses. Because of this, the relocation table must be at a fixed absolute address rather than adjacent to a randomized function. The relocation table is only used for function calls, and does not need to be used for accesses to global data.

4.3 Statistical Analysis

This section presents an analysis that explains how STABILIZER's randomization results in normally distributed execution times for most programs. Section 4.4 empirically verifies this analysis across our benchmark suite.

The analysis proceeds by first considering programs with a reasonably trivial structure (running in a single loop), and successively weakens this constraint to handle increasingly complex programs.

We assume that STABILIZER is only used on programs that consist of more than a single function. Because STABILIZER performs code layout randomization on a per-function basis, the location of code in a program consisting of just a single function will not be re-randomized. Since most programs consist of a large number of functions, we do not expect this to be a problem in practice.

4.3.1 Base case: a single loop

Consider a small program that runs repeatedly in a loop, executing at least one function. The space of all possible layouts l for this program is the population L. For each layout, an iteration of the loop will have an execution time t. The population of all iteration execution times is E. Clearly, running the program with layout l for 1000 iterations will take time:

$$T_{random} = 1000 * t$$

For simplicity, assume that when this same program is run with STABILIZER, each iteration is run with a different layout l_i with execution time t_i (we refine the notion of "iteration" below).

Running this program with STABILIZER for 1000 iterations will thus have total execution time:

$$T_{stabilized} = \sum_{i=1}^{1000} t_i$$

The values of t_i comprise a sample set x from the population E with mean:

$$\bar{x} = \sum_{i=1}^{1000} \frac{t_i}{1000}$$

The central limit theorem tells us that $\bar{x}$ must be normally distributed (30 samples is typically sufficient for normality). The value of $\bar{x}$ only differs from $T_{stabilized}$ by a constant factor. Multiplying a normally distributed random variable by a constant factor simply shifts and scales the distribution. The result remains normally distributed. Therefore, for this simple program, STABILIZER leads to normally distributed execution times. Note that the distribution of E was never mentioned—the central limit theorem guarantees normality regardless of the sampled population's distribution.

The above argument relies on two conditions. The first is that STABILIZER runs each iteration with a different layout. STABILIZER actually uses wall clock time to trigger re-randomization, but the analysis still holds. As long as STABILIZER re-randomizes roughly every n iterations, we can simply redefine an "iteration" to be n passes over the same code. The second condition is that the program is simply a loop repeating the same code over and over again.

4.3.2 Programs with phase behavior

In reality, programs have more complex control flow and may even exhibit phase-like behavior. The net effect is that for one randomization period, where STABILIZER maintains the same random layout, one of any number of different portions of the application code could be running. However, the argument still holds.

A complex program can be recursively decomposed into subprograms, eventually consisting of subprograms equivalent to the trivial looping program described earlier. These subprograms will each comprise some fraction of the program's total execution, and will all have normally distributed execution times. The total execution time of the program is thus a weighted sum of all the subprograms. A similar approach is used by SimPoint, which accelerates architecture simulation by drawing representative samples from all of a program's phases [36].

Because the sum of two normally distributed random variables is also normally distributed, the program will still have a normally distributed execution time. This decomposition also covers the case where STABILIZER's re-randomizations are out of phase with the iterations of the trivial looping program.

4.3.3 Heap accesses

Every allocation with STABILIZER returns a randomly selected heap address, but live objects are not relocated because C/C++ semantics do not allow it. STABILIZER thus enforces normality of heap access times as long as the program contains a sufficiently large number of short-lived heap objects (allowing them to be effectively re-randomized). This behavior is common for most applications and corresponds to the generational hypothesis for garbage collection, which has been shown to hold in unmanaged environments [22, 61].

STABILIZER cannot break apart large heap allocations, and cannot add randomization to custom allocators. Programs that use custom allocators or operate mainly small objects from a single large array may not have normally distributed execution times because STABILIZER cannot sufficiently randomize their layout.

4.4 Evaluation

We evaluate STABILIZER in two dimensions. First, we test the claim that STABILIZER causes execution times to follow a Gaussian distribution. Next, we look at the overhead added by STABILIZER with different randomizations enabled.

All evaluations are performed on a dual-core Intel Core i3-550 operating at 3.2GHz and equipped with 12GB of RAM. Each core has a 256KB L2 cache. Cores share a single 4MB L3 cache. The system runs version 3.5.0-22 of the Linux kernel (unmodified) built for `x86_64`. All programs are built using `gcc` version 4.6.3 as a front-end, with `dragonegg` and LLVM version 3.1.

4.4.1 Benchmarks

We evaluate STABILIZER across all C benchmarks in the SPEC CPU2006 benchmark suite. The C++ benchmarks `omnetpp`, `xalancbmk`, `dealII`, `soplex`, and `povray` are not run because they use exceptions, which STABILIZER does not yet support. We plan to add support for exceptions

Benchmark	Shapiro-Wilk Randomized	 Re-randomized	Brown-Forsythe
`astar`	**0.000**	0.194	**0.001**
`bzip2`	0.789	0.143	0.078
`cactusADM`	**0.003**	**0.003**	**0.001**
`gcc`	0.420	0.717	**0.013**
`gobmk`	0.072	0.563	**0.000**
`gromacs`	**0.015**	0.550	**0.022**
`h264ref`	**0.003**	0.183	**0.002**
`hmmer`	0.552	**0.016**	0.982
`lbm`	0.240	0.530	0.161
`libquantum`	0.437	0.115	0.397
`mcf`	0.991	0.598	**0.027**
`milc`	0.367	0.578	0.554
`namd`	0.254	0.691	0.610
`perlbench`	**0.036**	0.188	**0.047**
`sjeng`	0.240	0.373	**0.000**
`sphinx3`	0.727	0.842	0.203
`wrf`	0.856	0.935	0.554
`zeusmp`	0.342	0.815	**0.000**

Table 4.1: Normality and variance results for SPEC CPU2006 benchmarks run with STABILIZER using one-time and repeated randomization. P-values for the Shapiro-Wilk test of normality and the Brown-Forsythe test for homogeneity of variance. A p-value less that $\alpha = 0.05$ is sufficient to reject the null hypothesis (indicated in bold). Shapiro-Wilk tests the null hypothesis that the data are drawn from a normal distribution. Brown-Forsythe tests whether the one-time randomization and re-randomization samples are drawn from distributions with the same variance. Boldface indicates statistically significant non-normal execution times and unequal variances, respectively. Section 4.4.2 explores these results further.

by rewriting LLVM's exception handling intrinsics to invoke STABILIZER-specific runtime support for exceptions. STABILIZER is also evaluated on all Fortran benchmarks, except for `bwaves`, `calculix`, `gamess`, `GemsFDTD`, and `tonto`. These benchmarks fail to build on our system when using `gfortran` with the LLVM plugin.

4.4.2 Normality

We evaluate the claim that STABILIZER results in normally distributed execution times across the entire benchmark suite. Using the Shapiro-Wilk test for normality, we can check if the execution times of each benchmark are normally distributed with and without STABILIZER. Every benchmark is run 30 times each with and without STABILIZER's re-randomization enabled.

Table 4.1 shows the p-values for the Shapiro-Wilk test of normality. Without re-randomization, five benchmarks exhibit execution times that are not normally distributed with 95% confidence: astar, cactusADM, gromacs, h264ref, and perlbench. With re-randomization, all of these benchmarks exhibit normally distributed execution times except for cactusADM. The hmmer benchmark has normally distributed execution times with one-time randomization, but not with re-randomization. This anomaly may be due to hmmer's use of alignment-sensitive floating point operations.

Figure 4.5 shows the distributions of all 18 benchmarks on QQ (quantile-quantile) plots. QQ plots are useful for visualizing how close a set of samples is to a reference distribution (Gaussian in this case). Each data point is placed at the intersection of the sample and reference distributions' quantiles. Points will fall along a straight line if the observed values come from the reference distribution family.

A steeper slope on the QQ plot indicates a greater variance. We test for homogeneity of variance using the Brown-Forsythe test [27]. For eight benchmarks, astar, gcc, gobmk, gromacs, h264ref, perlbench, sjeng, and zeusmp, re-randomization leads to a statistically significant decrease in variance. This decrease is the result of *regression to the mean*. Observing a very high execution time with re-randomization would require selecting many more "unlucky" than "lucky" layouts. In two cases, cactusADM and mcf, re-randomization yields a small but statistically significant increase in variance. The p-values for the Brown-Forsythe test are shown in Table 4.1.

Result: STABILIZER nearly always imposes a Gaussian distribution on execution time, and tends to reduce variance.

4.4.3 Efficiency

Figure 4.6 shows the overhead of STABILIZER relative to unrandomized execution. Every benchmark is run 30 times in each configuration. With all randomizations enabled, STABILIZER adds a median overhead of 6.7%.

Most of STABILIZER's overhead can be attributed to reduced locality. Code and stack randomization both add additional logic to function invocation, but this has limited impact on execution time.

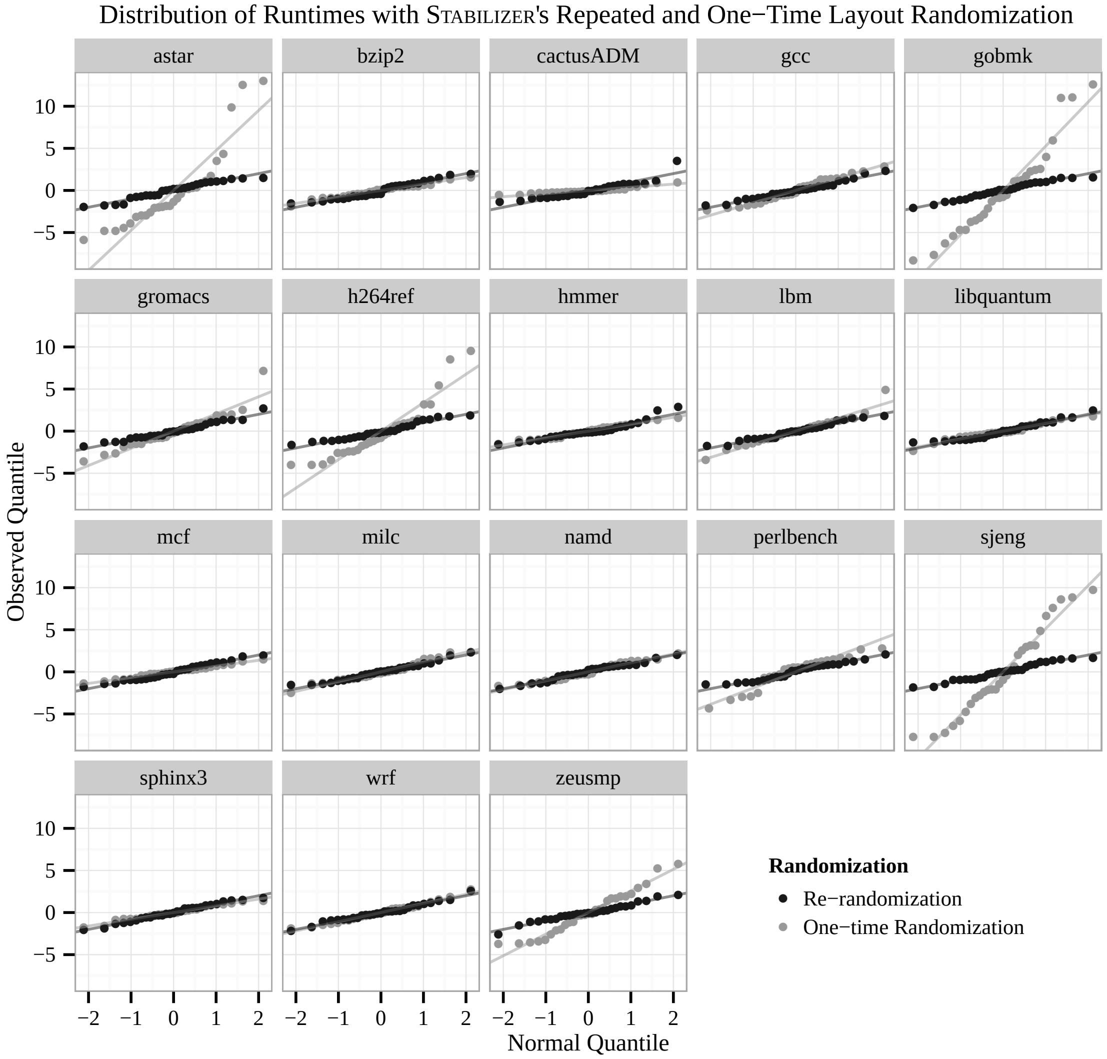

Figure 4.5: Quantile-quantile plots showing the distributions of execution times with STABILIZER. **Gaussian distribution of execution time:** Quantile-quantile plots comparing the distributions of execution times to the Gaussian distribution. Samples are shifted to a mean of zero, and normalized to the standard deviation of the re-randomized samples. Solid diagonal lines show where samples from a Gaussian distribution would fall. Without re-randomization, `astar`, `cactusADM`, `gromacs`, `h264ref`, and `perlbench` have execution times that are not drawn from a Gaussian distribution. With re-randomization, all benchmarks except `cactusADM` and `hmmer` conform to a Gaussian distribution. A steeper slope on the QQ plot indicates a greater variance. See Section 4.4.2 for a discussion of these results.

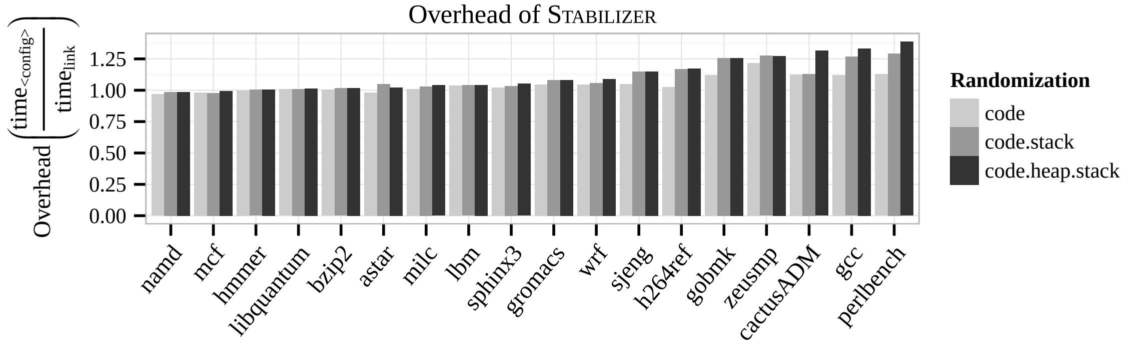

Figure 4.6: Distribution of runtimes with STABILIZER one-time and repeated randomization. Overhead of STABILIZER relative to runs with randomized link order (lower is better). With all randomizations enabled, STABILIZER adds a median overhead of 6.7%, and below 40% for all benchmarks.

Programs run with STABILIZER use a larger portion of the virtual address space, putting additional pressure on the TLB.

With all randomizations enabled, STABILIZER adds more than 30% overhead for just four benchmarks. For `gobmk`, `gcc`, and `perlbench`, the majority of STABILIZER's overhead comes from stack randomization. These three benchmarks all have a large number of functions, each with its own stack pad table (described in Section 4.2).

The increased working set size increases cache pressure. If STABILIZER allowed functions to share stack pad tables, this overhead could be reduced. STABILIZER's heap randomization adds most of the overhead to `cactusADM`. This benchmark allocates a large number of arrays on the heap, and rounding up to power of two size classes leads to a large amount of wasted heap space.

STABILIZER's overhead does not affect its validity as a system for measuring the impact of performance optimizations. If an optimization has a statistically significant impact, it will shift the mean execution time over all possible layouts. The overhead added by STABILIZER also shifts this mean, but applies equally to both versions of the program. STABILIZER imposes a Gaussian distribution on execution times, which enables the detection of *smaller* effects than an evaluation of execution times with unknown distribution.

4.4.3.1 Performance Improvements

In four cases, STABILIZER (slightly) improves performance. `astar`, `hmmer`, `mcf`, and `namd` all run faster with code randomization enabled. We attribute this to the elimination of branch aliasing [42]. It is highly unlikely that a significant fraction of a run's random code layouts would exhibit branch aliasing problems. It is similarly unlikely that a significant fraction of random layouts would result in large performance improvements. The small gains with STABILIZER suggest the default program layout is slightly worse than the median layout for these benchmarks.

4.5 Sound Performance Analysis

The goal of STABILIZER is to enable statistically sound performance evaluation. We demonstrate STABILIZER's use here by evaluating the effectiveness of LLVM's `-O3` and `-O2` optimization levels. Figure 4.7 shows the speedup of `-O2` and `-O3`, where speedup of `-O3` is defined as:

$$\frac{\text{time}_{\texttt{-O2}}}{\text{time}_{\texttt{-O3}}}$$

LLVM's `-O2` optimizations include basic-block level common subexpression elimination, while `-O3` adds argument promotion, global dead code elimination, increases the amount of inlining, and adds global (procedure-wide) common subexpression elimination.

Execution times for all but three benchmarks are normally distributed when run with STABILIZER. These three benchmarks, `hmmer`, `wrf`, and `zeusmp`, have p-values below $\alpha = 0.05$ for the Shapiro-Wilk test. For all benchmarks with normally distributed execution times, we apply the two-sample t-test to determine whether `-O3` provides a statistically significant performance improvement over `-O2`, and likewise for `-O2` over `-O1`. The three non-normal benchmarks use the Wilcoxon signed-rank test, a non-parametric equivalent to the t-test [85].

At a 95% confidence level, we find that there is a statistically significant difference between `-O2` and `-O1` for 17 of 18 benchmarks. There is a significant difference between `-O3` and `-O2` for 9 of 18 benchmarks. While this result is promising, it does come with a caveat: `bzip2`, `libquantum`, and `milc` show a statistically significant *increase* in execution time with `-O2` optimizations. The `bzip2`, `gobmk`, and `zeusmp` benchmarks show a statistically significant performance degradation with `-O3`.

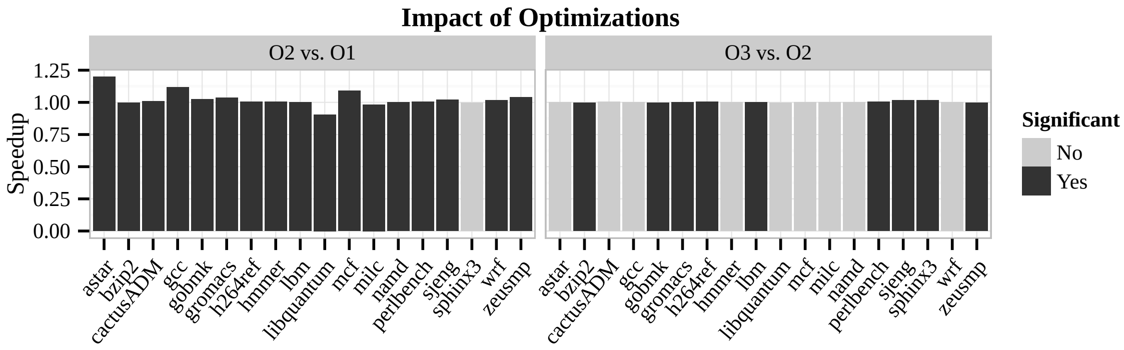

Figure 4.7: Speedup of `-O2` over `-O1`, and `-O3` over `-O2` optimizations in LLVM. A speedup above 1.0 indicates the optimization had a positive effect. Asterisks mark cases where opFtimization led to *slower* performance. Benchmarks with dark bars showed a statistically significant average speedup (or slowdown). 17 of 18 benchmarks show a statistically significant change with `-O2`, and 9 of 18 show a significant change with `-O3`. In three cases for `-O2` and three for `-O3`, the statistically significant change is a performance *degradation*. Despite per-benchmark significance results, the data do not show significance across the entire suite of benchmarks for either `-O3` or `-O2` optimizations (Section 4.5.1).

4.5.1 Analysis of Variance

Evaluating optimizations with pairwise t-tests is error prone. This methodology runs a high risk of erroneously rejecting the null hypothesis (a type-I error). The parameter $\alpha = 0.05$ is the probability of observing the measured speedup, given that the optimization actually has no effect. Figure 4.7 shows the results for 36 hypothesis tests, each with a 5% risk of a false positive. We expect $36 * 0.05 = 1.8$ of these tests to show that an optimization had a statistically significant impact when in reality it did not.

Analysis of variance (ANOVA) allows us to test the significance of each optimization level over all benchmarks simultaneously. ANOVA relies on a normal assumption, but has been show to be robust to modest deviations from normality [27]. We run ANOVA with the same 18 benchmarks to test the significance of `-O2` over `-O1` and `-O3` over `-O2`.

ANOVA takes the total variance in execution times and breaks it down by source: the fraction due to differences between benchmarks, the impact of optimizations, interactions between the independent factors, and random variation between runs. Differences between benchmarks should not be included in the final result. We perform a one-way analysis of variance within subjects to ensure execution times are only compared between runs of the same benchmark.

For the speedup of `-O2`, the results show an F-value of 3.235 for one degree of freedom (the choice between `-O1` and `-O2`). The F-value is drawn from the F distribution [27]. The cumulative probability of observing any value drawn from $F(1) > 3.235 = 0.0898$ is the p-value for this test. The results show that `-O2` optimizations are significant at a 90% confidence level, but not at the 95% level.

The F-value for `-O3` is 1.335, again for one degree of freedom. This gives a p-value of 0.264. We fail to reject the null hypothesis and must conclude that compared to `-O2`, `-O3` *optimizations are not statistically significant*.

4.6 Conclusion

Researchers and software developers require effective performance evaluation to guide work in compiler optimizations, runtime libraries, and large applications. Automatic performance regression tests are now commonplace. Standard practice measures execution times before and after applying changes, but modern processor architectures make this approach unsound. Small changes to a program or its execution environment can perturb its layout, which affects caches and branch predictors. Two versions of a program, regardless of the number of runs, are only two samples from the distribution over possible layouts. Statistical techniques for comparing distributions require more samples, but randomizing layout over many runs may be prohibitively slow.

This chapter presents STABILIZER, a system that enables the use of the powerful statistical techniques required for sound performance evaluation on modern architectures. STABILIZER forces executions to sample the space of memory configurations by efficiently and repeatedly randomizing the placement of code, stack, and heap objects at runtime. Every run with STABILIZER consists of many independent and identically distributed (i.i.d.) intervals of random layout. Total execution time (the sum over these intervals) follows a Gaussian distribution by virtue of the Central Limit Theorem. STABILIZER thus enables the use of parametric statistical tests like ANOVA. We demonstrate STABILIZER's efficiency ($< 7\%$ median overhead) and its effectiveness by evaluating the impact of LLVM's optimizations on the SPEC CPU2006 benchmark suite. We find that the performance impact of `-O3` over `-O2` optimizations is indistinguishable from random noise.

We encourage researchers to download STABILIZER to use it as a basis for sound performance evaluation: it is available at `http://www.stabilizer-tool.org`.

CHAPTER 5

COZ: FINDING CODE THAT COUNTS WITH CAUSAL PROFILING

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

This chapter introduces *causal profiling*, a new approach to software profiling that guides developers to code where optimizations will improve program performance. A causal profiler conducts a series of *performance experiments* to empirically observe the effect of a potential optimization. Of course it is not possible to automatically speed up any line of code by an arbitrary amount. Instead, a causal profiler uses the novel technique of *virtual speedups* to mimic the effect of optimizing a specific line of code by a fixed amount. A line is virtually sped up by inserting pauses to slow all other threads each time the line runs. The key insight is that this slowdown has the same relative effect as running that line faster, thus "virtually" speeding it up.

Each performance experiment measures the effect of virtually speeding up one line of code by a specific amount. By conducting many performance experiments over the range of virtual speedup from between 0% (no change) and 100% (the line is completely eliminated), a causal profiler can predict the effect of any potential optimization on a program's performance.

Figure 5.1 shows a simple program that illustrates the shortcomings of existing profilers, originally introduced in Section 3.2. This program spawns two threads, which invoke functions f_a and f_b respectively. Most profilers will report that these functions comprise roughly half of the total execution time. Other profilers may report that f_a is on the critical path, or that the main thread spends roughly equal time waiting for f_a and f_b [41]. While accurate, all of this information is

```
 1 void a() { // ~6.7 seconds
 2   for(volatile size_t x=0; x<2000000000; x++) {}
 3 }
 4 void b() { // ~6.4 seconds
 5   for(volatile size_t y=0; y<1900000000; y++) {}
 6 }
 7 int main() {
 8   // Spawn both threads and wait for them.
 9   thread a_thread(a), b_thread(b);
10   a_thread.join(); b_thread.join();
11 }
```

Figure 5.1: A simple multithreaded program that illustrates the shortcomings of existing profilers. Optimizing f_a will improve performance by no more than 4.5%, while optimizing f_b would have no effect on performance.

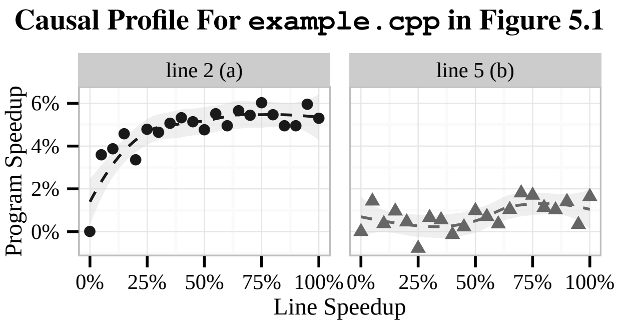

Figure 5.2: A causal profile for example.cpp, shown in Figure 5.1. This causal profile shows the potential impact of optimizing either f_a or f_b. The x-axis shows the amount of speedup to either f_a or f_b, and the y-axis shows the predicted program speedup. The gray area shows standard error. Optimizing f_a will improve performance by at most 4.5%, and optimizing f_b would have no effect on performance. The causal profile predicts both outcomes within 0.5%.

potentially misleading. Optimizing f_a away entirely will only speed up the program by 4.5% because f_b becomes the new critical path.

To demonstrate the effectiveness of causal profiling, we have developed COZ, a causal profiler for Linux. Figure 5.2 shows the results of running COZ on the program in Figure 5.1. This profile plots the hypothetical speedup of a line of code (x-axis) versus its impact on execution time (y-axis). The graph correctly shows that optimizing f_a or f_b in isolation would have little effect on program performance; COZ's predictions are within 0.5% of the actual effect of optimizing this code.

Causal profiling further departs from conventional profiling by making it possible to view the effect of optimizations on both *throughput* and *latency*. To profile throughput, developers specify a ***progress point***, indicating a line in the code that corresponds to the end of a unit of work. For example, a progress point could be the point at which a transaction concludes, when a web page finishes rendering, or when a query completes. A causal profiler then measures the rate of visits to each progress point to determine any potential optimization's effect on throughput. To profile latency, programmers instead place 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 show that causal profiling accurately predicts optimization opportunities, and that it is effective at guiding optimization efforts. We apply COZ to Memcached, SQLite, and the extensively studied PARSEC benchmark suite. Guided by COZ's output, we optimized the performance of Memcached by 9%, SQLite by 25%, and six PARSEC applications by as much as 68%. These optimizations typically involved modifying under 10 lines of code. When possible to accurately measure the size of our optimizations on the line(s) identified by COZ, we compare the observed performance improvements to COZ's predictions: in each case, we find that the real effect of our optimization matches COZ's prediction.

5.0.1 Contributions

This chapter makes the following contributions:

1. It presents **causal profiling**, which identifies code where optimizations will have the largest impact. Using *virtual speedups* and *progress points*, causal profiling directly measures the effect of potential optimizations on both throughput and latency (§5.1).

2. It presents **COZ**, a causal profiler that works on unmodified Linux binaries. It describes COZ's implementation (§5.2), and demonstrates its efficiency and effectiveness at identifying optimization opportunities (§5.3).

5.1 Causal Profiling Overview

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

5.1.1 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.

5.1.2 Experiment Initialization

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

5.1.3 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 5.2.4, we show the equivalence between the virtual speedup mechanism shown in Figure 5.3 and the sampling approach used by 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.

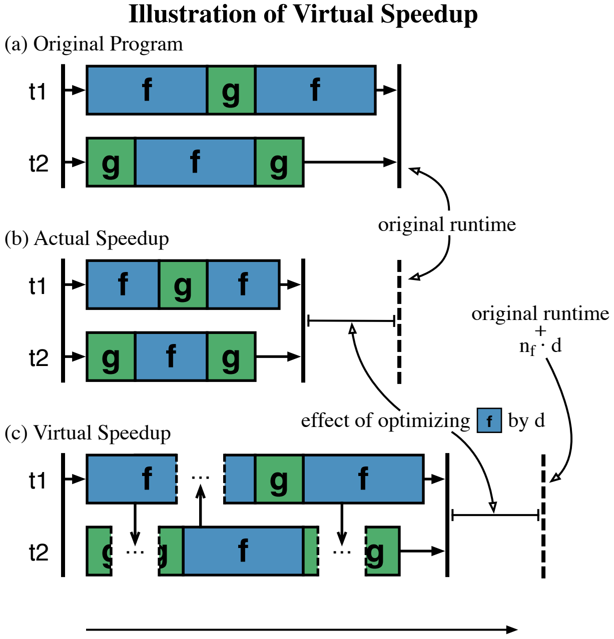

Figure 5.3: An illustration of virtual speedup: (a) shows the original execution of two threads running functions `f` and `g`; (b) shows the effect of a *actually* speeding up `f` by 40%; (c) shows the effect of *virtually* speeding up `f` by 40%. Each time `f` runs in one thread, all other threads pause for 40% of `f`'s original execution time (shown as ellipsis). The difference between the runtime in (c) and the original runtime plus $n_f \cdot d$—the number of times `f` ran times the delay size—is the same as the effect of actually optimizing `f`.

5.1.4 Ending an Experiment

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

5.1.5 Producing a Causal Profile

After an application has been profiled with Coz, the results of all the performance experiments can be combined to produce a causal profile. Each experiment has two independent variables: the line chosen for virtual speedup and the amount of virtual speedup. Coz records the dependent variable, the rate of visits to each progress point, in two numbers: the total number of visits to each progress point and the effective duration of the experiment (the real runtime minus the total length of all pauses). Experiments with the same independent variables can be combined by adding the progress point visits and experiment durations.

Once experiments have been combined, COZ groups experiments by the line that was virtually sped up. Any lines that do not have a measurement of 0% virtual speedup are discarded; without this baseline measurement we cannot compute a percent speedup relative to the original program. Measuring this baseline separately for each line guarantees that any line-dependent overhead from virtual speedups, such as the additional cross-thread communication required to insert delays when a frequently-executed line runs, will not skew profile results. By default, COZ will also discard any lines with fewer than 5 different virtual speedup amounts (a plot that only shows the effect of a 75% virtual speedup is not particularly useful). Finally, we compute the percent program speedup for each grouped experiment as the percent change in rate of visits to each progress point over the baseline (virtual speedup of 0%). COZ then plots the resulting table of line and program speedups for each line, producing the profile graphs shown in this chapter.

5.1.6 Interpreting a Causal Profile

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

5.2 Implementation

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

5.2.1 Core Mechanisms

COZ uses sampling to implement both virtual speedups and progress points. When a user starts a program with the `coz` command, COZ injects a profiling runtime library into the program's address space using `LD_PRELOAD`. This runtime library creates a dedicated profiler thread to run performance experiments, but also intercepts each thread startup and shutdown to start and stop sampling in the thread using the `perf_even` API. Using the `perf_event` API, COZ collects both the current program counter and user-space call stack from each thread every 1ms. To keep overhead low, COZ processes samples in batches of ten by default (every 10ms). Processing samples more frequently is unlikely to improve accuracy, as the additional overhead would distort program execution.

5.2.1.1 Attributing Samples to Source Locations

COZ uses DWARF debug information to map sampled program counter values to source locations. The profiled program does not need to contain DWARF line information; COZ will use the same search procedure as GDB to locate external debug information if necessary [28]. Note that debug information is available even for optimized code, and most Linux distributions offer packages that include this information for common libraries.

By default, COZ will only collect debug information for the main executable. This means COZ will only test potential optimizations in the main program's source files. Users can specify a source scope to control which source files COZ will select lines from to evaluate potential optimizations. Likewise, users can specify a binary scope to control which executables and libraries will be profiled. Users should use these scope options to specify exactly which code they are willing or able to change to improve their program's performance.

5.2.2 Performance Experiment Implementation

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

5.2.2.1 Starting a Performance Experiment

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

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

Lines for virtual speedup 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 speedup amount have been selected, COZ saves the current values of all progress point counters and begins the performance experiment.

5.2.2.2 Running a Performance Experiment

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

5.2.3 Progress Point Implementation

COZ supports three mechanisms for monitoring progress points: *source-level*, *breakpoint*, and *sampled*.

5.2.3.1 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 `COZ_PROGRESS` macro in the program's source code at the appropriate location.

5.2.3.2 Breakpoint Progress Points

Breakpoint progress points are specified at the command line. COZ uses the Linux `perf_event` API to set a breakpoint at the first instruction in a line specified in the profiler arguments.

5.2.3.3 Sampled Progress Points

Sampled progress points are specified on 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 exact counts are unknown.

5.2.3.4 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: $L = \lambda W$ [52]. 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.

5.2.4 Virtual Speedup Implementation

A critical component of any causal profiler is the ability to virtually speed up any fragment of code. A naive implementation of virtual speedups is shown in Figure 5.3; each time the function `f` runs, all other threads are paused briefly. If `f` has an average runtime of $\bar{t_f}$ each time it is called and threads are paused for time d each time `f` runs, then `f` has an *effective* average runtime of $\bar{t_f} - d$.

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

5.2.5 Implementing Virtual Speedup with Sampling

The previous discussion of virtual speedups assumes an implementation where every time a specific line of code executes all other threads instantaneously pause for a very brief time (a fraction of the time require to run a single line). Unfortunately, this approach would incur prohibitively high overhead that would distort program execution, making the profile useless. Instead, COZ periodically samples the program counter and counts samples that fall in the line selected for virtual speedup. Then, other threads are delayed proportionally to the number of samples. The number of samples in the selected line, s, is approximately

$$s \approx \frac{n \cdot \bar{t}}{P} \tag{5.1}$$

where P is the period of time between samples, $\bar{t}$ is the average 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 $\bar{t} - d$, while executions of the selected line that are not sampled simply take time $\bar{t}$. The *effective* average time to run the selected line is

$$\bar{t}_e = \frac{(n - s) \cdot \bar{t} + s \cdot (\bar{t} - d)}{n}. \tag{5.2}$$

Using (5.1), this reduces to

$$\bar{t}_e = \frac{n \cdot \bar{t} \cdot (1 - \frac{\bar{t}}{P}) + \frac{n \cdot \bar{t}}{P} \cdot (\bar{t} - d)}{n} = \bar{t} \cdot (1 - \frac{d}{P}) \tag{5.3}$$

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

$$\Delta\bar{t} = 1 - \frac{\bar{t}_e}{\bar{t}} = \frac{d}{P}. \tag{5.4}$$

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

5.2.5.1 Pausing Other Threads

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

5.2.5.2 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.

5.2.5.3 Thread Creation

To start sampling and adjust delays, COZ interposes on the `pthread_create` function. COZ first initiates `perf_event` sampling in the new thread. It then inherits the parent thread's local delay count; any previously inserted delays to the parent thread also delayed the creation of the new thread.

5.2.5.4 Handling Suspended Threads

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

However, a thread can also be suspended while waiting for a mutex or other POSIX synchronization primitive. As with blocking I/O, required delays will accumulate while the thread is suspended, but COZ may not need to insert all of these delays when the thread resumes. When one thread

Potentially *unblocking* calls	
`pthread_mutex_unlock`	unlock a mutex
`pthread_cond_signal`	wake one waiter on a c.v.
`pthread_cond_broadcast`	wake all waiters on c.v.
`pthread_barrier_wait`	wait at a barrier
`pthread_kill`	send signal to a thread
`pthread_exit`	terminate this thread

Table 5.1: POSIX functions that Coz intercepts to handle thread wakeup. 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`	wait on a condition variable
`pthread_barrier_wait`	wait at a barrier
`pthread_join`	wait for a thread to complete
`sigwait`	wait for a signal
`sigwaitinfo`	wait for a signal
`sigtimedwait`	wait for a signal (with timeout)
`sigsuspend`	wait for a signal

Table 5.2: POSIX functions that Coz intercepts to handle thread blocking. Coz intercepts calls to these functions to update delay counts before and after blocking.

resumes after waiting for a mutex, another thread must have unlocked that mutex. If the unlocking thread has executed all the required delays, then the blocked thread has effectively already been delayed; it should not insert any additional delays after unblocking.

To correctly handle suspended threads, a causal profiler must follow a simple rule: If a suspended thread resumes execution because of another thread, the suspended thread should be "credited" for any delays inserted in the thread responsible for waking it up. Otherwise, the thread should insert all the necessary delays that accumulated during the time the thread was suspended. To simplify the implementation of this policy, Coz forces a thread to execute all required delays before it does anything that could block that thread (see Table 5.2) or wake a suspended thread (shown in Table 5.1). This means that any resumed thread can skip any required delays after returning from a call which may have blocked the thread. Note that this special handling is only required for operations that can suspend a thread. Coz can accommodate programs with ad-hoc synchronization that does not suspend threads with no special handling.

5.2.5.5 Attributing Samples to Source Lines

Samples are attributed to source lines using the source map constructed at startup. When a sample does not fall in any in-scope source line, the profiler walks the sampled callchain to find the first in-scope address. This 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.

5.2.5.6 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 be equal. 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.

5.2.5.7 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:

$$P = \frac{T}{N} = \frac{t_A + t_B}{N}. \tag{5.5}$$

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:

$$\mathbb{E}[s_{obs}] = s \cdot \frac{t_{obs}}{t_A}, \quad \text{therefore} \quad t_A \approx s \cdot \frac{t_{obs}}{s_{obs}}. \tag{5.6}$$

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:

$$\Delta p_A = \frac{\frac{t_A}{n_A} - \frac{{t_A}'}{n_A}}{\frac{t_A}{n_A}} = \frac{t_A - {t_A}'}{t_A} \tag{5.7}$$

where n_A is the number of progress point visits during phase A. Using (5.5), 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 (5.6) and (5.7),

$$\Delta P = \Delta p_A \frac{t_A}{T} \approx \Delta p_A \cdot \frac{t_{obs}}{s_{obs}} \cdot \frac{s}{T}. \tag{5.8}$$

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

Summary of Optimization Results

Application	*Speedup*	*Diff Size*	*LOC*
blackscholes	$2.56\% \pm 0.41\%$	$-61, +4$	342
dedup	$8.95\% \pm 0.27\%$	$-3, +3$	2,570
ferret	$21.27\% \pm 0.17\%$	$-4, +4$	5,937
fluidanimate	$37.5\% \pm 0.56\%$	$-1, +0$	1,015
streamcluster	$68.4\% \pm 1.12\%$	$-1, +0$	1,779
swaptions	$15.8\% \pm 1.10\%$	$-10, +16$	970
Memcached	$9.39\% \pm 0.95\%$	$-6, +2$	10,475
SQLite	$25.60\% \pm 1.00\%$	$-7, +7$	92,635

Table 5.3: Results for benchmarks optimized using COZ. All benchmarks were run ten times before and after optimization. Standard error for speedup was computed using Efron's bootstrap method, where speedup is defined as $\frac{t_0 - t_{opt}}{t_0}$. All speedups are statistically significant at the 99.9% confidence level ($\alpha = 0.001$) using the one-tailed Mann-Whitney U test, which does not rely on any assumptions about the distribution of execution times. Lines of code do not include blank or comment-only lines.

5.3 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?

5.3.1 Experimental Setup

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

5.3.2 Effectiveness

We demonstrate causal profiling's effectiveness through case studies. Using COZ, we collect causal profiles for Memcached, SQLite, and the PARSEC benchmark suite. Using these causal

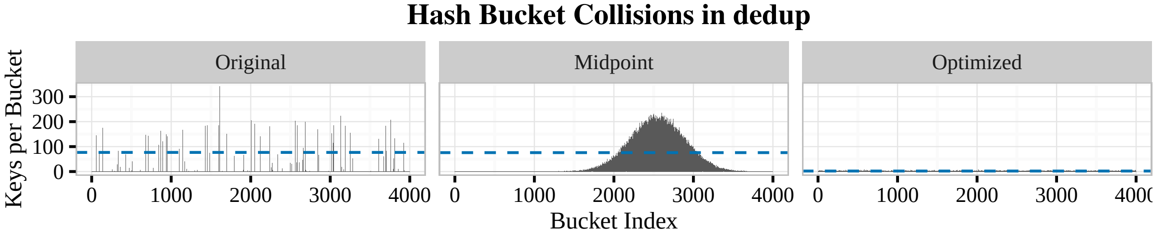

Figure 5.4: Hash function performance in dedup. In the dedup benchmark, COZ identified hash bucket traversal as a bottleneck. These plots show collisions per-bucket before, mid-way through, and after optimization of the dedup benchmark (note different y-axes). The dashed horizontal line shows average collisions per-utilized bucket for each version. Fixing dedup's hash function improved performance by 8.9%.

profiles, we were able to make small changes to two of the real applications and six PARSEC benchmarks, resulting in performance improvements as large as 68%. Table 5.3 summarizes the results of our optimization efforts. We describe our experience using COZ below, with three general outcomes: (1) cases where COZ found optimization opportunities that gprof and perf did not (dedup, ferret, and SQLite); (2) cases where COZ identified contention (fluidanimate, streamcluster, and Memcached); and (3) cases where both COZ and a conventional profiler identified the optimization we implemented (blackscholes and swaptions).

5.3.2.1 Case Study: dedup

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

COZ identifies the source line `hashtable.c:217` as the best opportunity for optimization. This code is the top of the `while` loop in `hashtable_search` that traverses the linked list of entries that have been assigned to the same hash bucket. This suggests that dedup's shared hash table has a significant number of collisions. Increasing the hash table size had no effect on performance. This led us to examine dedup's hash function, which could also be responsible for the large number of hash table collisions. We discovered that dedup's hash function maps keys to just 2.3% of the available buckets; over 97% of buckets were never used during the entire execution.

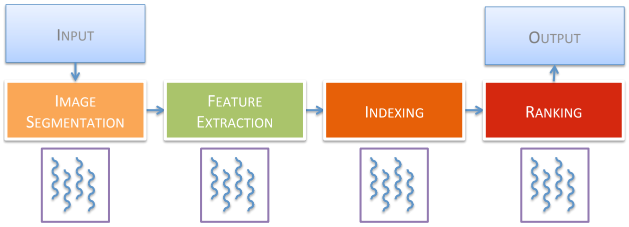

Figure 5.5: An illustration of ferret's pipeline. The middle four stages each have an associated thread pool; the input and output stages each consist of one thread. The colors represent the impact on throughput of each stage, as identified by COZ: green is low impact, orange is medium impact, and red is high impact.

The original hash function adds characters of the hash table key, which leads to virtually no high order bits being set. The resulting hash output is then passed to a bit shifting procedure intended to compensate for poor hash functions. We removed the bit shifting step, which increased hash table utilization to 54.4%. We then changed the hash function to bitwise XOR 32 bit chunks of the key. This increased hash table utilization to 82.0% and resulted in an $8.95\% \pm 0.27\%$ performance improvement. Figure 5.4 shows the rate of bucket collisions of the original hash function, the same hash function without the bit shifting "improvement", and our final hash function. The entire optimization required changing just three lines of code. As with ferret, this result was achieved by one graduate student who was initially unfamiliar with the code; the entire profiling and tuning effort took just two hours.

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

5.3.2.2 Case Study: ferret

The ferret benchmark performs a content-based image similarity search. Ferret consists of a pipeline with six stages: the first and the last stages are for input and output. The middle four stages perform image segmentation, feature extraction, indexing, and ranking. Ferret takes two arguments:

an input file and a desired number of threads, which are divided equally across the four middle stages. We first inserted a progress point in the final stage of the image search pipeline to measure throughput (`ferret-parallel.c:398`). We then ran COZ with the source scope set to evaluate optimizations only in `ferret-parallel.c`, rather than across the entire ferret toolkit.

Figure 5.6 shows the top three lines identified by COZ, using its default ranking metric. Lines 320 and 358 are calls to `cass_table_query` from the indexing and ranking stages. Line 255 is a call to `image_segment` in the segmentation stage. Figure 5.5 depicts ferret's pipeline with the associated thread pools (colors indicate COZ's computed impact on throughput of optimizing these stages).

Because each important line falls in a different pipeline stage, and because COZ did not find any important lines in the queues shared by adjacent stages, we can easily "optimize" a specific line by shifting threads to that stage. We modified femrret to let us specify the number of threads assigned to each stage separately, a four-line change.

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

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

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

5.3.2.3 Case Study: SQLite

The SQLite database library is widely used by many applications to store relational data. The embedded database, which can be included as a single large C file, is used for many applications

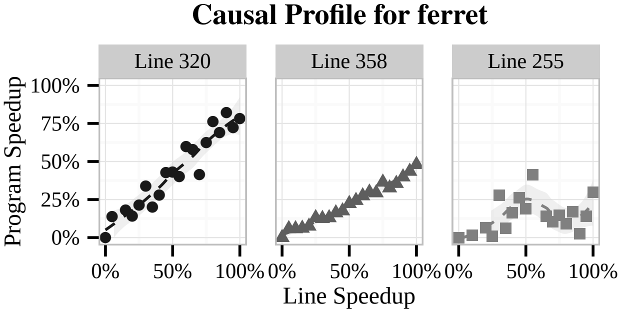

Figure 5.6: COZ output for the unmodified ferret application. The x-axis shows the amount of virtual speedup applied to each line, versus the resulting change in throughput on the y-axis. The top two lines are executed by the indexing and ranking stages; the third line is executed during image segmentation.

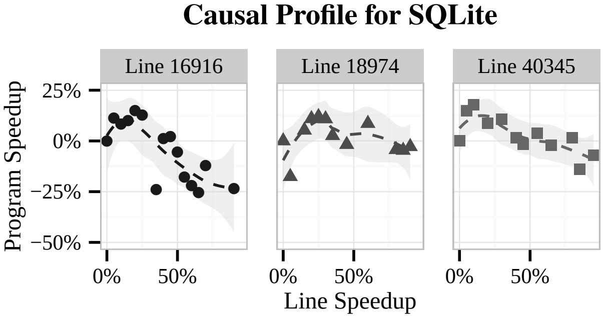

Figure 5.7: A causal profile for SQLite. The three lines shown from correspond to the function prologues for `sqlite3MemSize`, `pthreadMutexLeave`, and `pcache1Fetch`. A small optimization to each of these lines will improve program performance, but beyond about a 25% speedup to each line, COZ predicts that the optimization would actually lead to a slowdown. Changing indirect calls into direct calls for these functions improved overall performance by $25.6\% \pm 1.0\%$.

including Firefox, Chrome, Safari, Opera, Skype, iTunes, and is a standard component of Android, iOS, Blackberry 10 OS, and Windows Phone 8. We evaluated SQLite performance using a write-intensive parallel workload, where each thread rapidly inserts rows to its own private table. While this benchmark is synthetic, it exposes any scalability bottlenecks in the database engine itself because all threads should theoretically operate independently. We placed a progress point in the benchmark itself (which is linked with the database), which executes after each insertion.

COZ identified three important optimization opportunities, shown in Figure 5.7. At startup, SQLite populates a large number of structs with function pointers to implementation-specific functions, but most of these functions are only ever given a default value determined by compile-time options. The three functions COZ identified unlock a standard pthread mutex, retrieve the next item

Conventional Profile for SQLite

```
% Runtime    Symbol
  85.55%     _raw_spin_lock
  1.76%      x86_pmu_enable_all
    ... 30 lines ...
  0.10%      rcu_irq_enter
  0.09%      sqlite3MemSize
  0.09%      source_load
    ... 26 lines ...
  0.03%      __queue_work
  0.03%      pcache1Fetch
  0.03%      kmem_cache_free
  0.03%      update_cfs_rq_blocked_load
  0.03%      pthreadMutexLeave
  0.03%      sqlite3MemMalloc
```

Figure 5.8: A conventional profile for SQLite, collected using the Linux perf tool. The lines we found using COZ's output and optimized are shown in bold. These lines make up just 0.15% of total program runtime, but small optimizations to these lines produced a $25.6\% \pm 1.0\%$ program speedup.

from a shared page cache, and get the size of an allocated object. These simple functions do very little work, so the overhead of the indirect function call is relatively high. Replacing these indirect calls with direct calls resulted in a $25.60\% \pm 1.00\%$ speedup.

5.3.2.3.1 Comparison with conventional profilers. Unfortunately, running SQLite with gprof segfaults immediately. The application does run with the Linux perf tool, which reports that the three functions COZ identified account for a total of just 0.15% of total runtime (shown in Figure 5.8). Using perf, a developer would be misled into thinking that optimizing these functions would be a waste of time. COZ accurately shows that the opposite is true: optimizing these functions has a dramatic impact on performance.

5.3.2.4 Case Study: fluidanimate

The fluidanimate benchmark, also provided by Intel, is a physical simulation of an incompressible fluid for animation. The application spawns worker threads that execute in eight concurrent phases, separated by a barrier. We placed a progress point immediately after the barrier, so it executes each time all threads complete a phase of the computation.

COZ identifies a single modest potential speedup in the thread creation code, but there was no obvious way to speed up this code. However, COZ also identified two significant points of contention, indicated by a downward sloping causal profile. Figure 5.9 shows COZ's output for these two lines.

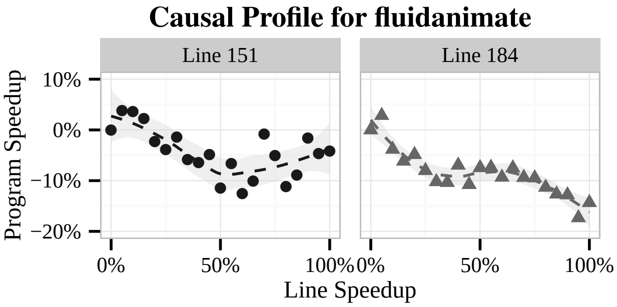

Figure 5.9: COZ output for fluidanimate, prior to optimization. COZ finds evidence of contention in two lines in `parsec_barrier.cpp`, the custom barrier implementation used by both fluidanimate and streamcluster. This causal profile reports that optimizing either line will slow down the application, not speed it up. These lines precede calls to `pthread_mutex_trylock` on a contended mutex. Optimizing this code would increase contention on the mutex and interfere with the application's progress. Replacing this inefficient barrier implementation sped up fluidanimate and streamcluster by 37.5% and 68.4% respectively.

This result tells us that optimizing the indicated line of code would actually *slow down* the program, rather than speed it up. Both lines COZ identifies are in a custom barrier implementation, immediately before entering a loop that repeatedly calls `pthread_mutex_trylock`. Removing this spinning from the barrier would reduce the contention, but it was simpler to replace the custom barrier with the default `pthread_barrier` implementation. This one line change led to a $37.5\% \pm 0.56\%$ speedup.

5.3.2.5 Case Study: streamcluster

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

5.3.2.6 Case Study: Memcached

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

Most of the lines COZ identifies are cases of contention, with a characteristic downward-sloping causal profile plot. One such line is at the start of `item_remove`, which locks an item in the cache and then decrements its reference count, freeing it if the count goes to zero. To reduce lock initialization overhead, Memcached uses a static array of locks to protect items, where each item selects its lock using a hash of its key. Consequently, locking any one item can potentially contend with independent accesses to other items whose keys happen to hash to the same lock index. Because reference counts are updated atomically, we can safely remove the lock from this function, which resulted in a $9.39\% \pm 0.95\%$ speedup.

5.3.2.7 Case Study: blackscholes

The blackscholes benchmark, provided by Intel, solves the Black–Scholes differential equation to price a portfolio of stock options. We placed a progress point after each thread completes one round of the iterative approximation to the differential equation (`blackscholes.c:259`). COZ identifies many lines in the `CNDF` and `BlkSchlsEqEuroNoDiv` functions that would have a small impact if optimized. This same code was identified as a bottleneck by ParaShares [45]; this is the only optimization we describe here that was previously reported. This block of code performs the main numerical work of the program, and uses many temporary variables to break apart the complex computation. Manually eliminating common subexpressions and combining 61 piecewise calculations into 4 larger expressions resulted in a $2.56\% \pm 0.41\%$ program speedup.

5.3.2.8 Case Study: swaptions

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

Results for Unoptimized Applications

Benchmark	*Progress Point*	*Top Optimization*
bodytrack	TicketDispenser.h line 106	ParticleFilter.h line 262
canneal	annealer_thread.cpp line 87	netlist_elem.cpp line 82
facesim	taskQDistCommon.c line 109	MATRIX_3X3.h line 136
freqmine	fp_tree.cpp line 383	fp_tree.cpp line 301
raytrace	BinnedAllDimsSaveSpace.cxx line 98	RTEmulatedSSE.hxx line 784
vips	threadgroup.c line 360	im_Lab2LabQ.c line 98
x264	encoder.c line 1165	common.c line 687

Table 5.4: Summary of progress points and causal profile results for the remaining PARSEC benchmarks.

COZ identified three significant optimization opportunities, all inside nested loops over a large multidimensional array. One of these loops zeroed out consecutive values. A second loop filled part of the same large array with values from a distribution function, with no obvious opportunities for optimization. The third nested loop iterated over the same array again, but traversed the dimensions in an irregular order. Reordering these loops and replacing the first loop with a call to `memset` sped execution by $15.8\% \pm 1.10\%$.

5.3.2.9 Effectiveness Summary

Our case studies confirm that COZ is effective at identifying optimization opportunities and guiding performance tuning. In every case, the information COZ provided led us directly to the optimization we implemented. In most cases, COZ identified around 20 lines of interest, with as many as 50 for larger programs (Memcached and x264). COZ identified optimization opportunities in all of the PARSEC benchmarks, but some required more invasive changes that are out of scope for an evaluation of COZ's effectiveness. Table 5.4 summarizes our findings for the remaining PARSEC benchmarks. We have submitted patches to the developers of all the applications we optimized.

5.3.3 Accuracy

For most of the optimizations described above, it is not possible to quantify the effect our optimization had on the specific lines that COZ identified. However, for two of our case studies—ferret and dedup—we can directly compute the effect our optimization had on the line COZ identified

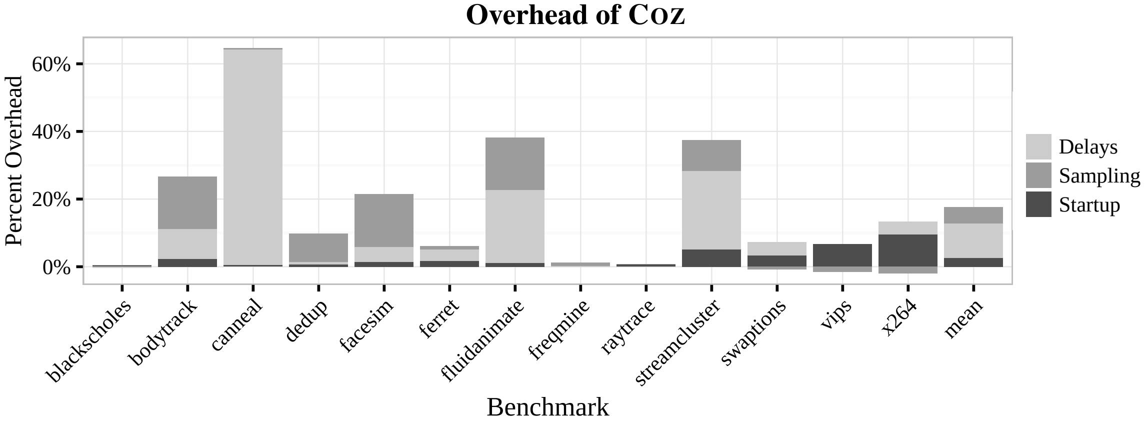

Figure 5.10: Percent overhead for each of COZ's possible sources of overhead. *Delays* are the overhead due to adding delays for virtual speedups, *Sampling* is the cost of collecting and processing samples, and *Startup* is the initial cost of processing debugging information. Note that sampling results in slight performance *improvements* for swaptions, vips, and x264.

and compare the resulting speedup to COZ's predictions. Our results show that COZ's predictions are highly accurate.

To optimize ferret, we increased the number of threads for the indexing stage from 16 to 22, which increases the throughput of line 320 by 27%. COZ predicted that this improvement would result in a 21.4% program speedup, which is nearly the same as the 21.2% we observe.

For dedup, COZ identified the top of the `while` loop that traverses a hash bucket's linked list. By replacing the degenerate hash function, we reduced the average number of elements in each hash bucket from 76.7 to just 2.09. This change reduces the number of iterations from 77.7 to 3.09 (accounting for the final trip through the loop). This reduction corresponds to a speedup of the line COZ identified by 96%. For this speedup, COZ predicted a performance improvement of 9%, very close to our observed speedup of 8.95%.

5.3.4 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 5.10 breaks down the total overhead of running COZ on each of the PARSEC benchmarks by category. The average overhead with COZ is 17.6%. COZ collects debug information at startup, which contributes 2.6% to the average overhead. Sampling during program execution and attributing these samples to lines using debug information is responsible for 4.8% of the average overhead. The remaining overhead (10.2%) comes from the delays COZ inserts to perform virtual speedups.

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

5.3.4.1 Reducing Overhead

Most programs have sufficiently long running times (mean: 103s) to amortize the cost of processing debug information, but especially large executables can be expensive to process at startup (`x264` and `vips`, for example). COZ could be modified to collect and process debug information lazily to reduce startup overhead. Sampling overhead comes mainly from starting and stopping sampling with the `perf_event` API at thread creation and exit. This cost could be amortized by sampling globally instead of per-thread, which would require root permissions on most machines. If the `perf_event` API supported sampling all threads in a process this overhead could be eliminated. Delay overhead, the largest component of COZ's total overhead, could be reduced by allowing programs to execute normally for some time between each experiment. Increasing the time between experiments would significantly reduce overhead, but a longer profiling run would be required to collect a usable profile.

5.3.4.2 Efficiency Summary

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

5.4 Conclusion

Profilers are the primary tool in the programmer's toolbox for identifying performance tuning opportunities. Previous profilers only observe actual executions and correlate code with execution time or performance counters. This information can be of limited use because the amount of time spent does not necessarily correspond to where programmers should focus their optimization efforts. Past profilers are also limited to reporting end-to-end execution time, an unimportant quantity for servers and interactive applications whose key metrics of interest are throughput and latency. Causal profiling is a new, experiment-based approach that establishes causal relationships between hypothetical optimizations and their effects. By virtually speeding up lines of code, causal profiling identifies and quantifies the impact on either throughput or latency of any degree of optimization to any line of code. Our prototype causal profiler, COZ, is efficient, accurate, and effective at guiding optimization efforts.

CHAPTER 6

AMP: SCALABILITY PREDICTION AND GUIDANCE WITH LOAD AMPLIFICATION

Performance evaluation is an important part of the testing process before software is released, but test environments bear little resemblance to deployment. One key difference between testing and deployment is load; testing a full-scale version of a software service would require developers to have exclusive control over an exact duplicate of the production environment, which would be a substantial burden for most developers. Additional resources are required to generate realistic load for testing, potentially requiring hundreds or thousands of machines [25]. While some companies have the necessary resources to perform comprehensive performance testing, the required infrastructure is out of reach for most developers. As a result, software is almost certainly deployed without comprehensive performance testing.

Figure 6.1 shows a simple multithreaded program that uses the producer–consumer pattern to handle arriving work items. This program creates a set of producer threads, which generate tasks and add them to a work queue, as well as a set of consumer threads, which take tasks from the work queue and complete them. To make it as easy as possible to reason about the performance of this program, the time to produce a task is configured to be exactly the same as the time required to run the task. Synchronization is not shown in this example, but the actual implementation uses an unbounded queue; producers will never block when adding a task to the work queue, but consumers will block when the queue is empty.

Figure 6.2 shows the performance of this program under increasing load using two producer threads. For this application, increased load means a shorter time between task creation events in the two producer threads. With two consumers, the system is balanced; the mean time between task creations is exactly equal to task execution time divided by the number of consumer threads that actually execute these tasks. Decreasing the task creation time results in a *very slight* performance

Code for `example.cpp`

```cpp
work_queue tasks;

void producer_fn() {
  while(!done) {
    // Create a new task
    task t = task::produce();
    // Add the task to the work queue
    tasks.add(t);
  }
}

void consumer_fn() {
  while(!done) {
    // Get the next task or block if non are available
    task t = tasks.get_next();
    // Run the new task
    t.run();
  }
}

int main(int argc, char** argv) {
  thread_pool producers(NUM_PRODUCERS, producer_fn);
  thread_pool consumers(NUM_CONSUMERS, consumer_fn);
  consumers.join();
  producers.join();
  return 0;
}
```

Figure 6.1: A simple multithreaded program with a scalability issue that will not appear in testing. This program creates a thread pool with `NUM_THREADS` threads. Each thread waits for a task to arrive, acquires a lock, executes the task in isolation, then releases the lock.

improvement, likely because a higher load reduces the number of times a consumer thread will suspend briefly before the next item is added to the task queue. With three and four consumer threads, the system is starved waiting for inputs; increasing the load improves the throughput in both versions up to the point where the rate of task creations matches task completion time divided by the number of consumer threads. These results were collected by changing the time each producer waits before creating a new task, but this approach is unlikely to work in a real system; task creation may depend on network events, I/O operations, or computationally-intensive work. To evaluate the scalability of a real system, developers need a way to generate higher load than is actually feasible in a test environment.

This chapter introduces *load amplification*, which allows developers to create the effect of increased load on a software system. By simulating the effect of increased load in a running program, load amplification can expose scalability issues that result from either hardware and software contention. To demonstrate the effectiveness of this approach we present AMP, an extension

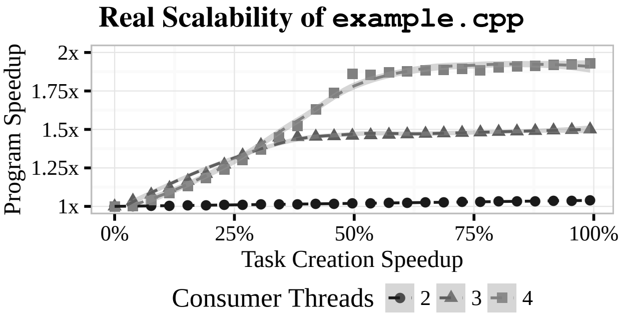

Figure 6.2: Scalability measurements for the program in Figure 6.1. The x-axis shows the amount of increased load, where 0% is the original load on the program and 100% reduces the task inter-arrival time to zero. The y-axis shows the percent increase in program performance; a 2x performance increase is a 50% reduction in program runtime. Increasing load only improves performance when consumers outnumber producers, but tapers off beyond the point where load is sufficient to keep all consumers busy. All points are the average execution time over ten runs.

to COZ. AMP can predict the scalability of an application under increasing load, or can use load amplification in combination with causal profiling to pinpoint lines of code where optimizations would improve scalability. To use AMP, developers simply mark an *arrival point* using a macro in the program source. This point should execute once each time a task arrives. AMP uses *virtual speedup* to create the effect of an increased arrival rate with no modifications to the program.

Figure 6.3 shows AMP's predictions for the example program in Figure 6.1, with an arrival point placed just before line 8 in `example.cpp`. Unlike in Figure 6.2, no change was made to the task creation time; these results were collected without modifications to the example program, except to vary the number of consumer threads. These predictions show that AMP can be used to accurately predict the scalability of programs where it is impractical or impossible to generate increased load in a test environment. Furthermore, AMP can be used in combination with COZ to identify potential optimizations that will improve performance *at increased load*. This process, which is described in Section 6.2, gives developers the ability to find and fix scalability bottlenecks on software that has not yet been run at scale.

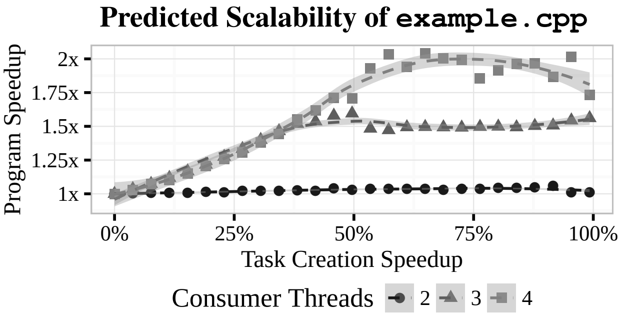

Figure 6.3: AMP's scalability predictions for the program in Figure 6.1. Each point is the average effective runtime (adjusted using virtual speedup) over ten runs. Using load amplification, AMP is able to create the effect of increased load on the program, then directly measure the performance under this increased load. AMP's predictions closely match the real scalability of this program in each configuration, shown in Figure 6.2.

6.0.1 Contributions

This chapter makes the following contributions:

1. It presents **load amplification**, an extension to virtual speedup that can simulate the effect of increased load on a system.

2. It presents **AMP**, an implementation of load amplification as an extension to COZ. Developers simply mark an *arrival point* in the code, and AMP uses virtual speedups to multiply the rate of requests by some specified amount.

3. It outlines the use of AMP and COZ in combination, which gives developers the ability to identify potential optimizations that will improve a system's performance *at scale*.

6.1 Design and Implementation

This section describes the usage model for AMP and the basic mechanisms required to perform load amplification.

6.1.1 Using AMP

Before a program can be run with AMP, the developer must mark at least one point in the program as an *arrival point* by inserting the `COZ_ARRIVAL` macro somewhere in the program. This point should execute once every time a request arrives at the system. For programs that use COZ's latency profiling points, the `COZ_BEGIN` point is automatically considered an arrival point. The latter method of specifying an arrival point allows AMP to measure the effect of increasing load on both the rate of requests completing this latency-critical section (rate of visits to the `COZ_END` point) and the effect on average latency between the two points, while the former measures only throughput at the program's progress points.

Once the developer has specified an arrival point, AMP can use virtual speedup—a mechanism shared with causal profiling—to increase the effective rate of visits to this point in the program. The amount that this rate is increased is up to the developer; unlike with causal profiling, it would not make sense for AMP to select a load increase at random. The developer is responsible for determining a target load on the system, the amount of load that is actually generated under test conditions, and computing an arrival speedup to make up the difference.

Once developers have added an arrival point, they can then run the program with AMP using the `coz` driver with the addition of the `--arrival-speedup=N` option, where `N` is number of nanoseconds that each arrival event should be sped up. To run the program with load amplification alone, they should include the `--fixed-speedup=0` flag, which tell COZ not to virtually speed up any lines of code. The profile collected for this run will report the effective progress rate under the amplified load during each experiment. Adding the `--end-to-end` flag makes the entire execution run as part of a single experiment, where the duration of the experiment is the effective program runtime with amplified load.

6.1.2 Amplifying Load with Virtual Speedup

While AMP relies on the virtual speedup mechanism from COZ to increase load, there are some differences in the accounting. First, COZ does not have an accurate count of the number of visits to a particular line of code; instead, the profiler uses samples to approximate time spent executing a particular line of code. By virtually speeding up lines of code by a percentage rather than an absolute amount, the number of visits to the line cancels out in the virtual speedup equation.

For load amplification, we have the opposite problem: we can track the exact number of visits to an arrival point, but the time spent actually executing the arrival point is not meaningful. In this case, we can only reliably create the effect of an absolute speedup rather than a percent change like with causal profiling. It is up to the developer to select an arrival rate that makes sense for the application under examination. To do this, the developer should first measure the actual arrival rate on the test environment. Say for example the test environment generates 1000 requests per second, while a deployed service would need to handle 4000 requests per second. This means the inter-arrival time in test is 1 millisecond, while the deployed inter-arrival time is 250 microseconds. In this case, an arrival speedup of 750 microseconds would create the effect of full deployed load when executing in the test environment.

A thread i keeps a count of the number of times it has executed the arrival point, a_i. Each time, this thread will be virtually sped up by delaying all other threads by some time d. Over a period of time T_i, the effective execution time $T_i{}'$ is:

$$T_i{}' = T_i - d \times a_i \tag{6.1}$$

Over this period of time, the *effective* inter-arrival period in thread i is:

$$a_i{}' = \frac{T_i - d \times a_i}{a_i} = \frac{T_i}{a_i} - d \tag{6.2}$$

In other words, the delay size d is just the difference between the real and effective inter-arrival periods.

6.1.3 Implementation

The two core parts of load amplification are arrival points and inserting virtual speedup. Arrival points allow AMP to track the number of task arrivals over time, and AMP uses this count to virtually speed up threads that execute arrival points in order to create the effect of increased load.

Arrival points are implemented much like progress points in COZ, except instead of a single global count of visits, an arrival point tracks per-thread arrivals. This is required because only the thread that executes an arrival point should be virtually sped up with AMP. The first time each thread executes an arrival point, it registers the arrival point with the AMP runtime, which then returns the location of a thread-specific counter. The thread saves this counter location and increments the counter on each visit.

AMP is implemented as an extension to COZ, so much of the remaining functionality is either built on top of COZ's mechanisms or uses them directly. COZ periodically pauses threads to handle samples during program execution. At the end of this handling code, AMP checks the current thread's arrival count and virtually speeds the thread up proportionally to the number of arrivals since the last check. The virtual speedup is inserted using the same two counter protocol used for causal profiling, which means the calculation of effective runtime, optimizations to eliminate unnecessary delays, and the complex handling of blocking/unblocking operations are all inherited from COZ.

The only other change required for load amplification is in profile output; AMP logs the amount of load amplification with each experiment in the causal profile, which allows developers to conduct experiments over a variety of load levels and process the results separately.

One interesting consequence of building AMP on top of COZ is that the two can be used simultaneously; this allows developers to identify potential optimizations that will improve performance under a target load. The following section describes this use of load amplification in detail.

6.2 Causal Profiling with Amplified Load

Deploying software without adequate performance testing is likely to expose unexpected scalability bottlenecks. Load amplification can expose these issues before deployment, but AMP alone does not help developers identify and fix scalability bottlenecks. Combining load amplification with causal profiling makes this possible; running a program with COZ and AMP together gives users a causal profile that shows how much an optimization will improve performance with the program running under higher load. This section describes the interactions between causal profiling and load amplification, guided by the example program in Figure 6.1 in the beginning of this chapter.

Recall that this example program uses a simple producer–consumer model to create and process tasks using a shared work queue. In its initial state, the system is configured to be completely balanced; tasks are completed at the same rate they are created. Increasing the number of consumer threads does not improve performance because there is not enough work to keep additional consumer threads busy. Likewise, improving the performance of the consumer threads would not speed up the program. However, under increased load, the version of the program with two consumers *would* run faster if consumers were optimized. Figure 6.4 shows these results; Both parts of the figure show the effect of optimizing consumer threads (the `t.run()` call on line 17 of `example.cpp`) on program performance. The left plot shows this effect on the program with unmodified load, while the

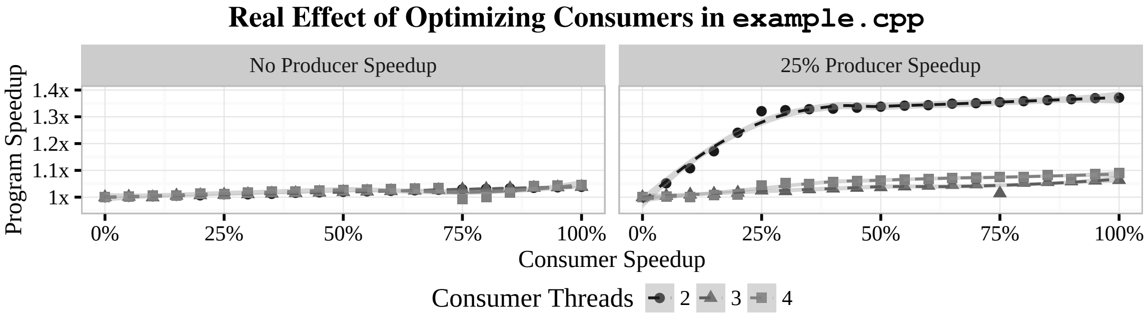

Figure 6.4: The real effect of speeding up consumers in the program from Figure 6.1. The x-axis shows the amount that consumer threads are sped up, where 0% is the original consumer performance and 100% reduces the execution time of each consumer to zero. The y-axis shows the percent increase in program performance. Under the original program load, increasing consumer performance has a negligible effect on program runtime, but under 25% increased load the two consumer version of the program would benefit from improved consumer performance.

right plot shows a different effect when load is increased by 25%. These results were collected by directly modifying the load and consumer execution time, not using AMP and COZ.

These results show that varying load on a program changes the importance of different lines of code. This information would be invaluable to a developer responsible for improving performance in deployment; both conventional and causal profiling would not show that consumer performance is important if profiling was done at the unmodified program load. Without the ability to collect a causal profile at increased load, it is likely this optimization opportunity would simply go unnoticed. Figure 6.5 shows the same information, this time collected using COZ and AMP in combination rather than directly modifying the program to create a real increase in load or implement a real speedup in consumer threads. These results closely match the real effect of varying load and consumer performance.

6.3 Evaluation

In this section, we use AMP to evaluate the scalability the ferret and dedup applications with the performance fixes from Chapter 5. Using AMP, we predict the scalability of each application, and then use AMP and COZ in combination to identify optimization opportunities that would improve the scalability of each application.

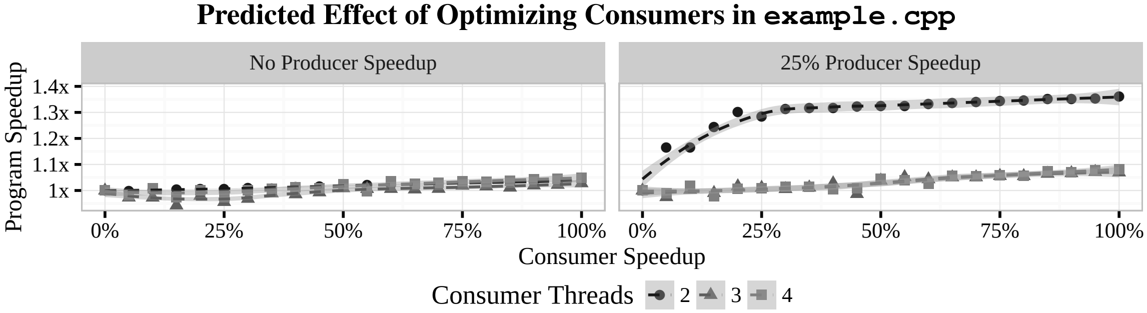

Figure 6.5: The effect of speeding up consumers in the program from Figure 6.1. These predictions are made using load amplification in combination with causal profiling. These results show the same trend as the real effect of speeding up consumers shown in Figure 6.4, but no changes were made to the actual load or consumer performance; both the increased load and sped up consumers were emulated using virtual speedup.

All results in this section were collected on a system with two Intel Xeon E5-2698 v3 CPUs with 16 cores each and 128GB of RAM. Evaluations were run with Intel Turbo Boost disabled, which caps chip frequency at the maximum sustainable frequency rather than allowing CPUs to throttle up and down in response to temperature.

6.3.1 Case Study: Ferret

Ferret is an image similarity search pipeline from the PARSEC benchmark suite, which we first used to evaluate COZ in Chapter 5. The PARSEC driver for ferret provides a series of 3500 images to the application, averaging one image every 4ms. The PARSEC benchmark driver pre-loads inputs, which is unrealistic from a deployment standpoint. Disabling this code increases the image arrival period to 8ms. Using this 8ms inter-arrival period as a baseline, we then predict ferret's scalability using AMP to reduce the inter-arrival period down to 0ms in ten steps. The results of this experiment, shown in Figure 6.6, indicate that ferret scales well up to the original 4ms inter-arrival period, but the program becomes CPU bound when the time between requests goes below 4ms.

To find out how ferret could be modified to improve scalability, we run it with both AMP and COZ. The predicted effect of optimizing most lines is unchanged with increased load, but AMP and COZ identify one significant change at increased load. The ranking stage in ferret's pipeline, specifically line 357 of `ferret-parallel.c`, is bottleneck that was originally uncovered in the evaluation of COZ alone. Figure 6.7 shows the causal profile for this line with and without load

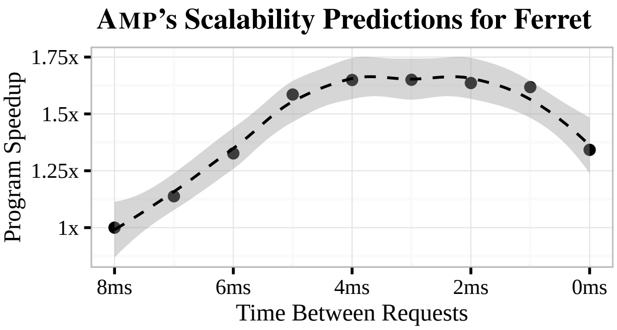

Figure 6.6: AMP's scalability predictions for ferret, after incorporating optimizations from Chapter 5. The x-axis shows the time between requests, which is artificially decreased from 8ms to 0ms using load amplification. The y-axis the program speedup beyond the original performance at 8ms between requests. The results show that ferret is I/O bound up to 4ms between requests, and does not scale beyond this point.

amplification. These results show that ferret's ranking stage becomes a significant bottleneck at increased load. This line is important at both load levels, but the effect of optimizing the line by 40% is nearly doubled at increased load.

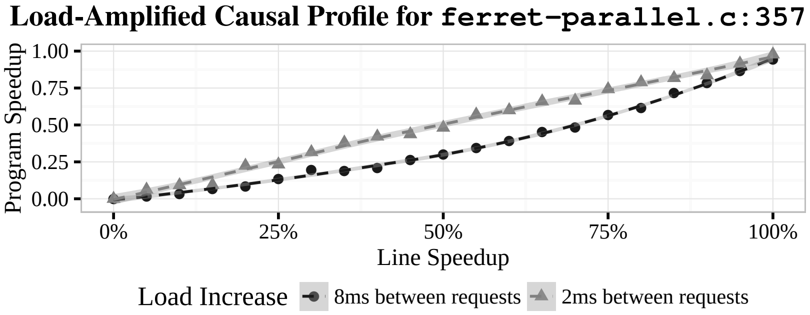

Figure 6.7: The causal profile results for the ranking stage in ferret with and without load amplification. The x-axis shows the speedup line 357 in `ferret-parallel.c`, which dominates the ranking stage. The y-axis shows the effect of optimizing this line of code at each load level. As load increases, this line becomes more and more important; at a load of 2ms between arrivals, the effect of optimizing this line nearly doubles over baseline load up to a 50% line speedup.

6.3.2 Case Study: Dedup

Dedup is a system for lossless compression that eliminates the need to store multiple copies of blocks of data that appear multiple times in the input stream. Chapter 5 introduces dedup, and presents an improvement to dedup's hash function that we incorporated before running dedup with AMP. Figure 6.8 shows AMP's scalability predictions for dedup. The results show that dedup scales

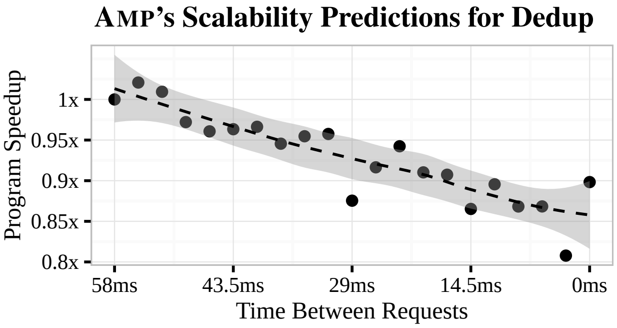

Figure 6.8: AMP's scalability predictions for dedup, after incorporating optimizations from Chapter 5. The x-axis shows the interval at which file chunks are delivered to dedup's pipeline. The baseline of 58ms is the average time between chunks without load amplification when dedup's input pre-loading is disabled. The y-axis shows the program performance, where any value less than 1x corresponds to *slower* execution.

very slightly with increased load, but the unmodified system is already running very near maximum capacity.

Next, we run dedup with AMP and COZ to identify lines where optimizations could improve dedup's scalability. We collect causal profiles at the baseline load of 58ms between chunks and an increased load with just 43.5ms between chunks. Dedup runs near its maximum capacity; most causal profile results are very similar regardless of load. For example, the line `encoder.c:102` is has the largest optimization potential in both profiles. Figure 6.9 shows the causal profile for this line with and without load amplification.

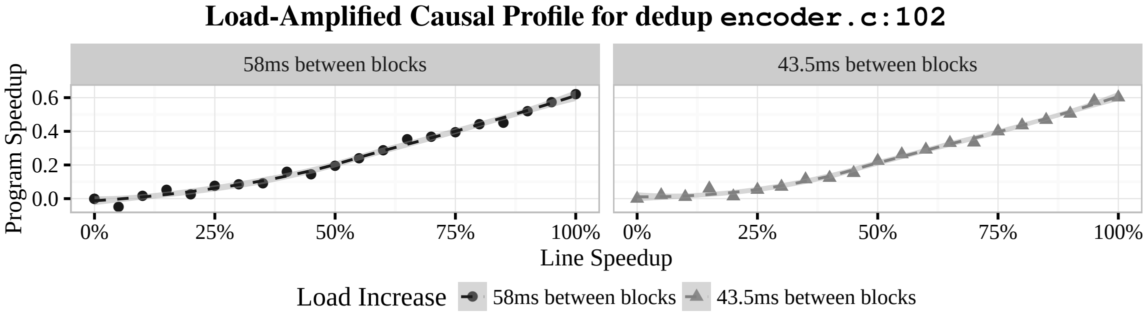

Figure 6.9: Load-amplified causal profile results for `encoder.c:102` in dedup. The x-axis shows the speedup line 102 in `encoder.c`, which compresses a block of data using zlib. The y-axis shows the effect of optimizing this line of code at each load level. The baseline load on dedup is enough to make the program primarily CPU-bound. As a result, this source line has roughly the same importance regardless of load on the program.

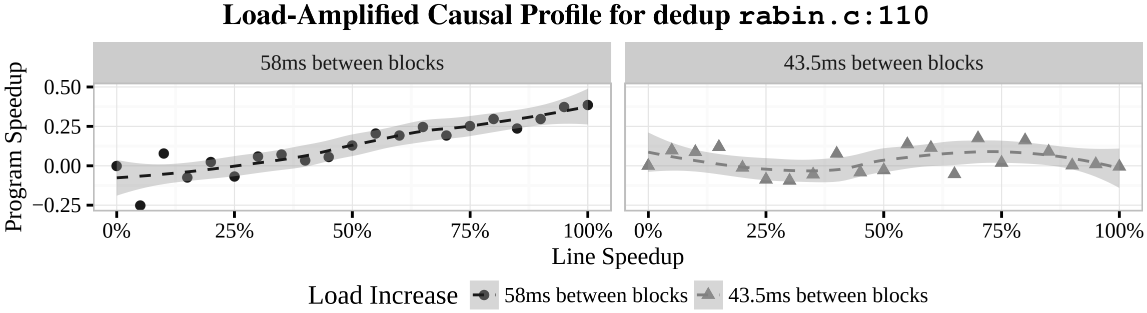

Figure 6.10: Load-amplified causal profile results for `rabin.c:110` in dedup. The x-axis shows the speedup of this line, which computes a Rabin fingerprint of each incoming chunk of data. The y-axis shows the effect of optimizing this line of code at each load level. At the baseline load, speeding up this line effectively increases the rate at which blocks are moved through the deduplication pipeline. However, at increased load the pipeline is already saturated so the line no longer has any optimization potential.

While `encoder.c:102` dominates the causal profile regardless of load, there are two smaller changes with increasing load. Line 110 of `rabin.c`, which computes a Rabin fingerprint of each chunk of data before compressing it, has some optimization potential at baseline load but is no longer important at increased load. Figure 6.10 shows the causal profiles for this line with and without load amplification. The diminished impact of this line at increased load makes sense; computing fingerprints faster increases the rate at which chunks of data can be provided to the compression pipeline, but increased load accomplishes the same thing. This pipeline is already saturated at increased load, so the optimization is no longer important.

The other line of interest is the same hash table traversal code that was optimized using COZ, `hashtable.c:220`. Again, this line has significantly less optimization potential at increased load. Figure 6.11 shows results for this line. Optimizing this line will still help performance, but the effect is roughly halved at the increased load.

6.3.3 Evaluation Summary

This section presented an analysis of the scalability of both ferret and dedup. We find that ferret is already running at or above peak performance on test inputs, and will not scale to handle additional load. Dedup will scale slightly at higher load (under 2.5% increased throughput). A further examination with AMP and COZ in combination shows that a reallocation of threads in ferret would

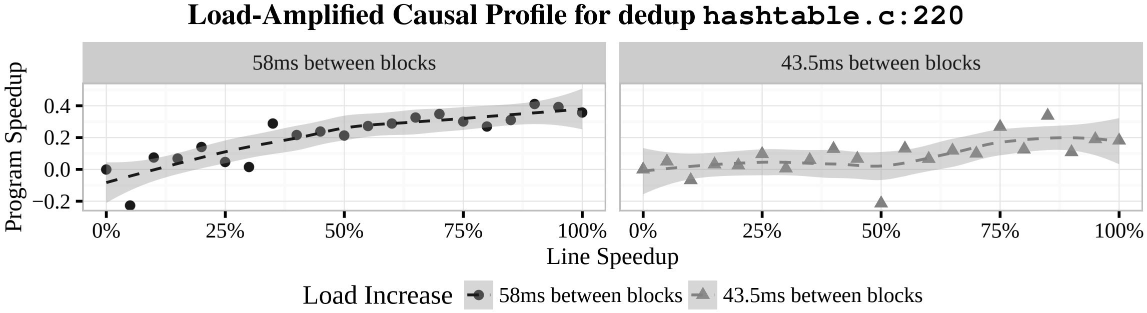

Figure 6.11: Load-amplified causal profile results for `hashtable.c:220` in dedup. The x-axis shows the speedup of this line, which traverses elements in the same bucket of dedup's hash table. The y-axis shows the effect of optimizing this line of code at each load level. As with the previous line, the optimization potential of this line decreases significantly at increased load.

improve scalability. Noisy profile results for dedup hint at a potential optimization opportunity in the hash table implementation, and show that enqueueing elements into the compression stage's work queue becomes a bottleneck at higher load.

6.4 Conclusion

This chapter introduced load amplification, a technique that allows developers to predict the scalability of their software and, in combination with causal profiling, identify code where optimizations will improve scalability. Load amplification makes it possible to evaluate software scalability before deployment, even when it is difficult or impossible to generate realistic load on a system in a test environment. To demonstrate the effectiveness of this approach, we presented AMP, an extension to COZ that implements load amplification. We use AMP to predict the scalability of two applications, ferret and dedup, and use COZ with AMP to collect load-amplified causal profiles. Developers can use load-amplified causal profiles to rule out potential optimizations that would only have an effect at low load, and to identify scalability bottlenecks that appear under high load.

CHAPTER 7

RELATED WORK

This chapter describes relevant prior work in performance evaluation, debugging, and prediction.

7.1 Performance Evaluation

Mytkowicz et al. observe that environmental sensitivities can degrade program performance by as much as 300% [62]. While Mytkowicz et al. show that layout can dramatically impact performance, their proposed solution, *experimental setup randomization* (the exploration of the space of different link orders and environment variable sizes), is substantially different.

Experimental setup randomization requires far more runs than STABILIZER, and cannot eliminate bias as effectively. For example, varying link orders only changes inter-module function placement, so that a change of a function's size still affects the placement of all functions after it. STABILIZER instead randomizes the placement of every function independently. Similarly, varying environment size changes the base of the process stack, but not the distance between stack frames.

In addition, any unrandomized factor in experimental setup randomization, such as a different shared library version, could have a dramatic effect on layout. STABILIZER does not require *a priori* identification of all factors. Its use of dynamic re-randomization also leads to normally distributed execution times, enabling the use of parametric hypothesis tests.

Alameldeen and Wood find similar sensitivities in processor simulators, which they also address with the addition of non-determinism [3]. Tsafrir, Ouaknine, and Feitelson report dramatic environmental sensitivities in job scheduling, which they address with a technique they call "input shaking" [81, 82]. Georges et al. propose rigorous techniques for Java performance evaluation [30]. While prior techniques for performance evaluation require many runs over a wide range of (possibly unknown) environmental factors, STABILIZER enables efficient and statistically sound performance evaluation by breaking the dependence between experimental setup and program layout.

	Randomization			**Implementation**		
System	*code*	*stack*	*heap*	*recompilation*	*dynamic*	*re-randomization*
ASLR [79, 60]		✓	✓		✓	
TRR [88]		✓	✓		✓	
ASLP [48]	✓+	✓	✓		✓	
Address Obfuscation [13]	✓+	✓+	✓	✓	✓-	
Dynamic Offset Randomization [87]	✓±			✓	✓	
Bhatkar, Sekar, and DuVarney [14]	✓+	✓+	✓	✓	✓	
DieHard [11]			✓+		✓	✓
Stabilizer	✓+	✓+	✓+	✓	✓	✓

Table 7.1: Comparison of Stabilizer to prior work in program layout randomization. Prior work in layout randomization includes varying degrees of support for the randomizations implemented in Stabilizer. The features supported by each system are indicated with a checkmark. A ✓- indicates limited support for the corresponding feature. A ✓+ under the "Randomization" columns indicate support for *re-randomization*, and ✓± indicates limited support for re-randomization.

7.2 Program Layout Randomization

Nearly all prior work in program randomization has focused on security concerns. Randomizing the addresses of program elements makes it difficult for attackers to reliably trigger exploits. Table 7.1 gives an overview of prior work in program layout randomization.

7.2.1 Randomization for Security

The earliest implementations of layout randomization, Address Space Layout Randomization (ASLR) and PaX, relocate the heap, stack, and shared libraries in their entirety [79, 60]. Building on this work, Transparent Runtime Randomization (TRR) and Address Space Layout permutation (ASLP) have added support for randomization of code or code elements (like the global offset table) [88, 48]. Unlike Stabilizer, these systems relocate entire program segments.

Fine-grained randomization has been implemented in a limited form in the Address Obfuscation and Dynamic Offset Randomization projects, and by Bhatkar, Sekar, and DuVarney [13, 87, 14]. These systems combine coarse-grained randomization at load time with finer granularity randomizations in some sections. These systems do not re-randomize programs during execution, and do not apply fine-grained randomization to every program segment. Stabilizer randomizes code and data at a fine granularity, and re-randomizes during execution.

DieHard uses heap randomization to prevent memory errors [11]. Placing heap objects randomly makes it unlikely that use after free and out of bounds accesses will corrupt live heap data. DieHarder builds on this to provide probabilistic security guarantees [65]. STABILIZER can be configured to use DieHard as its substrate, although this can lead to substantial overhead.

7.2.2 Predictable Performance

Quicksort is a classic example of using randomization for predictable performance [39]. Random pivot selection drastically reduces the likelihood of encountering a worst-case input, and converts a $O(n^2)$ algorithm into one that runs with $O(n \log n)$ in practice.

Randomization has also been applied to probabilistically analyzable real-time systems. Quiñones et al. show that random cache replacement enables probabilistic worst-case execution time analysis, while maintaining good performance. This probabilistic analysis is a significant improvement over conventional hard real-time systems, where analysis of cache behavior relies on complete information.

7.3 Software Profiling

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.

7.3.1 General-Purpose Profilers

General-purpose profilers are typically implemented using instrumentation, sampling, or both. Systems based on sampling (including causal profiling) can arbitrarily reduce *probe effect*, although sampling must be unbiased [63].

The UNIX prof tool and oprofile both use sampling exclusively [80, 51]. 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 [33]. Gprof produces a call graph profile, which counts invocations of functions segregated by caller. Cho, Moseley, et al. reduce the overhead of Gprof's call-graph profiling by interleaving instrumented and un-instrumented execution [20]. Path profilers add further detail, counting executions of each path through a procedure, or across procedures [9, 5].

7.3.2 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.

7.3.2.1 Critical Path Profiling

IPS uses traces from message-passing programs to identify the critical path, and reports the amount of time each procedure contributes to the critical path [59]. IPS-2 extends this approach with limited support for shared memory parallelism [89, 57]. Other critical path profilers rely on languages with first-class threads and synchronization to identify the critical path [38, 68, 73]. Identifying the critical path helps developers find code where optimizations will have some impact, but these approaches to not give developers any information about how much performance gain is possible before the critical path changes. Hollingsworth and Miller introduce two new metrics to approximate 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 [40]. These metrics are similar to the optimization potential measured by a causal profiler, but can only be computed with a complete program activity graph. Collection of a program activity graph is costly, and could introduce significant probe effect.

7.3.2.2 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 [35]. 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 extend this approach further to support Cilk programs, with an added category for time spent managing parallelism [74]. The WAIT tool adds fine-grained categorization to identify bottlenecks in large-scale production Java systems [4]. Unlike causal profiling, these profilers can only capture interference between threads that directly affects their scheduler state.

7.3.2.3 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 for later analysis [47, 58]. Aguilera, Mogul et al. use network-level tracing to identify probable "causal paths" that may be responsible for high latency in distributed systems of black boxes [2]. AppInsight uses a similar technique to identify sources of latency in mobile application event handlers [70].

Tracing is only practical in domains where thread interaction is limited. Program threads may interact not only via synchronization operations but also due to cache coherence protocols, race conditions, scheduling dependencies, lock-free algorithms, and I/O, all of which causal profiling implicitly accounts for.

7.3.3 Performance Guidance and Experimentation

Several systems have employed delays to extract information about program execution times. Mytkowicz et al. use delays to validate the output of profilers on single-threaded Java programs [63]. Snelick, JáJá et al. use delays to profile parallel programs [71]. This approach measures the effect of slowdowns in combination, which requires a complete execution of the program for each of an exponential number of configurations. Active Dependence Discovery (ADD) introduces performance perturbations to distributed systems and measures their impact on response time [17]. ADD requires a complete enumeration of system components, and requires developers to insert performance perturbations manually. Gunawi, Agrawal et al. use delays to identify causal dependencies between events in the EMC Centera storage system to analyze Centera's protocols and policies [34]. Song and Lu use machine learning to identify performance anti-patterns in source code [72]. Unlike causal profiling, these approaches do not predict the effect of potential optimizations.

7.4 Scalability and Bottleneck Identification

Several approaches have used hardware performance counters to identify hardware-level performance bottlenecks [56, 18, 23]. Techniques based on binary instrumentation can identify cache and heap performance issues, contended locks, and other program hotspots [64, 8, 53]. ParaShares and Harmony identify basic blocks that run during periods with little or no parallelism [45, 44].

Code identified by these tools is a good candidate for parallelization or classic serial optimizations. Bottlenecks, a profile analysis tool, uses heuristics to identify bottlenecks using call-tree profiles [6]. Given call-tree profiles for different executions, Bottlenecks can pinpoint which procedures are responsible for the difference in performance. The FreeLunch profiler and Visual Studio's contention profiler identify locks that are responsible for significant thread blocking time [21, 32]. BIS uses similar techniques to identify highly contended critical sections on asymmetric multiprocessors, and automatically migrates performance-critical code to faster cores [43]. Bottle graphs present thread execution time and parallelism in a visual format that highlights program bottlenecks [24]. Unlike causal profiling, these tools do not predict the performance impact of removing bottlenecks. All these systems can only identify bottlenecks that arise from explicit thread communication, while causal profiling can measure parallel performance problems from any source, including cache coherence protocols, scheduling dependencies, and I/O. Load amplification takes this one step further, enabling developers to identify bottlenecks that would otherwise only be detected after deployment.

7.4.1 Profiling for Parallelization and Scalability

Several systems have been developed to measure potential parallelism in serial programs [83, 90, 29]. Like causal profiling, these systems identify code that will benefit from developer time. Unlike causal profiling, these tools are not aimed at diagnosing performance issues in code that has already been parallelized.

Kulkarni, Pai, and Schuff present general metrics for available parallelism and scalability [49]. The Cilkview scalability analyzer uses performance models for Cilk's constrained parallelism to estimate the performance effect of adding additional hardware threads [37].

Causal profiling alone can only detect performance problems that result from poor scaling on the current hardware platform, while load amplification with AMP allows developers to predict scalability and pinpoint bottlenecks before deploying their software. Unlike the above systems, which look only for code where parallelization appears to be possible, COZ and AMP do not tell developers *how* to optimize code, just that it would be beneficial if optimizations are possible. Limiting the profiling to code that can be parallelized may miss important optimization opportunities; in fact, none of the optimizations we implemented using COZ's results involved parallelizing serial code.

CHAPTER 8
CONCLUSION

Decades of improvements in processor speeds have allowed developers to largely ignore performance. Instead of improving their code, they could simply wait for the next generation of processors to come out and automatically "optimize" their programs. This rapid progress led to incredible growth in the capabilities and uses for software. Unfortunately, this era has come to an end and performance is once again a first-class concern. Instead of increasing clock speeds, new processors gain additional support for parallelism and hardware complexity targeted at improving performance.

The tools and practices for performance analysis and debugging largely come from an era of simpler hardware and software: techniques for measuring performance do not account for the inherent unpredictability of performance on modern processors; software profilers do not guide developers to code where local performance improvements will have a real effect on system performance; and methods for evaluating software in deployment do not consider the effects of scalability in deployment.

This dissertation introduces a novel approach to performance analysis and debugging, implemented in three systems: STABILIZER enables both predictable performance in deployment and statistically sound performance evaluation by controlling performance variability; COZ guides developers to code where optimizations will have the largest effect; and AMP allows developers to evaluate system scalability in a testing environment and guides developers to optimizations that will improve scalability. These systems give developers the information they need to effectively measure, control, and improve the performance of their software.

BIBLIOGRAPHY

[1] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, September 2015.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89. ACM, 2003.

[3] A. Alameldeen and D. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 7–18. IEEE Computer Society, 2003.

[4] E. R. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA*, pages 739–753. ACM, 2010.

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

[6] G. Ammons, J.-D. Choi, M. Gupta, and N. 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] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. In *SIGMETRICS*, pages 115–125, 1990.

[8] M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with Pin. *Computer*, 43(3):34–41, Mar. 2010.

[9] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, pages 46–57, 1996.

[10] L. E. Bassham, III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo. SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2010.

[11] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 158–168. ACM, 2006.

[12] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing High-Performance Memory Allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124. ACM, 2001.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 8–8. USENIX Association, 2003.

[14] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *SSYM '05: Proceedings of the 14th Conference on USENIX Security Symposium*, pages 271–286, Berkeley, CA, USA, 2005. USENIX Association.

[15] S. M. Blackburn, A. Diwan, M. Hauswirth, A. M. Memon, and P. F. Sweeney. Workshop on Experimental Evaluation of Software and Systems in Computer Science (Evaluate 2010). In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 291–292. ACM, 2010.

[16] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, et al. TR 1: Can You Trust Your Experimental Results? Technical report, Evaluate Collaboratory, 2012.

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

[18] M. Burtscher, B.-D. Kim, J. R. Diamond, J. D. McCalpin, L. Koesterke, and J. C. Browne. PerfExpert: An easy-to-use performance diagnosis tool for HPC applications. In *SC*, pages 1–11. IEEE, 2010.

[19] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. Proartis: Probabilistically analyzable real-time systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s):94:1–94:26, May 2013.

[20] H. K. Cho, T. Moseley, R. E. Hank, D. Bruening, and S. A. Mahlke. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *CGO*, pages 1–10. IEEE Computer Society, 2013.

[21] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*, pages 291–307, 2014.

[22] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 261–269. ACM, 1990.

[23] J. R. Diamond, M. Burtscher, J. D. McCalpin, B.-D. Kim, S. W. Keckler, and J. C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *ISPASS*, pages 32–43. IEEE Computer Society, 2011.

[24] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *OOPSLA*, pages 355–372, 2013.

[25] C. Dumitrescu, I. Raicu, M. Ripeanu, and I. T. Foster. Diperf: An automated distributed performance testing framework. In R. Buyya, editor, *5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings*, pages 289–296. IEEE Computer Society, 2004.

[26] R. Durstenfeld. Algorithm 235: Random Permutation. *Communications of the ACM*, 7(7):420, 1964.

[27] W. Feller. *An Introduction to Probability Theory and Applications*, volume 1. John Wiley & Sons Publishers, 3rd edition, 1968.

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

[29] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI*, pages 458–469. ACM, 2011.

[30] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 57–76. ACM, 2007.

[31] R. Giladi and N. Ahituv. SPEC as a performance evaluation measure. *IEEE Computer*, 28(8):33–42, 1995.

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

[33] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

[34] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing commodity storage clusters. In *ISCA*, pages 60–71. IEEE Computer Society, 2005.

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

[36] G. Hamerly, E. Perelman, J. Lau, B. Calder, and T. Sherwood. Using Machine Learning to Guide Architecture Simulation. *Journal of Machine Learning Research*, 7:343–378, Dec. 2006.

[37] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156. ACM, 2010.

[38] J. M. D. Hill, S. A. Jarvis, C. J. Siniolakis, and V. P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *PDP*, pages 286–294, 1998.

[39] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

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

[41] Intel. *Intel VTune Amplifier 2015*, 2014.

[42] D. A. Jiménez. Code Placement for Improving Dynamic Branch Prediction Accuracy. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 107–116. ACM, 2005.

[43] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, pages 223–234. ACM, 2012.

[44] M. Kambadur, K. Tang, and M. A. Kim. Harmony: Collection and analysis of parallel block vectors. In *ISCA*, pages 452–463. IEEE Computer Society, 2012.

[45] M. Kambadur, K. Tang, and M. A. Kim. Parashares: Finding the important basic blocks in multithreaded programs. In *Euro-Par*, Lecture Notes in Computer Science, pages 75–86, 2014.

[46] kernel.org. *perf: Linux profiling with performance counters*, 2014.

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

[48] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 339–348. IEEE Computer Society, 2006.

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

[50] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004*, pages 75–86. IEEE Computer Society, 2004.

[51] J. Levon and P. Elie. oprofile: A system profiler for Linux. http://oprofile.sourceforge.net/, 2004.

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

[53] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[54] G. Marsaglia. Random Number Generation. In *Encyclopedia of Computer Science, 4th Edition*, pages 1499–1503. John Wiley and Sons Ltd., Chichester, UK, 2003.

[55] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 79–86. IEEE Computer Society, 2004.

[56] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[57] B. P. Miller, M. Clark, J. K. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel Distributed Systems*, 1(2):206–217, 1990.

[58] B. P. Miller, C. Macrander, and S. Sechrest. A distributed programs monitor for berkeley UNIX. In *ICDCS*, pages 43–54, 1985.

[59] B. P. Miller and C.-Q. Yang. IPS: An interactive and automatic performance measurement tool for parallel and distributed programs. In *ICDCS*, pages 482–489, 1987.

[60] I. Molnar. Exec-Shield. `http://people.redhat.com/mingo/exec-shield/`.

[61] D. A. Moon. Garbage Collection in a Large LISP System. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 235–246. ACM, 1984.

[62] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proceedings of the 14th Internation Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276. ACM, 2009.

[63] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *PLDI*, pages 187–197. ACM, 2010.

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

[65] G. Novark and E. D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[66] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. *Communications of the ACM*, 51(12):87–95, 2008.

[67] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun. Mining precise performance-aware behavioral models from existing instrumentation. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 484–487. ACM, 2014.

[68] Y. Oyama, K. Taura, and A. 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.

[69] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 412–423, New York, NY, USA, 2007. ACM.

[70] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *OSDI*, pages 107–120. USENIX Association, 2012.

[71] R. Snelick, J. JáJá, R. Kacker, and G. Lyon. Synthetic-perturbation techniques for screening shared memory programs. *Software Practice & Experience*, 24(8):679–701, 1994.

[72] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014.

[73] Z. Szebenyi, F. Wolf, and B. J. N. Wylie. Space-efficient time-series call-path profiling of parallel applications. In *SC*. ACM, 2009.

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

[75] The Chromium Project. Performance Dashboard. `http://build.chromium.org/f/chromium/perf/dashboard/overview.html`.

[76] The LLVM Team. Clang: a C Language Family Frontend for LLVM. `http://clang.llvm.org`, 2012.

[77] The LLVM Team. Dragonegg - Using LLVM as a GCC Backend. `http://dragonegg.llvm.org`, 2013.

[78] The Mozilla Foundation. Buildbot/Talos. `https://wiki.mozilla.org/Buildbot/Talos`.

[79] The PaX Team. The PaX Project. `http://pax.grsecurity.net`, 2001.

[80] K. Thompson and D. M. Ritchie. *UNIX Programmer's Manual*. Bell Telephone Laboratories, 1975.

[81] D. Tsafrir and D. Feitelson. Instability in Parallel Job Scheduling Simulation: the Role of Workload Flurries. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE Computer Society, 2006.

[82] D. Tsafrir, K. Ouaknine, and D. G. Feitelson. Reducing Performance Evaluation Sensitivity and Variability by Input Shaking. In *Proceedings of the 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 231–237. IEEE Computer Society, 2007.

[83] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP*, pages 185–196. ACM, 2008.

[84] C. Watson, S. Emmons, and B. Gregg. A miscroscope on microservices. http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html, 2015.

[85] F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[86] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. *Lecture Notes in Computer Science*, 986, 1995.

[87] H. Xu and S. J. Chapin. Improving Address Space Randomization with a Dynamic Offset Randomization Technique. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 384–391. ACM, 2006.

[88] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 260–269. IEEE Computer Society, 2003.

[89] C.-Q. Yang and B. P. Miller. Performance measurement for parallel and distributed programs: A structured and automatic approach. *IEEE Transactions on Software Engineering*, 15(12):1615–1629, 1989.

[90] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO*, pages 47–58. IEEE Computer Society, 2009.